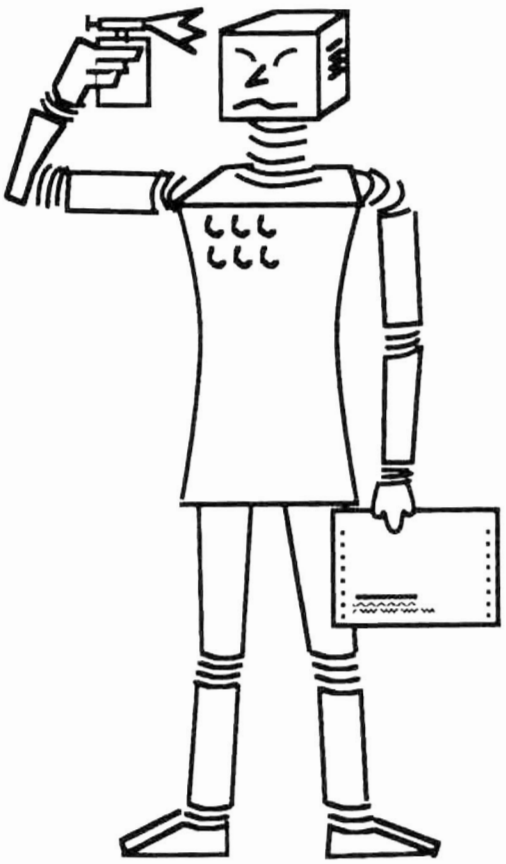
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT



Simplification and Reduction for Automated Theorem Proving

Rolf Socher-Ambrosius

SEKI Report SR-90-10

Simplification and Reduction for Automated Theorem Proving

Rolf Socher-Ambrosius
Fachbereich Informatik, Universität Kaiserslautern
Postfach 30 49, D-6750 Kaiserslautern, W.-Germany

This work was supported by the Deutsche Forschungsgemeinschaft, SFB 314.

Simplification and Reduction for Automated Theorem Proving

Vom Fachbereich Informatik der Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von

Dipl.-Math. Rolf Socher-Ambrosius

Datum der wissenschaftlichen Aussprache:	13. Juni 1990
Berichterstatter:	Prof. Dr. Jörg Siekmann
	Prof. Dr. Jürgen Avenhaus
Dekan:	Prof. Dr. Gerhard Zimmermann

D 386

Danksagung

Ich möchte mich bei allen bedanken, die zum Entstehen dieser Arbeit beigetragen haben.

Mein besonderer Dank gilt Professor Jörg Siekmann und Norbert Eisinger. Jörg hat mich auf den Geschmack der KI und des automatischen Beweisens gebracht, und mit seinem Optimismus und mit viel Zuspruch hat er mich mehrfach „über die Runden gerettet". Er hat die „Markgraf Karl" Beweisergruppe aufgebaut, die eine ideale Umgebung für die Entstehung dieser Arbeit war. Von Norbert habe ich mit viel Mühe und einigen Fehlschlägen gelernt, wie man wissenschaftliche Texte schreibt. Seine Geduld und auch sein schier unerschöpflicher Erfahrungsschatz im Umgang mit Klauseln, Graphen und dergleichen waren mir eine große Hilfe, die ich an dieser Stelle gar nicht ausreichend würdigen kann. Für jede meiner schlechten Ideen hatte er ein passendes Gegenbeispiel parat.

Bedanken möchte ich mich auch bei Professor Jürgen Avenhaus für seine Unterstützung und seine Verbesserungsvorschläge, insbesondere zur Gestaltung des Anhangs.

Walter Olthoff hat mir geholfen, die Fruchtlosigkeit eines früheren Dissertationsthemas zu erkennen, und Manfred Schmidt-Schauß und Hans Jürgen Ohlbach haben dann für das endgültige Thema der Arbeit die entscheidenden Impulse gegeben.

Für eine Zeit von zwei Jahren ist diese Dissertation von einem Projekt der IBM Deutschland unterstützt worden. Den Initiatoren möchte ich hiermit danken.

Schließlich möchte ich noch Larry Wos' Buch „Automated Reasoning: 33 Basic Research Problems" erwähnen. Die entscheidenden Anstöße zu den wichtigen Kapiteln 5 und 6 dieser Arbeit kamen aus der Lektüre dieses lesenswerten Buches.

Abstract

The most severe obstacle on the way to the efficient automation of theorem proving is the size of the search space for drawing new inferences. There are two well known ways to overcome this difficulty. One solution comes under the term "refutation strategies", which denotes techniques to choose candidates for the next inference step. The other solution is termed "reduction", which subsumes all techniques to remove those elements of the search space that do not contribute to the solution and thus are redundant.

The second approach's most critical part is the test on redundancy. Since each element of the search space has to be subjected to such a test, its efficiency is crucial for the value of the reduction approach. Subsumption, being one of the most important types of redundancy, is also a most problematic one. In this thesis, new and efficient tests for the variant and the subsumption property are developed, both based on the well known algorithms for detecting isomorphism of directed graphs.

A most undesired aspect of redundancy is the derivation of subsumed clauses. Besides the problem with the subsumption test, the amount of computer time, which is spent for the derivation and normalization of such a clause, is purely wasted. In this thesis, the two approaches, strategy and reduction, are combined by a strategy to decrease the number of redundant information derived. This strategy is heavily based on a special treatment of logical equivalence. It turns out that this strategy represents a first step towards the answer of several open questions in automated theorem proving, like the problem with the derivation of redundant clauses, the choice of the appropriate representation and inference rule, the question for a theory to demodulate on the literal level, and finally the choice of clauses to apply a given inference rule. These problems are discussed in Wos' (1988) 33 Basic Research Problems.

Contents

1 Introduction

The use of the two notions *simplification* and *reduction* for automated reasoning is not a homogenous one. Mostly workers in the field do not even clearly distinguish between the two words. The two notions' most basic common feature can best be seen by comparing them with *deduction*. Deduction is considered mainly a mechanism for deriving new knowledge from a given knowledge base, in order to find a problem's solution. Simplification and reduction, on the other side, are both regarded in this thesis as first and foremost a means to find as simple as possible a representation for given information. This does not contradict the fact that under certain circumstances both simplification and reduction are able to find trivial solutions of their own. The difference between the two techniques will be seen rather as a technical than a conceptual one.

1.1 Simplification

Simplification is usually understood to be part of the preprocessing of formulae, which takes place before they are subjected to some automated reasoning program. Historically, simplification developed from serious problems with proving verification conditions. Although these formulae are usually of a very trivial nature, their structure and size makes them difficult to prove, if not unprovable in practice, for general purpose reasoning programs. Fast and efficient simplification procedures, which are tailored for particular classes of objects, are widely accepted to be a solution to this problem. Their application should result in a formula, which is exempted from all easy to solve parts, such that the general prover's comparably inefficient mechanisms have to deal only with the really hard kernel of the problem. In particular this implies that simplifiers should be able to find proofs of their own for trivial theorems.

Simplification procedures are always bound to a particular theory like arithmetic, theories of orderings or theories of data structures. The simplification task can very generally be described as the task to transform a term of the underlying theory with given rules into a simpler equivalent term. Thus the notions of *transformation rules*, *equivalence* and *simplicity* are

relevant in this context. The transformation rules mostly derive from the equations that define the theory. The equivalence relation is just equality in the underlying theory. More problems, however, come along with the notion of simplicity. A formal definition seems difficult, since the alleged simplicity of terms heavily depends on their use for subsequent algorithms. The following general criteria for simplicity may be interesting: the length of the term, its "readability" or its nesting depth. For theories with canonical term rewriting systems, however, the simplicity problem can easily be bypassed. Canonical forms are widely accepted as simplest forms, even if the criteria mentioned above do not apply. For instance, the arithmetical expression $(x+1)^3$ is shorter and even better "readable" than its canonical form x^3+3x^2+3x+1.

Procedures that simplify formulae on the basis of their Boolean properties, can be seen as theory simplifiers under the theory of Boolean algebra. According to their nature, these procedures are more closely related to reduction procedures than any other theory simplification. This thesis will only deal with what we call Boolean simplification. This notion subsumes all techniques that transform formulae given in prenex negation normal form[1] along the rules of Boolean algebra.

Boolean simplification originated in the 1950s in connection with the problem of minimizing the number of components of a given switching circuit. Although the theory of Boolean algebra does not admit a canonical reduction system (Hullot 1980), it admits a certain *normal form* in the following sense: Instead of a canonical rewriting system, there exists an algorithm that transforms each Boolean algebra term into a unique form. This uniquely determined form is called the *set of prime implicants* and can be given either as a conjunction of disjunctions, or as a disjunction of conjunctions. For purposes of automated reasoning, the clausal form, that is a conjunction of disjunctions, is usually preferred. A number of algorithms to obtain the set of prime implicants of a given formula has been developed,

---

[1] Prenex negation normal form denotes a form $\forall x_1 \ldots x_n$ M, where M is a quantifierfree formula containing only the connectives &, $\vee$ and $\neg$, and negation is moved directly in front of the literals.

for instance by Quine (1952) and (1959), Slagle, Chang & Lee (1970), and Tison (1969). In the context of theorem proving, the problem of minimizing Boolean expression comes along with the multiplication of formulae into clausal form. Most theorem provers require the conversion of formulae to be proved into clausal form. This is the case for resolution based systems (Robinson 1965), for matrix methods (Bibel 1981, Andrews 1981), as well as for completion theorem proving (Hsiang 1982 and 1985). Although some efforts have been spent on developing non-clausal methods, for instance non-clausal resolution by Murray & Rosenthal (1987), or non-clausal completion theorem proving by Hsiang (1985), as well as matrix methods for negation normal form (andrews 1981), they all suffer from a serious drawback, namely the problem with unifying whole formulae instead of literals. The importance of an efficient clausal form transformation thus becomes obvious, in particular, when the shortcomings of the naive approach are taken into account. The transformation's most critical step is the multiplication of a nested sequence of conjunctions and disjunctions into clausal form. This step can result in an exponential increase of formulae (Eisinger & Weigele 1983). The difficulties to prove the formula of Andrew's example (see Henschen 1980) are caused by this inflation, since here it is the pure number of clauses produced by the clausal form transformation that constitute the problem. For this example the number of clauses produced can range from 128 to over 16000, depending on how the transformation is performed.

Most of the known algorithms to produce the prime implicants require the formula already given in clausal form. In view of the problems coming with the clausal form transformation, however, an integration of the Boolean minimization techniques into the clausal form transformation seems far more favourable. In other words, the clausal form transformation should be organized in such a way that it already produces the prime implicant form. This idea has its origins in Slagle, Chang & Lee's (1970) algorithm, and it is elaborated in chapter 3 of this thesis. The algorithm presented in this chapter is based on a matrix method very similar to Andrew's and Bibel's techniques for proving first order formulae. It takes as input a formula in negation normal form, and generates the prime implicants of this formula. This approach follows the strategy to avoid the generation of

redundancy (in our particular case during the clausal form transformation) instead of removing it after its derivation, an idea, which is even more emphasized in chapter 5 of this thesis.

1.2 Reduction

Reduction is regarded as part of the inference system. While deduction infers new information from given one, reduction is the part that removes redundant information from a given set and thus helps to keep the search space small. Experience with automated theorem provers has shown that the derivation of redundant information is one of the greatest obstacles to the efficiency of reasoning programs. As is the case with simplification, reduction is required to be fast and efficient, since after each step that infers new information, each element of the actual problem set is potentially redundant and has thus to be subjected to the reduction procedures.

Two basic types of redundant information face automated reasoning systems: Information can be redundant on account of its *logical* form, which means that it can be removed without changing the logical value of the given information. On the other hand, information can also be redundant, if it cannot contribute to a proof. From the second type, only the so called *purity check* has gained some attention, however, without posing greater theoretical difficulties. In this thesis, only the first type of redundancy will be considered.

For resolution based reasoning systems, reduction traditionally is an operation that discards redundant clauses from a set of clauses, while preserving the logical value of this clause set. The very general idea is the removal of any clause that is logically implied by another already present clause. In this generality, however, reduction is infeasible, since it is undecidable, whether one clause implies another (Schmidt-Schauß 1986). Two syntactic concepts stronger than implication are commonly employed in most reasoning systems, and most other types of reduction, like replacement factoring or replacement resolution (Markgraf 1984) are refinements or derivatives of these two. One is *tautology*, and the other is *subsumption* (Robinson 1965). While tautology is most simple a concept, and being easily recognized, subsumption is far more intricate, yet also far

more powerful. There is a broad accordance that subsumption is essential to solving more complicated problems (see for instance Wos (1988), Eisinger (1981), or Markgraf (1984)). However, the subsumption test is rather expensive, and it has to be repeated over and over again during a refutation. The efficiency of the subsumption test thus seems decisive for its use.

Not always, however, is the use of a complete subsumption test essential for completing proofs. There are examples, where no subsumed clauses are generated at all (for instance, during the proof that some formula represents a shortest axiom of equivalential calculus, subsumption did not take place at all, see Wos et al (1984)). There are other examples, where only particular types of subsumption occur. A frequently occurring type of subsumption is that where two clauses are variants of each other, that is, they are just identical up to renaming of variables. In particular, the clausal form transformation tends to produce this type of redundancy. In chapter 4 it is shown that the decision problem, whether a given clause is a variant of another, amounts to a generalization of the graph isomorphism problem. Using the well-known graph technique of characteristic matrices, the variant test represents an efficient means to test the variant relation between clauses. Moreover, this test can be generalized in a straightforward way to a subsumption test that uses characteristic matrices to improve the inherently exponential merging procedure, which is usually required by subsumption algorithms.

Subsumption is usually classified into what is called backward subsumption and forward subsumption (Overbek 1975). Backward subsumption is a process for discarding already retained clauses, when a new clause is derived that subsumes it. Forward subsumption is the process that removes a newly generated clause because it is subsumed by another, already retained clause. Backward subsumed clauses are nearly inevitable in any refutation. For instance, any possible resolution step between clauses of the set {PQ, ¬PQ, P¬Q, ¬P¬Q} derives a unit clause that (backward) subsumes two already present clauses. Forward subsumed clauses, on the other hand, represent a highly undesirable derivation, since the whole time required to perform the resolution step, including unification, and to search the whole database for the appropriate clauses to apply the inference rule is wasted with deriving an unneeded result. Moreover, newly generated clauses must

be processed with demodulation, term simplification, and various other standard procedures, before they are recognized to be redundant. Finally, the test on subsumption itself is rather expensive, as already mentioned. Altogether, a strategy to prevent the derivation of such redundant information would thus obviously be better than the removal of redundancy a posteriori. It seems, however, that developing such a strategy requires a deep understanding on how clauses can be derived that are subsumed by already retained clauses.

In chapter 5 we isolate two typical structures that would systematically generate subsumed clauses. One such source of redundancy consists in what we call (*forward*) *ancestor subsumption*, which denotes the subsumption of a newly derived clause by one of its own ancestors. A frequently occurring case of ancestor subsumption is caused by the symmetry clause $\neg Pxy \vee Pyx$. Resolving a clause C against this clause can be seen as exchanging the arguments in a P-literal of the clause C, and performing this operation twice obviously yields the original clause.

On closer inspection, it seems that the distinction between forward and backward subsumption, although apparently evident at a first glance, becomes irrelevant in many cases. While in the case of ancestor subsumption the already present and the newly deduced clause can definitely be determined, it is not that easy for the following example: Consider the set $S = \{PQ, \neg PR, \neg QS\}$. This set admits two different derivations of the clause RS, which could sequentially be executed. However, a distinction between an already present clause RS and the newly derived clause RS seems meaningless. Instead of forward or backward subsumption, one should better speak of a parallel derivation of identical clauses in this example.

Syntactic characterizations of the ancestor subsumption structure and of the parallel derivation of identical clauses are given in chapter 5, and it is shown that the generation of redundancy rests on "hidden" redundancies (partially inherent in links) that are inherited. Similar concepts of redundant links and their inheritance are investigated by Walther (1981), and by Ohlbach (1988). These characterizations lead to a strategy to prevent these unwanted derivations.

A better understanding of reduction for resolution based systems could be gained by taking a look at reduction in other reasoning systems, in particular the completion based systems, which traditionally emphasize the role of reduction (see for instance Hsiang (1982) and (1985), Kapur & Narendran (1985), or Müller (1988)). The relation between the completion approach and the resolution approach to theorem proving as revealed to some extent in Socher (1990) and Müller & Socher (1988), provides a means to integrate some of the strong reduction potentials of the completion method also into resolution based systems. One such possibility, which is further investigated in chapter 6, consists in an extension of the resolution calculus by equivalences. The problems caused by equivalences are already addressed in chapter 5, and their systematic treatment in chapter 6 provides a partial solution. A resolution calculus extended by logical equivalence is given in this chapter, and its soundness and completeness is proved.

2 Logical Foundations

This chapter provides the definitions for the basic notions and concepts that are used throughout this thesis.

2.1 Terms, Substitutions, and Unifiers

2.1.1 Definition:

Given a signature $\mathbb{F} = \bigcup_{n\in N} \mathbb{F}_n$ of finite sets of n-ary **function** symbols and a denumerable set $\mathbb{V}$ of **variable** symbols, the **term** set $\mathbb{T} = T(\mathbb{F},\mathbb{V})$ is the least set with $\mathbb{V} \subseteq \mathbb{T}$ and $ft_1...t_n \in \mathbb{T}$ whenever $f\in\mathbb{F}_n$ and $t_1,...,t_n \in \mathbb{T}$. This set is the carrier of the absolutely free term algebra, whose operators are the usual term constructors induced by the function symbols.

Parentheses may be used throughout this thesis for better readibility. For any object o containing variables we define $\mathbb{V}(o)$ to be the set of all **variables occurring in** o. A term t is said to be **ground**, iff $\mathbb{V}(t)=\emptyset$.

2.1.2 Definition:

A **substitution** is an endomorphism on the term algebra, which is identical almost everywhere on $\mathbb{V}$. For a substitution σ we define $dom(\sigma)$, the **domain** of σ, as the set $\{x\in\mathbb{V} \mid x\sigma\neq x\}$, and $cod(\sigma)$, the **codomain** of σ, as the set $\{x\sigma \mid x\in dom(\sigma)\}$.

A substitution σ with domain $\{x_1,...,x_n\}$ is usually represented as a set $\{x_1\rightarrow x_1\sigma,...,x_n\rightarrow x_n\sigma\}$ of argument-image pairs. The application of the substitution σ to any term t is denoted by $t\sigma$. The application of σ to any object containing terms is defined in the obvious way.

The set Σ of all substitutions on the term algebra together with the functional composition (which is formally denoted by juxtaposition of the substitutions), constitutes a monoid with the identity substitution ε as neutral element.

2.1.3 Definition:

A **permutation** is an invertible substitution, i.e. $\sigma\in\Sigma$ is a permutation, iff there exists a $\sigma^-\in\Sigma$ with $\sigma\sigma^- = \varepsilon$. A substitution σ is **idempotent**, iff $\sigma\sigma = \sigma$.

The set of permutations is denoted by Σ^-, the set of idempotent substitutions by Σ^*.

It is easy to see that the permutations are just the finite automorphisms of the term algebra. Another characterization of idempotent substitutions is the requirement that $dom(\sigma) \cap \mathbb{V}(cod(\sigma)) = \emptyset$ holds for each σ (see Herold 1983).

2.1.4 Definition:

For $\sigma \in \Sigma$ and $V \subseteq \mathbb{V}$ the **restriction** $\sigma|_V$ is the substitution with $dom(\sigma|_V) \subseteq V$ and which agrees with σ on V.

2.1.5 Definition:

The following relations are defined for $s,t \in \mathbb{T}$ and $\sigma,\tau \in \Sigma$ and $V \subseteq \mathbb{V}$.

$s \leq t$	iff there exists $\sigma \in \Sigma$ with $t=s\sigma$ (s *subsumes* t)[1]
$s \equiv t$	iff $s \leq t$ and $t \leq s$ holds (s is **subsumption equivalent** to t)
$s \equiv t$	iff there exists $\sigma \in \Sigma^-$ with $s=t\sigma$ (s is a **variant** of t)
$\sigma = \tau[\![V]\!]$	iff $\sigma|_V = \tau|_V$
$\sigma \leq \tau[\![V]\!]$	iff there exists $\lambda \in \Sigma$ with $\tau=\sigma\lambda[\![V]\!]$ (σ subsumes τ on V)
$\sigma \equiv \tau[\![V]\!]$	iff $\sigma \leq \tau[\![V]\!]$ and $\tau \leq \sigma[\![V]\!]$ hold (σ is subsumption equivalent to τ)
$\sigma \equiv \tau[\![V]\!]$	iff there exists $\lambda \in \Sigma^-$ with $\sigma=\tau\lambda$ (s is a variant of t)

In the following, we will omit the suffix $[\![V]\!]$, if $V=\mathbb{V}$.

Synonyms for "s subsumes t" are "t is an instance of s" or "s is more general than t". Synonyms for "s is a variant of t" are "s is a copy (or a duplicate) of t" or "s and t are equal up to renaming". The relation $\equiv$ is the equivalence relation generated by the preordering $\leq$. It can be shown that the variant relation and the subsumption equivalence coincide for terms and substitutions (Herold 1983). For $\sigma,\tau \in \Sigma^*$ the subsumption relation $\sigma \leq \tau$ can also be defined by $\sigma\tau=\tau$.

---

[1] Beware of the fact that the subsumption relation $\leq$ is sometimes written in the opposite direction.

2.1.6 Definition:

A **weak renaming** ρ is a substitution with $cod(\rho) \subseteq \mathbb{V}$. A **renaming** is a weak renaming, which is injective on its domain. The set of weak renamings is denoted by $\mathbb{P}^W$, the set of renamings by $\mathbb{P}$[1].

2.1.7 Definition:

Let $s,t \in \mathbb{T}$. A **unifier** for s and t is a substitution σ with $s\sigma = t\sigma$. A **most general unifier** for s and t is a unifier, which is minimal in the set of all unifiers of s and t. A **weak unifier** of s and t is a pair (σ,τ) with $s\sigma=t\tau$. The terms s and t are said to be (weakly) unifiable, if there exists a (weak) unifier.

Robinson (1965) and Huet (1976) show that the most general unifier of two unifiable terms is unique up to renaming, that is $\sigma \equiv \tau$ holds for any two most general unifiers σ and τ of s and t. According to Herold (1983), lemma III.2, one of the most general unifiers is always idempotent.

2.1.8 Definition:

Let $\sigma,\tau \in \Sigma^*$. A **unifier** for σ and τ is a substitution λ with $\sigma\lambda = \tau\lambda$. A **most general unifier** for σ and τ is a unifier, which is minimal in the set of all unifiers of σ and τ.
The substitutions σ and τ are said to be **compatible**, iff they have a unifier λ. They are said to be **strongly compatible**, iff $\sigma\tau = \tau\sigma$. If σ and τ are compatible substitutions, then the merge $\sigma^*\tau$ of σ and τ is the substitution $\sigma\lambda$, where λ is a most general unifier of σ and τ.

2.1.9 Lemma:

Let $\sigma,\tau \in \Sigma^*$. If σ and τ are strongly compatible, then they are compatible.

Proof: Take $\lambda = \sigma\tau$. ■

The converse is not true in general, as the example $\sigma=\{x\rightarrow y\}$, $\tau=\{y\rightarrow z\}$ shows. The substitutions σ and τ are compatible, with common instance $\sigma\tau$, but $\sigma\tau \neq \tau\sigma$. However, it is easy to see that for ground substitutions the converse is true:

---

[1] The letter $\mathbb{P}$ is the capital greek rho, which should not be mistaken for the latin letter P.

2.1.10 Lemma:

Two ground substitutions σ and τ are compatible, iff they are strongly compatible.

Proof: We have only to show that compatibility implies strong compatibility. Let σ and τ be compatible ground substitutions, that is, there is a $\theta \in \Sigma$ with $\sigma\theta = \tau\theta$. If $x \notin dom(\sigma) \cup dom(\tau)$, then obviously $x\sigma\tau = x = x\tau\sigma$. If $x \in dom(\sigma) \setminus dom(\tau)$, then $x\sigma\tau = x\sigma = x\tau\sigma$, and similarly for $x \in dom(\tau) \setminus dom(\sigma)$, and finally, if $x \in dom(\sigma) \cap dom(\tau)$, then $x\sigma\tau = x\sigma = x\sigma\theta = x\tau\theta = x\tau = x\tau\sigma$. ■

The following lemma gives an alternative definition of weak unifiability, which will be useful in later chapters.

2.1.11 Lemma:

For $s,t \in \mathbb{T}$ the following two assertions are equivalent:

a) s and t are weakly unifiable

b) There are $\rho \in \mathbb{P}$ and $\sigma \in \Sigma$ with $s\rho\sigma = t\sigma$.

Proof: See Eisinger (1988), Lemma 4.1.12.

2.2 Clauses and Resolution

2.2.1 Definition:

Let $\mathbb{P} = \bigcup_{n \in N} \mathbb{P}_n$ consist of finite sets of n-ary **predicate** symbols. The **atom** set $\mathbb{A} = \mathbb{A}(\mathbb{P},\mathbb{F},\mathbb{V})$ is the set consisting of the elements $Pt_1 \ldots t_n$ for $P \in \mathbb{P}_n$ and $t_1,\ldots,t_n \in \mathbb{T}$. If A is an atom, then +A (usually written A) and -A (usually written $\neg$A) are **literals**. The atom of the literal L is denoted by $\mathbb{A}(L)$, the predicate (symbol) of L is denoted by $\mathbb{P}(L)$. Literals L, K with the same atom but different sign are called **complementary**. As before, ground atoms, literals or clauses are objects containing no variables.

2.2.2 Definition:

A **clause** is a finite set of literals. The cardinality of the clause C is denoted by $|C|$. A clause C with $|C|=1$ is called a **unit** (clause). As a matter of convention, the empty clause is denoted by $\square$.

Clauses are usually written without set braces and commas. In particular, often we do not distinguish between a unit clause and its (single) literal. The

notions defined for terms apply to atoms and literals in the obvious way. Two literals are called **(weakly) unifiable**, if their signs are equal and their atoms are (weakly) unifiable. The set of unifiers of the literals $L \in C$ and $K \in D$ will be denoted by uni(C,L,D,K), where the clauses C and D can be omitted. Moreover, we define $uni(C,L,D) = \{\sigma \in uni(C,L,D,K) \mid K \in D\}$. Two literals are called **(weakly) resolvable**, if their signs are different and their atoms are (weakly) unifiable. Clauses containing complementary literals are called **tautologies**.

2.2.3 Definition:

Let C and D be clauses.

$C \leq D$	iff there exists $\sigma \in \Sigma$ with $C\sigma \subseteq D$ (C **subsumes** D)[1]
$C \equiv D$	iff $C \leq D$ and $C \leq D$ holds (C is **subsumption equivalent** to D)
$C \cong D$	iff there exists $\sigma \in \Sigma^{-}$ with $C = D\sigma$ (C is a **variant** of D)
$C \lneq D$	iff there exists $D' \subseteq D$ with $C \cong D'$
$C < D$	iff $C \leq D$ and $C \not\equiv D$ (C **properly subsumes** D)

Note that in particular we have $\square \leq C$ for each clause C. Furthermore, $C \leq \square$, iff $C = \square$.

Contrary to terms and substitutions, the variant relation for clauses does not coincide with the subsumption equivalence. A simple counterexample consists of the two clauses C = PxPyQz and D = PuQvQw, where $C \leq D$ and $D \leq C$ holds, but not $C \cong D$.

2.2.4 Definition:

Let C be a clause and let σ be a substitution such that $|C\sigma| < |C|$. Then $C\sigma$ is called a **factor** of C. A factor of C, which also subsumes C is called a **subsuming factor** of C, and so is each of its variants. A clause is called **irreducible**, if it possesses no subsuming factor. For any clause C, let C^* denote the **irreducible subsuming factor** of C.

---

[1] Beware of the fact that some authors, for instance Loveland (1978), call this relation *θ-subsumption*, whereas their term subsumption denotes *implication*. Sometimes the additional condition $|C| \leq |D|$ can be found in the literature, which is a technical requirement to prevent factors of a given clause C, which are in fact subsumed by C, to be eliminated by reduction procedures.

In general there exist several subsuming factors of a clause. In the next section, however, it will be shown that there always exists a unique "smallest" subsuming factor, that is, all irreducible subsuming factors are equal up to renaming. This justifies the definition of C^*.

2.2.5 Lemma:

If C' is a subsuming factor of C, then either $C' \lesssim C$, or there is a subsuming factor C'' of C' with $C'' \lesssim C$.

Proof: Let C' be a subsuming factor of C. By definition, there are literals $L,K \in C$ and $\sigma \in mgu(L,K)$, and a substitution μ such that $C' = C\sigma$ and $C'\mu \subseteq C$. Then $C'\mu\sigma \subseteq C\sigma = C'$.
Case 1: $|C'\mu| < |C'|$. Then μ must unify at least two literals of C', that is, $C'\mu$ is a factor of C'. Moreover, since $C'\mu\sigma \subseteq C'$, $C'\mu$ is also a subsuming factor of C'. Finally, $C'\mu \subseteq C$ holds. Take $C'' = C'\mu$.
Case 2: $|C'\mu| = |C'|$. We have $C'=C\sigma$. Since σ is idempotent, $C'\sigma=C'$, hence σ cannot unify two literals of C'. Then, a fortiori, σ cannot unify two literals of $C'\mu$, which implies $|C'\mu\sigma| = |C'\mu| = |C'|$. Together with $C'\mu\sigma \subseteq C'$ this implies $C'\mu\sigma = C'$. Hence μ must be injective, which implies $C' \cong C'\mu \subseteq C$. Thus $C' \lesssim C$. ■

Since irreducible clauses possess no subsuming factors, we have the following

2.2.6 Corollary:

For any clause C, $C^* \lesssim C$ holds. ■

2.2.7 Lemma:

Let C and D be clauses. Then the following conditions are equivalent:

a) $C \cong D$.

b) There are $\sigma,\tau \in \Sigma$ with $C\sigma = D$ and $D\tau = C$.

Proof: See Eisinger (1988), lemma 4.2.3. ■

The next lemma gives an alternative characterization of subsumption equivalence.

2.2.8 Lemma:

Let C and D be clauses. Then the following conditions are equivalent:

a) $C \equiv D$.

b) There are subsuming factors C' of C and D' of D with $C' \cong D'$.

c) $C^* \cong D^*$.

Proof: a) $\Rightarrow$b): Let $C \leq D$ and $D \leq C$. Then there are $\sigma,\tau \in \Sigma$ with $D\tau \subseteq C$ and $C\sigma \subseteq D$. From this we obtain $D\tau\sigma \subseteq C\sigma$ and $C\sigma\tau \subseteq D\tau$. Continuing this way, we obtain the following two chains:

$C \supseteq D\tau \supseteq C\sigma\tau \supseteq D\tau\sigma\tau \supseteq \ldots$ and

$D \supseteq C\sigma \supseteq D\tau\sigma \supseteq C\sigma\tau\sigma \supseteq \ldots$

Since the sets C and D are finite, there must be some $n \in \mathbb{N}$ such that $C(\sigma\tau)^j = D\tau\,(\sigma\tau)^j = D(\tau\sigma)^j\tau$ for all $j \geq n$. Analogously there must be an $m \in \mathbb{N}$ with $D(\tau\sigma)^k = C\sigma(\tau\sigma)^k = C(\sigma\tau)^k\sigma$ for all $k \geq m$. Let r be the maximum of m and n. Let $C' = C(\sigma\tau)^r \subseteq C$ and $D' = D(\tau\sigma)^r \subseteq D$. Then we have $C'\sigma = D'$ and $D'\tau = C'$. Lemma 2.2.5 now implies $C' \cong D'$.

b) $\Rightarrow$ c): follows from the fact that the irreducible factor of a clause is unique up to renaming.

c) $\Rightarrow$ a): Let $\sigma,\tau \in \Sigma$ with $C\sigma \subset C$ and $D\tau \subset D$ and $C\sigma \cong D\tau$. There are $\rho,\rho' \in \mathbb{P}$ with $C\sigma\rho = D\tau \subset D$ and $D\tau\rho' = C\sigma \subset C$, that is $C \leq D$ and $D \leq C$. ■

In particular, the previous lemma implies that subsumption equivalence and the variant relation coincide for irreducible clauses.

2.2.9 Definition:

a) If C and D are clauses with $\mathbb{V}(C) \cap \mathbb{V}(D) = \emptyset$, and $L \in C$, $K \in D$ are resolvable with most general unifier σ, then $(C \setminus \{L\})\sigma \cup (D \setminus \{K\})\sigma$ is called a **(binary) resolvent** of C and D. If R is a binary resolvent of factors C' and D' of C and D, respectively, then R is called a **resolvent** of C and D.

b) For each clause C, let $\mathbb{L}^+(C)$ denote the set of positive literals of C. If $N, E_1, \ldots, E_n$ are variable disjoint clauses with $\mathbb{L}^+(N) = \{L_1, \ldots, L_n\} \neq \emptyset$, $\mathbb{L}^+(E_i) = \emptyset$ and $K_i \in N_i$ for $i \in \{1,\ldots,n\}$, and σ is a most general simultaneous unifier of L_i and K_i for $i \in \{1,\ldots,n\}$, then

$$(N \setminus P)\sigma \cup \bigcup_{i=1}^{n} E_i \setminus \{L_i\}$$

is called a (negative) (binary) **hyperresolvent** of N with $\{E_1,\ldots,E_n\}$. The clause N is called the **nucleus**, the clauses D_i are the **electrons** of the hyperresolution step. A binary hyperresolvent of factors N', $E_1',\ldots,E_n'$ of $N,E_1,\ldots,E_n$ is hyperresolvent of $N,E_1,\ldots,E_n$. The definition of a positive hyperresolvent is obtained by reverting the signs in the above definition.

2.3 Uniqueness of the Irreducible Factor

In this section it is shown that the irreducible subsuming factor of a clause is unique up to renaming. According to lemma 2.2.5, we can restrict ourselves to those subsuming factors of a given clause C that are also subsets of C. In the following such a clause $D\subseteq C$, which is a subsuming factor of C, will be called a subsuming factor **in** C.

We consider a clause C together with the set $\Sigma_C = \{\sigma\in\Sigma \mid C\sigma\subseteq C\}$. Obviously, Σ_C together with the concatenation of substitutions is a semigroup with identy element ε. We define a quasiorder[1] $\subseteq$ on Σ_C by $\sigma\subseteq\tau$ iff $C\sigma\subseteq C\tau$ and an equivalence relation $\approx$ by $\sigma\approx\tau$ iff $C\sigma=C\tau$.

From the definition of Σ_C it is clear that the mapping $\Phi:\sigma\rightarrow C\sigma$ yields a surjective mapping from Σ_C on the set of all subsuming factors in C. The equivalence relation induced by this mapping is just the relation $\approx$. Thus there is a one-to-one correspondence between the subsuming factors in C and the elements of $\Sigma_C/\approx$, where the irreducible factors of C correspond to the minimal elements in $\Sigma_C/\approx$ w.r.t. $\subseteq$.

2.3.1 Lemma:
If τ is minimal in $\Sigma_C/\approx$ w.r.t. $\subseteq$, then $\tau\approx\sigma\tau$ holds for each $\sigma\in\Sigma_C$.

Proof: Let $\sigma\in\Sigma_C$. $C\sigma\subseteq C$ implies $C\sigma\tau\subseteq C\tau$, hence $\sigma\tau\subseteq\tau$. From the minimality of τ w.r.t. $\subseteq$ follows $\tau\approx\sigma\tau$. ■

2.3.2 Theorem:
If both $C\sigma$ and $C\tau$ are irreducible factors in C, then $C\sigma\cong C\tau$ holds.

---

[1] A quasiorder is a reflexive and transitive relation.

Proof: If both $C\sigma$ and $C\tau$ are irreducible factors in C then both σ and τ are minimal in in $\Sigma_C/\approx$ w.r.t. $\subseteq$. Lemma 2.3.1 implies that $\tau \approx \sigma\tau$ and $\sigma \approx \tau\sigma$ hold. Hence $C\tau = C\sigma\tau$ and $C\sigma = C\tau\sigma$. According to 2.2.7, this implies $C\sigma \equiv C\tau$. ■

2.4 Resolution and its Properties

2.4.1 Definition:

An **interpretation** $\mathfrak{I}$ is a maximal set of ground literals, containing no pair of complementary literals. $\mathfrak{I}$ **satisfies** its member literals and **falsifies** all other ground literals. An interpretation $\mathfrak{I}$ satisfies a ground clause, if it satisfies some literal of the clause. It satisfies an arbitrary clause, if it satisfies all of its ground instances, and satisfies a set of clauses, if it satisfies each member of the set. Any object is said to be **satisfiable** if there exists an interpretation satisfying it, and **unsatisfiable** otherwise. We also write $\mathfrak{I} \models o$, if $\mathfrak{I}$ satisfies the object o. An interpretation $\mathfrak{I}$ satisfying S is also called a **model** of S.

There is a very useful technique, which can serve various purposes, such as proving completeness of resolution: If S is an unsatisfiable clause set, then construct a clause set S(L) from S by removing all clauses containing the literal L, and deleting the literal $\neg L$ from the remaining clauses. Then the resulting clause set S(L) is also unsatisfiable, since each model $\mathfrak{I}$ of S(L) could easily be extended to a model $\mathfrak{I}'$ for S by adding the literal L. This construction will also be used in the following.

2.4.2 Definition:

a) Let S be a set of clauses. The **semantic closure** of S is the set of all clause sets S', such that $\mathfrak{I} \models S$ implies $\mathfrak{I} \models S'$. We write $\langle S \rangle$ for the semantic closure of S. By abuse of notation, we write $\langle C \rangle$ for $\langle \{C\} \rangle$.

b) Let S and S' be clause sets. S **implies** S', iff $S' \in \langle S \rangle$. S is (logically) **equivalent** to S', written $S \simeq S'$, iff S implies S' and S' implies S.

The implication relation S implies S' can also be expressed as follows: Each model of S is also a model of S'.

2.4.3 Lemma:

Let S be a set of clauses and let $C=\{L_1,\ldots,L_n\}$ be a clause. S implies C, iff $S \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\}$ is unsatisfiable.

Proof: Let $C \in \langle S \rangle$, and assume $S \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\}$ has a model $\mathfrak{I}$. Then a fortiori $\mathfrak{I}$ satisfies S, and from $C \in \langle S \rangle$ follows that $\mathfrak{I}$ satisfies C. Then there is one literal L_i of C, such that $\mathfrak{I} \models L_i$, in contradiction to the fact that $\mathfrak{I} \models S \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\}$. Conversely, assume that $S \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\}$ is unsatisfiable. Then each model of S must satisfy one element of $\{L_1,\ldots,L_n\}$, and hence satisfies C. ■

2.4.4 Lemma:

a) C implies D, iff $\langle D \rangle \subseteq \langle C \rangle$.

b) C is equivalent to D, iff $\langle C \rangle = \langle D \rangle$.

b) The implication relation on clauses defines a partial order.

Proof: a) If $\langle D \rangle \subseteq \langle C \rangle$, then obviously $\{D\} \in \langle D \rangle \subseteq \langle C \rangle$. Conversely, let $\{D\} \in \langle C \rangle$, and let $S \in \langle D \rangle$. Then each interpretation $\mathfrak{I}$ satisfying D also satisfies S. Let $\mathfrak{I}$ be an interpretation with $\mathfrak{I} \models C$. Then, since $\{D\} \in \langle C \rangle$, $\mathfrak{I} \models D$, and hence also $\mathfrak{I} \models S$. As $\mathfrak{I}$ was arbitrary, we have shown that $S \in \langle C \rangle$. This proves $\langle D \rangle \subseteq \langle C \rangle$.

b) and c) follow from a). ■

2.4.5 Lemma:

Let C and D be clauses, and let S be a clause set.

a) If S', $S'' \in \langle S \rangle$, then $S' \cup S'' \in \langle S \rangle$.

b) C implies D, iff $\{C,D\} \simeq \{C\}$.

c) If $C \leq D$, then C implies D.

Proof: a) Follows from the definition.

b) If C implies D, then $\{D\} \in \langle C \rangle$, with $\{C\} \in \langle C \rangle$ and a) we obtain $\{C,D\} \in \langle C \rangle$. Furthermore $\{C\} \in \langle S \rangle$ follows from the definition, hence we have $\{C,D\} \simeq \{C\}$. The converse is proved analogously.

c) Assume $C \leq D$, and $D=\{L_1,\ldots,L_n\}$. According to lemma 2.3.4 we have to show that $\{C\} \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\}$ is unsatisfiable. There is a substitution σ such that $C\sigma \subseteq D$, that is, there is $k \leq n$ such that $C\sigma=\{L_1,\ldots,L_k\}$. Let ϕ_{gr} be a ground substitution with $dom(\sigma)=\mathbb{V}(D)$. Then

$$(\{C\sigma\} \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\})\phi_{gr} = (\{L_1,\ldots,L_k\} \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\})\phi_{gr}$$

is a ground instance of $\{C\} \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\}$, which is obviously unsatisfiable. ■

From the previous lemma, part a), follows that a clause that is implied by another clause, can be discarded from a clause set without changing the

logical value of this set. In particular, by part c) of the lemma, this holds for subsumed clauses. In other words: Subsumed clauses are redundant.

2.4.6 Definition:

Let C,D,R be clauses, and let S be a clause set.

a) We write $C \rightarrow_D R$, iff R is a resolvent of C and D. C and D are called **parents** of R.
b) We write $S \rightarrow S \cup \{R\}$, iff there are $C,D \in S$, such that $C \rightarrow_D R$.
c) The reflexive, transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.
d) C is called an **ancestor** of R in a given deduction $S \rightarrow S'$, iff C is a parent of R, or C is parent of some ancestor C' of R.

2.4.7 Definition:

A **resolution refutation** of S is a derivation $S \rightarrow^* S'$ with $\square \in S'$.

Soundness and completeness of the resolution method are stated by the fact that a clause set is unsatisfiable, iff it admits a resolution refutation. This classical theorem correlates the semantic notion of unsatisfiability with the syntactic deduction relation. The proof of completeness usually divides into two parts: A completeness proof for ground resolution and a lifting lemma stating that each ground resolution step can be "lifted" to the appropriate first order resolution step. Several techniques, including semantic trees, or induction on the number of literals, to prove completeness of ground resolution can be found in the literature (see e.g. Chang & Lee (1973), Loveland (1978), Gallier (1988), Socher-Ambrosius (1989c)). Hyperresolution derivations are defined similarly to resolution refutations, and they can also be shown to be sound and complete.

As already noted, implied clauses may be discarded from clause sets without changing the logical value of the set. Still, this does not imply that implied clauses may be removed during a resolution refutation after each resolution step. For instance, each resolvent is implied by its parents, but discarding each resolvent immetiadely after its generation obviously precludes any successful refutation. For subsumed clauses, however, it has been shown (see Loveland (1978)), that their removal preserves completeness of resolution, even if certain restrictions are imposed on the resolution procedure.

3 Simplification

This chapter provides several techniques for *Boolean simplification*, which are based on the equations and rules of the theory of Boolean algebra. This particular theory does not admit a canonical term rewriting system. Nevertheless, there exists a normal form for Boolean algebra terms in the following sense: There is a transformation algorithm that transforms each Boolean algebra term t into another uniquely determined term t^*, and each term t', which is equal to t under the theory of Boolean algebra, is transformed into the same term t^*. This normal form is called the *set of prime implicants*. Historically, due to the early investigations of switching circuits, the prime implicants have been denoted in a disjunctive form. For the purposes of automated reasoning, however, the conjunctive form is usually preferred.

Throughout this chapter equality under the theory of Boolean algebra will be denoted by $\cong$, since no confusion with the variant relation can occur.

In the following the reader is assumed to be familiar with the basic notions of (equational) term rewriting.

3.1 Boolean Algebra and Prime Implicants

3.1.1 Definition:

A **Boolean algebra** is an algebra $(B,\wedge,\vee,\neg)$ with the binary operators $\wedge,\vee$ and the unary operator $\neg$, satisfying the following properties:

a) $(B,\wedge,\vee)$ is a distributive lattice, that is for all $a,b\in B$:

$a\vee b = b\vee a$	$a\wedge b = b\wedge a$	Commutativity
$a\vee(b\vee c) = (a\vee b)\vee c$	$a\wedge(b\wedge c) = (a\wedge b)\wedge c$	Associativity
$(a\vee b)\wedge b = b$	$(a\wedge b)\vee b = b$	Absorption
$a\wedge(b\vee c) = (a\wedge b)\vee(a\wedge c)$	$a\vee(b\wedge c) = (a\vee b)\wedge(a\vee c)$	Distributivity

b) $(a\wedge\neg a)\vee b = b$ $\qquad$ $(a\vee\neg a)\wedge b = b$

The set B is called the **carrier** of the Boolean algebra. Henceforth we shall denote a Boolean algebra by its carrier. The abbreviations $\Rightarrow$ and $\Leftrightarrow$ are defined as $a\Rightarrow b = \neg a\vee b$, and $a\Leftrightarrow b = a\Rightarrow b \wedge b\Rightarrow a$. The axioms of Boolean

algebra imply the following well-known properties of the operators $\vee,\wedge$, and $\neg$:

3.1.2 Lemma:

Let B be a Boolean algebra. Then there exist $0,1 \in B$, such that for all $a,b \in B$:

$0 \vee a \cong a$	$1 \wedge a \cong a$	
$1 \vee a \cong 1$	$0 \wedge a \cong 0$	
$a \vee a \cong a$	$a \wedge a \cong a$	Idempotency
$\neg(a \vee b) \cong \neg a \wedge \neg b$	$\neg(a \wedge b) \cong \neg a \vee \neg b$	
$\neg\neg a \cong a$		■

3.1.3 Lemma:

Let B be a Boolean algebra. For any $a,b \in B$:

(i) $a \vee b \cong 0$ iff $a \cong 0$ and $b \cong 0$.

(ii) $a \wedge b \cong 1$ iff $a \cong 1$ and $b \cong 1$. ■

The simplest Boolean algebra is the algebra $\mathbb{B}$ with carrier $\{0,1\}$. Given a set V of variables and a set C of constants, the set $\mathbb{T}(C,V)$ is the free term algebra whose operators are the term constructors induced by the symbols $\vee,\wedge$, and $\neg$. The Boolean algebra $\mathbb{B}[C,V]$, whose carrier is $\mathbb{T}(C,V)$, is also called the set of Boolean polynomials[1] over $\mathbb{B}$. A Boolean polynomial over the variables $\{x_1,\ldots,x_n\}$ can also be interpreted as a function $f: \mathbb{B}^n \rightarrow \mathbb{B}[C]$ (Rudeanu 1974).

3.1.4 Lemma:

There is no canonical term rewriting system modulo associativity and commutativity of $\wedge$ and $\vee$ for the theory of Boolean algebra[2].

Proof: See Appendix. ■

3.1.5 Definition:

Let B be a Boolean algebra, and let $s,t \in B$.

a) We define the relation $\leq$ on B by $s \leq t$, iff $s \vee t \cong t$.

---

[1] The Boolean polynomials are also called Boolean functions (Rudeanu 1974), or simply Boolean terms.

[2] As to the definitions of (equational) canonical term rewriting systems see for instance Huet & Oppen (1980).

b) The **principal filter** $\langle t \rangle$ generated by t, is defined by $\langle t \rangle = \{t' \in B \mid t \leq t'\}$.

3.1.6 Lemma:

Let $s,t \in B$. Then the following conditions are equivalent to $s \leq t$:

a) $s \vee t \cong t$

b) $s \wedge t \cong s$

c) $s \wedge \neg t \cong 0$

d) $\neg s \vee t \cong 1$

Proof: a) $\Leftrightarrow$ b): If $s \vee t \cong t$ then $s \cong s \wedge (s \vee t) \cong s \wedge t$. The converse is proved analogously.

b)$\Rightarrow$c): If $s \wedge t \cong s$, then $s \wedge \neg t \cong s \wedge t \wedge \neg t \cong 0$.

c)$\Rightarrow$d): If $s \wedge \neg t \cong 0$, then $\neg s \vee t \cong 1$.

d)$\Rightarrow$b): If $\neg s \vee t \cong 1$, then $s \cong s \wedge 1 \cong s \wedge (\neg s \vee t) \cong s \wedge t$. ■

In spite of lemma 3.1.4, there exists a normal form for Boolean algebra polynomials, the so called set of prime implicants. In order to define the notion of prime implicants, we first transfer the semantic notions for clauses and clause sets to Boolean algebra polynomials.

Given a set $\mathbb{A}$ of ground atoms, let

$\mathbb{B}[\mathbb{A}] = \{f(A_1,\ldots,A_n) \mid A_1,\ldots,A_n \in \mathbb{A}, f \in \mathbb{B}[\emptyset,V]\}$

Then $\mathbb{B}[\mathbb{A}]$ is isomorphic[1] to the Boolean algebra $\mathbb{B}[\mathbb{A},\emptyset]$. In the following we shall establish the relation between a given set S of ground clauses and the Boolean algebra $\mathbb{B}[\mathbb{A}]$, where $\mathbb{A}$ is the atom set of S.

3.1.7 Definition:

Let S be a set of ground clauses and let $\mathbb{A}=\mathbb{A}(S)$ be the set of atoms occurring in S. Let $\alpha: \mathbb{A} \rightarrow \mathbb{V}$ be an injective mapping, and let $V = \alpha(\mathbb{A})$.

We define a mapping $\phi: \mathbb{L}(S) \cup S \cup 2^S \rightarrow \mathbb{B}[\emptyset,V]$ as follows:

$\phi(+A) = A\alpha$, for any positive literal +A

$\phi(-A) = \neg A\alpha$, for any negative literal -A

$\phi(\{L_1,\ldots,L_n\}) = \phi(L_1) \vee \ldots \vee \phi(L_n)$ for any clause $\{L_1,\ldots,L_n\}$

$\phi(\{C_1,\ldots,C_m\}) = \phi(C_1) \wedge \ldots \wedge \phi(C_m)$ for any clause set $\{C_1,\ldots,C_m\} \subseteq S$

---

[1] The notion of homomorphism and isomorphism of Boolean algebras is defined in a canonical way, see for instance, Rudeanu (1974).

For any object O, the polynomial $\phi(O)$ will be denoted by f_O. Following the convention that a disjunction over the empty set is zero, we have $f_\square = 0$. As for clauses, the number n is called the **length** of f_C. The objects f_O will be called literal (clause, CNF-) polynomials. CNF stands for clausal normal form, or conjunctive normal form.
For any polynomial $f_O(x_1,\ldots,x_n) \in \mathbb{B}[\emptyset,V]$, we define a term $t_O \in \mathbb{B}[\mathbb{A}]$ by $t_O = f_O(x_1\alpha^{-1},\ldots,x_n\alpha^{-1})$. The objects t_O will be called literal (clause, CNF-) terms. Conversely, each clause term $t \neq 1$ defines a unique clause, and each CNF-term defines a (not necessarily unique) clause set. It can be shown that each $t \in \mathbb{B}[\mathbb{A}]$ is equivalent to some CNF-term (see, for instance, Rudeanu (1974)).[1]

While there exists no canonical AC-term rewriting system (that is, a system modulo associativity and commutativity) for Boolean algebra, there is at least a terminating (but not confluent) system, which is, however, even confluent on clause terms.

3.1.8 Lemma:
Let AC denote the set of equations representing commutativity and associativity of $\wedge$ and $\vee$, and let R be the following system of rules:

$a\vee(b\wedge c) \rightarrow (a\vee b)\wedge(a\vee c)$
$(a\wedge b)\vee b \rightarrow b$

$a\vee a \rightarrow a$	$a\wedge a \rightarrow a$
$\neg a\vee a \rightarrow 1$	$\neg a\wedge a \rightarrow 0$
$1\vee a \rightarrow 1$	$0\wedge a \rightarrow 0$
$1\wedge a \rightarrow a$	$0\vee a \rightarrow a$
$\neg 1 \rightarrow 0$	$\neg 0 \rightarrow 1$

$(a\vee b)\wedge(\neg a\vee b) \rightarrow b$

$\neg(a\vee b) \rightarrow \neg a\wedge\neg b$	$\neg(a\wedge b) \rightarrow \neg a\vee\neg b$

$\neg\neg a \rightarrow a$

Then the equational system $\mathfrak{R}=(AC,R)$ is terminating. On the set of clause polynomials, it is even confluent.

---

[1] The difference between polynomials and terms is often blurred in the literature by regarding a term like $P\vee Q$ as a polynomial in the *variables* P,Q. This view, however, is not quite correct because the polynomial $f(P)=P$ is identical to the polynomial $f(Q)=Q$, whereas the clauses {P} and {Q} are different.

Proof: See Appendix. ■

Now let $\mathfrak{I}$ be an interpretation, and let $t \in \mathbb{B}[\mathbb{A}]$. For any $A \in \mathbb{A}$, let $A(\mathfrak{I})$ be defined by $A(\mathfrak{I})=1$, if the literal $+A$ is in $\mathfrak{I}$, and $A(\mathfrak{I})=0$ otherwise.

3.1.9 Lemma:

Let S be a ground clause set with $\mathbb{A}(S)=\{A_1,\ldots,A_n\}$, and let $\mathfrak{I}$ be an interpretation. Then $\mathfrak{I} \models S$ iff $f_S(A_1(\mathfrak{I}),\ldots,A_n(\mathfrak{I})) \cong 1$.

Proof: Obviously $L \in \mathfrak{I}$ iff $f_L(A_1(\mathfrak{I}),\ldots,A_n(\mathfrak{I})) \cong 1$ holds for any literal L.
$\mathfrak{I}$ does not satisfy S, iff there is some $C \in S$ with $\mathfrak{I} \not\models C$
iff there is some $C \in S$ with $L \notin \mathfrak{I}$ for all $L \in C$
iff there is some $C \in S$ with $f_L(A_1(\mathfrak{I}),\ldots,A_n(\mathfrak{I})) \cong 0$ for all $L \in C$
iff there is some $C \in S$ with $f_C(A_1(\mathfrak{I}),\ldots,A_n(\mathfrak{I})) \cong 0$
iff $f_S(A_1(\mathfrak{I}),\ldots,A_n(\mathfrak{I})) \cong 0$. ■

3.1.10 Corollary:

Let S be a clause set. Then $t_S \cong 0$ iff S is unsatisfiable. ■

3.1.11 Lemma:

Let S be a ground clause, and let $C = \{L_1,\ldots,L_n\}$ be a single ground clause. Then $t_C \in \langle t_S \rangle$, iff $t_S \wedge \neg L_1 \wedge \ldots \wedge \neg L_n \cong 0$.

Proof: We have $t_C \in \langle t_S \rangle$, iff $t_S \leq t_C$ iff, by lemma 3.1.9, $t_S \wedge \neg t_C \cong 0$, iff $t_S \wedge \neg L_1 \wedge \ldots \wedge \neg L_n \cong 0$. ■

Obviously, the relation $\leq$ defines an order on B. The correspondence between the implication relation and semantic closure on the one hand and the notions for Boolean algebra given in the previous definition, is established by the following

3.1.12 Lemma:

Let S_1 and S_2 be a set of ground clauses, for i=1,2, let $t_i=t_{S_i}$, and let C be a ground clauses.

a) $t_1 \leq t_D$ iff S_1 implies D.

b) $t_1 \leq t_2$ iff S_1 implies S_2.

c) $\langle t_C \rangle = \{t_{C'} \mid C' \in \langle C \rangle\}$.

Proof: a) Follows immetiadely from the previous lemma and 2.4.3.

b) Let $S_2 = \{D_1,\ldots,D_n\}$, and for i=1,...,n, let $t_i=t_{D_i}$. We have

$t_1 \leq t_2$ iff $t_1 \wedge \neg t_2 \cong 0$ iff

$t_1 \wedge (\neg d_1 \vee \ldots \vee \neg d_n) \cong 0$ iff
$t_1 \wedge \neg d_1 \cong 0, \ldots, t_1 \wedge \neg d_n \cong 0$ iff
S_1 implies D_1 and ... and S_1 implies D_n iff
S_1 implies S_2.

c) $t_{C'} \in \langle t_C \rangle$ iff $t_C \leq t_{C'}$ iff C implies C' iff $C' \in \langle C \rangle$. ■

3.1.13 Corollary:

Let S' and S'' be ground clause sets. Then $t_{S'} \cong t_{S''}$ iff $S' \simeq S''$. ■

According to the previous lemma, the Boolean algebra $\mathbb{B}[\mathbb{A}]$ is the Lindenbaum algebra of S (see Gallier 1988). Now suppose S and S' are first order clause sets, and S_{gr}, S'_{gr} are ground clause sets obtained from S and S' by instantiating each variable with a new constant. If the two ground clause sets S_{gr} and S'_{gr} are logically equivalent, then the corresponding first order clause sets S and S' are also equivalent. From the previous lemma thus follows that transformations based on the rules of Boolean algebra on a clause set S_{gr} leave the logical value of S invariant. This means that Boolean transformations operate on clause sets (or, more generally, on arbitrary first-order formulae) as they do on the corresponding ground objects. Hence the transformations considered in the following chapter 3.2 will be stated for pure propositional logic, which in fact is equivalent to the logic of ground formulae.

3.1.14 Definition:

Let t be a Boolean algebra polynomial. Then a **prime implicant** of t is a minimal clause polynomial in $\langle t \rangle$ (w.r.t. the partial order $\leq$). If $t_1, \ldots, t_n$ are the prime implicants of t, then $\mathcal{P}(t) = t_1 \wedge \ldots \wedge t_n$ is called the **prime polynomial** of t.

3.1.15 Lemma:

For each polynomial t, $\mathcal{P}(t) \cong t$ holds.

Proof: Let $\mathcal{P}(t) = t_1 \wedge \ldots \wedge t_n$. First we show that $t \leq \mathcal{P}(t)$. Since $t_i \in \langle t \rangle$, $t \leq t_i$ holds for any $i \in \{1, \ldots, n\}$. Hence,

$$t \vee \mathcal{P}(t) = t \vee (t_1 \wedge \ldots \wedge t_n) \cong (t \vee t_1) \wedge \ldots \wedge (t \vee t_n) \cong t_1 \wedge \ldots \wedge t_n = \mathcal{P}(t).$$

Next we show that $\mathcal{P}(t) \leq t$. Let $s_1 \wedge \ldots \wedge s_m$ be a CNF-representation of t. Then, for each $j \in \{1, \ldots, m\}$, $t \leq s_j$, which implies $s_j \in \langle t \rangle$. As the t_i are the minimal clause polynomials in $\langle t \rangle$, and s_j is a clause polynomial, there is some

$i \in \{1,\ldots,n\}$ with $t_i \leq s_j$. Thus there are $t_1',\ldots,t_m' \in \{t_1,\ldots,t_n\}$ such that $t_j' \leq s_j$ for each $j \in \{1,\ldots,m\}$. Then we have

$$\mathcal{P}(t) = t_1 \wedge \ldots \wedge t_n \leq t_1' \wedge \ldots \wedge t_m' \leq s_1 \wedge \ldots \wedge s_m \equiv t. \quad \blacksquare$$

Besides the prime implicants' minimality property, which is given by the definition above, prime implicants also have the useful property of being minimal w.r.t. their length. In other words, each clause polynomial in $\langle t \rangle$, which is not a prime implicant of t, contains some superfluous literals. This property can be used to formulate one notion of simplicity and has been the major reason for the interest in prime implicants.

3.1.16 Definition:

Let C and D be ground clauses. Then t_C **subsumes** t_D, iff C subsumes D.

Thus the clause polynomial s subsumes the clause polynomial t, iff each literal occurring in s also occurs in t.

3.1.17 Lemma:

Let s,t be clause polynomials with $s,t \not\equiv 1$. Then $s \leq t$ iff s subsumes t.

Proof: If s subsumes t, then t is of the form $s \vee k_1 \vee \ldots \vee k_n$. Then $s \vee t = s \vee s \vee k_1 \vee \ldots \vee k_n = t$, that is, $s \leq t$. Conversely, assume that $s \leq t$. Let $s = k_1 \vee \ldots \vee k_n$ and let $t = h_1 \vee \ldots \vee h_m$. Then

$$k_1 \vee \ldots \vee k_n \vee h_1 \vee \ldots \vee h_m \equiv h_1 \vee \ldots \vee h_m.$$

The term rewriting system $\mathfrak{R}$ of lemma 3.1.8 is canonical for clause polynomials. W.l.o.g. we can assume that the polynomials s and t are already irreducible w.r.t. $\mathfrak{R}$. Thus we have

$$k_1 \vee \ldots \vee k_n \vee h_1 \vee \ldots \vee h_m \xrightarrow{*}_{\mathfrak{R}} h_1 \vee \ldots \vee h_m$$

Since $s,t \not\equiv 1$, the only rule of R applicable to the left hand side, is the idempotency rule $a \vee a \rightarrow a$. Hence for each k_i there is some h_j with $k_i = k_j$, that is, all literals occurring in s also occur in t. Thus s subsumes t. $\blacksquare$

This a particular case of Gottlob's (1987) result for first order logic on the equivalence of subsumption and implication for non-selfresolving clauses.

Several methods to compute the prime implicants of a given ground formula have been developed, for instance Quine's (1959) *method of iterated consensus*, or the algorithms of Tison (1967), and Slagle, Chang & Lee (1970). They all consider clauses and clause sets instead of polynomials over a Boolean algebra. The prime polynomial of a clause set S is thus called the *set*

of prime implicants of S. Moreover, the sets representing clauses are written without set braces and without separating commas. For instance, the clause set {{P,Q},{¬P,Q,R}} will be denoted simply by {PQ, ¬PQR}. Since these two formulations are equivalent according to lemma 3.1.10, we will also adopt the more convenient notation using clauses and clause sets in the following.

The method of iterated consensus appears to be the best-known technique to generate the prime implicants of a formula. It starts from a formula given in conjunctive normal form, that is, a set S of clauses. Then the nontautological resolvents of clauses of S are repeatedly formed and added to S. Each time a resolvent is generated, subsumed clauses are removed from S. When only such new clauses can be produced that are already subsumed by existing clauses, the set of prime implicants has been obtained. The following example is taken from Loveland & Shostak (1980).

3.1.18 Example:

Let

$S = \{\neg PRS, \neg PQR\neg S, PQRS\}$.

From the given clauses we can derive the resolvent ¬PQR, which subsumes its parent clause ¬PQR¬S, and we obtain

$S_1 = \{\neg PRS, \neg PQR, PQRS\}$

In the next step we derive QRS, which subsumes its parent PQRS, and we obtain

$S_2 = \{\neg PRS, \neg PQR, QRS\}$

No more resolvents can be derived from this set, which thus represents the prime implicants of S.

Since there is only a finite set of clauses built from a given set of literals, it is obvious that the algorithm terminates. In order to prove the algorithm's correctness, we first give a lemma concerning a property of resolution derivations. One of the resolution method's main advantages is the fact that not all possible inferences can be drawn from a given problem set S. However, the important inferences, namely the minimal clauses in ⟨S⟩ can be derived, as is shown by the following lemma. This lemma is due to Kowalski (1970), we will however, provide a shorter proof that basically uses the refutation completeness of resolution.

3.1.19 Lemma:

Let S be a set of ground clauses. For each clause $C \in \langle S \rangle$, there exists a clause D, which subsumes C and a resolution derivation of D from S.

Proof: Let $C=\{L_1,\ldots,L_n\}$. If $C \in \langle S \rangle$, then by lemma 2.4.3 $S \cup \{\neg L_1\} \cup \ldots \cup \{\neg L_n\}$ is unsatisfiable. According to the construction in section 2.4 let $S' = S(\neg L_1,\ldots,\neg L_n)$ be obtained from S by removing all clauses containg some $\neg L_i$, and deleting all literals L_j from the remaining clauses. Then S' is still unsatisfiable and thus admits a resolution refutation, that is a derivation $S' \rightarrow^* S''$, with $\square \in S''$. By adjoining the literals L_j back to those clauses, from which they were removed, we obtain a resolution derivation $S \rightarrow^* S'''$, with $D \in S'''$, where D is a subset of the literals $\{L_1,\ldots,L_n\}$, that is, D subsumes C. ■

3.1.20 Corollary:

Let S be a set of ground clauses, and let $\mathcal{P}$ be the set of clauses constructed by the method of iterated consensus from S. Then $\mathcal{P}$ is the set of prime implicants of S.

Proof: First of all, we remark that we deal only with nontautological clauses. As subsumption and implication coincide for nontautological clauses (lemma 3.1.16), the prime implicants are the minimal clauses w.r.t. the subsumption ordering. Furthermore, obviously each clause D generated by the algorithm is in $\langle S \rangle$.

Let C be a prime implicant of S. The previous lemma guarantees that C, which is a minimal clause in $\langle S \rangle$ w.r.t. subsumption, can be obtained by a resolution derivation from S. Moreover, since C is minimal, no clause C' which properly subsumes C, can be generated. Since the method produces all possible resolvents, that are not already subsumed by existing clauses, the clause C must be generated, if it is not already present in S. Moreover, it is guaranteed that the generated non minimal clauses in $\langle S \rangle$ are removed during the algorithm. ■

Most of the well known techniques for generating a formula's prime implicants require the formula already be given in clausal form. In the next section we will present an algorithm that generates the prime implicants of arbitrary formulae.

3.2 An Optimized CNF Transformation

Most resolution based theorem proving systems require that the logical formulae, which are to be proved, should be converted into clausal normal form. This transformation usually takes several steps including the elimination of implications and equivalences, skolemization and possibly splitting into several easier to prove subformulae (Eisinger & Weigele 1983). In any case, the procedure's last step consists in the multiplication of formulae in prenex negation normal form into disjunctive normal form or into conjunctive normal form. Usually the transformation into disjunctive form is required for formulae to be tested for splitting whereas conversion into conjunctive form is necessary for the single splitparts. This multiplication, possibly resulting in an inflation of the original formula, is the algorithm's most critical step. Multiplying a disjunctive form $\mathcal{D}$ to a conjunctive form $\mathcal{C}$ (or vice versa), the number of subformulae of $\mathcal{C}$ depends exponentially on the number of subformulae of $\mathcal{D}$. Yet often a significant part of the resulting formulae are redundant, as the following example shows:

3.2.1 Example:

Assume the propositional formula

$\mathcal{F} = (P \wedge R \wedge S) \vee (P \wedge Q \wedge S) \vee (Q \wedge R \wedge S)$

is to be transformed into CNF. In the following we drop the $\vee$ and write conjunctions as sets. The first step of this transformation is a distributive multiplication into clause form, which yields

{PPQ, PPR, PPS, PQQ, PQR, PQS, PSQ, PSR, PSS,
(1) RPQ, RPR, RPS, RQQ, RQR, RQS, RSQ, RSR, RSS,
SPQ, SPR, SPS, SQQ, SQR, SQS, SSQ, SSR, SSS}

We call this form the totally multiplied form of $\mathcal{F}$.

Using the commutativity and idempotence of $\vee$, this clause set can be simplified to

{PQ, PR, PS, PQ, PQR, PQS, PSQ, PSR, PS,
(2) RPQ, RP, RPS, RQ, RQ, RQS, RSQ, RS, RS,
SPQ, SPR, SP, SQ, SQR, SQ, SQ, RQ, S}

In this formula all multiple occurrences of clauses and all clauses that are subsumed by some other clause, are redundant. Deleting these redundant clauses yields

(3) {PQ, PR, RQ, S}

In this particular example the formula $\mathcal{F}$ to be transformed into CNF is given in the opposite, disjunctive, normal form. The transformation between opposite normal forms is the basic case of our algorithm and will be dealt with first. We will only consider the conversion into CNF, the transformation in the other direction is just symmetric.

The example shows that the multiplication process produces many terms that can be deleted by subsequent simplification steps. Would it not be better to avoid generating these redundant terms in the first place? Ideally such an algorithm's output should be a minimal representation of the original formula. In this chapter we present an algorithm that multiplies formulae into clausal normal form without producing redundant clauses. The output $\mathcal{P}$ of the algorithm, given a formula $\mathcal{F}$, is the set of *prime implicants* of $\mathcal{F}$. In particular, if $\mathcal{F}$ is an unsatisfiable ground formula then $\mathcal{P}$ consists only of the empty clause.

In the following we will briefly sketch the proceeding of the algorithm using example 3.2.1.

The method can be considered as an extension of the well-known matrix methods in automated theorem proving (Andrews 1981 and Bibel 1982). While it is sufficient to find a spanning set of appropriate paths through the matrix in order to refute a formula, we have to develop all paths through the matrix in order to generate the clausal form of a formula.

We write the formula $\mathcal{F}$ as a 3×4-matrix M, labeling the rows of M with the variables of $\mathcal{F}$ and the columns of M with the numbers of the subformulae of $\mathcal{F}$. We let M(P,k)=1 if the k-th term of $\mathcal{F}$ contains the propositional variable P and M(P,k)=0 otherwise. This results in the following matrix M for $\mathcal{F}$

M=

	1	2	3
P	1	1	0
Q	0	0	1
R	1	0	1
S	1	1	1

The columns of the matrix represent the conjunctions of the original formula $\mathcal{F}$ and the clauses of the totally multiplied form of $\mathcal{F}$ correspond to the paths through the matrix. A *path* is obtained by taking in each column of

M a nonzero entry and writing down the variable of the corresponding row. Thus a path is a sequence (P_1,P_2, P_3) where $M(P_1,1) = M(P_2,2) = M(P_3,3) = 1$. For instance (P,P,Q), (P,S,R) or (S,S,S) are paths through M. We say that a path p *subsumes* another path q, if all variables occurring in p also occur in q, i.e. if the clause corresponding to p subsumes the clause corresponding to q.

The terms of the reduced form (2) of $\mathcal{F}$ can be obtained by removing multiple occurrences of variables in the paths. Then (P,Q) stands for the path (P,P,Q) and (R,Q) corresponds to the path (R,Q,R) in M.

The terms of the totally reduced form (3) correspond to the following subset of paths of M: If a path p of M contains two entries of the same variable in two different columns i and j, for instance (P,P,Q), then all paths differing from P only in either i or j, namely the paths (P,P,Q), (P,S,Q), (R,P,Q) and (S,P,Q), are subsumed by (P,Q) or equal to (P,Q) up to permutation. The generation of these redundant paths can be avoided in the following way: Having developed a partial path $(P_1,P_2,...,P_i)$ as before, take only those 1-entries in the column i+1, which are not already on the path, that is, those entries satisfying $M(P_j, i+1)=0$ for all $j \in \{1,..,i\}$.

In our example, developing the paths beginning with the row S, we see that all columns except the first can be discarded. Once we have developed the path containing only the variable S, we can cancel the whole row: all paths starting with a different variable and containing S are subsumed by the path (S). This is the analogon to the transformation:

$$(P \wedge R \wedge S) \vee (P \wedge Q \wedge S) \vee (Q \wedge R \wedge S) \rightarrow S \wedge ((P \wedge R) \vee (P \wedge Q) \vee (Q \wedge R)).$$

The computation of unnecessary paths can be avoided by developing first the paths of the form (S,...): Beginning with the paths starting with P, for instance, one obtains the paths (P,Q), (P,R), (P,S), from which (P,S) is redundant, since it is subsumed by (S).

The computation of $\mathcal{P}$ from the matrix M proceeds as follows:

1. At the beginning the result set $\mathcal{P}$ is empty.

First we develop the S-row. The second and third column don't have to be considered, since they have a 1-entry at S. Having obtained the path (S) in this way, we add it to $\mathcal{P}$ and cancel the S-row from the matrix. Now we have $\mathcal{P}=\{(S)\}$ and

M=

	1	2	3
P	1	1	0
Q	0	1	1
R	1	0	1

2. Next we develop the P-row obtaining the two paths (P,Q) and (P,R) (the second column can be canceled, since it contains a 1-entry for P), add them to the solution set and obtain $\mathcal{P}$={(S),(P,Q),(P,R)}. Now the P-row can be removed too and it remains the matrix

M=

	1	2	3
Q	0	1	1
R	1	0	1

This matrix corresponds to the formula

$\mathcal{F} = R \vee Q \vee (Q \wedge R)$

Now the absorption law applies to $\mathcal{F}$ yielding

$\mathcal{F} = R \vee Q$

The analogon in our procedure is the deletion of the third column of M and hence

M=

	1	2
Q	0	1
R	1	0

3. Now the only path in M is (R,Q). This path is added to the solution set and we have obtained the result $\mathcal{P}$ = { (S), (P,Q), (P,R), (R,Q)}.
This is the set of prime implicants of the original formula $\mathcal{F}$ and since no pair of clauses is resolvable, it is already a simplest equivalent of $\mathcal{F}$.

After the introductory example we provide the definitions of normal form matrices and paths as well as some of their basic properties.
$\mathbb{P}$ is a set of propositional variable symbols. $\mathbb{L}$ is the set of all propositional literals (P and $\neg$P).
For any object o containing variables we define $\mathbb{P}$(o) as the set of all propositional **variables occurring in** o. $\mathbb{L}$(o) is the set of all **literals** occuring in o.

3.2.2 Definition:

A normal form matrix (NF-matrix) M is an n×k-matrix over the set {0,1}. The rows of M are labeled with different elements from $\mathbb{L}$. We write M(p,i) for the element of M in the p-th row and the i-th column. If i is a column of M, then we define

$$t^i(M) = \bigvee_{M(P,i)=1} P$$

to be the clause term corresponding to the column i. We define

$$t_C(M) = \bigwedge_{i=1}^{k} t^i(M),$$

the formula in disjunctive normal form belonging to M and analogously $t_D(M)$, the formula in conjunctive normal form belonging to M.

The notions of *subsumption* and *tautology* are then defined for columns of NF-matrices just as for the corresponding clause terms.

3.2.3 Definition:

Let M be an NF-matrix.

(i) A **complete path** p through M is a sequence $(P_1,...,P_n)$ of variables such that $M(P_i,i)=1$ for each i. A **path** through M is a subsequence of a complete path. A **clause path** is a path with no multiple occurrences of variables. The concatenation of two paths p and q is denoted by $p \oplus q$. We write $p(i)$ for the i-th element of the path p.

(ii) Let $\mathcal{P}$ be a set of paths through a matrix M. For any $p \in \mathcal{P}$, we define

$$t^p(M) = \bigvee_{P \in \mathbb{L}(p)} P$$

the clause term corresponding to p, and

$$t_C(\mathcal{P}) := \bigwedge_{p \in \mathcal{P}} t^p(M),$$

the term in conjunctive normal form belonging to $\mathcal{P}$ and analogously $t_D(\mathcal{P})$, the term in disjunctive normal form belonging to $\mathcal{P}$. We call these terms the **totally multiplied forms** of M.

From the definitions it is clear that the set $\mathcal{P}$ of all complete paths through an NF-matrix M represents the totally multiplied form of the term belonging to M, i.e. $t_C(\mathcal{P}) \cong t_D(M)$ and $t_D(\mathcal{P}) \cong t_C(M)$. Again, the notions of *subsumption* and *tautology* are defined for paths just as they are for the corresponding clause terms. Also, the notion of a *resolvent* of clause paths is

defined as for clauses. In the following we will consider only non-tautologous paths.

It is obvious that subsumed paths and columns can be canceled without changing the corresponding term's value.

3.2.4 Lemma:

(i) Let P be a row of M such that M(P,1)=1. Each complete path p containing P at a position k≠1with P≠p(1) is a properly subsumed path.

(ii) Let p be a complete path of M with p(j)=P. If there is a column i≠j, such that M(P,i)=1 and P≠p(i), then p is a properly subsumed path.

Proof: (i) Since M(P,1)=1, the path q defined by q(1)=P and q(j)=p(j) for j>1 is a complete path of M. Since P∈$\mathbb{L}(p)$, $\mathbb{L}(q) = \mathbb{L}(p) \setminus \{p(1)\} \subset \mathbb{L}(p)$.
(ii) Since M(P,i)=1, the path q defined by q(i)=P and q(j)=p(j) for j≠i is a complete path of M. Since P∈$\mathbb{L}(p)$, $\mathbb{L}(q) = \mathbb{L}(p) \setminus \{p(i)\} \subset \mathbb{L}(p)$. ■

Now we are ready to formulate the algorithm that performs an optimized multiplication between conjunctive and disjunctive normal form.

Algorithm **Transform**
INPUT: An n×k-matrix M .
OUTPUT: A set $\mathcal{P}$ of paths through M such that $t_C(M) \cong t_D(\mathcal{P})$ and $t_D(M) \cong t_C(\mathcal{P})$

1. $\mathcal{P} := \emptyset$. Cancel all tautological columns of M.
2. Cancel all subsumed columns of M. If M is now a matrix with only zero entries, go to step 5.
3. Take a row P of M that has a maximal number of 1-entries and permute the columns of M in such a way, that M(P,1) = 1. Generate all paths of M with initial part (P) at column 2 and add them to $\mathcal{P}$.
4. Cancel the row P of M and go to 2.
5. Remove properly subsumed paths from $\mathcal{P}$.
6. Return $\mathcal{P}$.

Generate all paths of **M** with initial part q at column **i**
INPUT: An n×k-matrix M corresponding to a formula $\mathcal{F}$ in conjunctive normal form, a path q, developed from column 1 to column i-1, and a column i of M.
OUTPUT: $Q = \{p \in \mathcal{P} \mid p$ is output of transform and p has initial part Q$\}$

COMMENT: All parameters are value parameters, in particular the matrix M is unchanged when the procedure has terminated.

1. If i is greater than the last column of M then return $\{q\}$.
2. If there is a $P \in \mathbb{P}(q)$ such that M(P,i) = 1 then i:=i+1; go to 1.
3. Q:=Ø. Let V = $\{Q \in \mathbb{L}(M) \setminus \{\neg P \mid P \in q\} \mid M(Q,i) = 1\}$. Sort V according to the number of 1-entries in the columns i+1 to k, such that the variable with the highest number of 1-entries in M becomes the first element.
   For all $Q \in$ V do
   - 3.1 Generate all paths of M with initial part $q \oplus (Q)$ at column i+1;
   - 3.2 Add these paths to Q;
   - 3.3 Cancel the row Q from M.
4. Return Q.

The following example shows that step 5 of the Transform algorithm is in fact necessary, as the paths generated by steps 1 until 4 still may contain redundancy:

3.2.5 Example:

Let M be the following matrix:

M=

	1	2	3	4
P	1	1	0	0
Q	0	1	1	0
R	1	0	0	1
S	0	0	1	1

Then, without removing subsumed paths, the algorithm produces the paths (P,S), (P,Q,R), and (Q,R), from which the second path is subsumed by the third.

In the following we consider some of the algorithm's elementary properties. $\mathcal{P}$(M) denotes the set of paths obtained by applying the algorithm to the matrix M. First we will show that $\mathcal{P}$(M) does not contain any redundancy.

3.2.6 Lemma:

Let p,q be paths generated by the algorithm. If p is a permutation of q, i.e. if $\mathbb{L}(p) = \mathbb{L}(q)$, then $p=q$.

Proof: Suppose $p \neq q$ and j is the first index for which $p(j) \neq q(j)$. Since q is a permutation of p, there is a $i \neq j$ with $q(j)=p(i)$. Suppose $i<j$. Since j is the first index, such that $p(j) \neq q(j)$ we have $q(j)=p(i)=q(i)$, but according to step 2 of the generate algorithm, q cannot have multiple occurrences of a literal, which is a contradiction. Hence we have $i>j$. In the same way we get $p(j)=q(m)$ for some $m>j$. Therefore p and q have the following form:

	1	j-1	j	i
$p =$	$(p(1),\ldots,$	$p(j-1),$	$p(j),\ldots,$	$p(i),\ldots)$
$q =$	$(p(1),\ldots,$	$p(j-1),$	$p(i),\ldots,$	$p(j),\ldots)$
	1	j-1	j	m

The paths p and q both have been developed by generating all paths with initial part $(p(1),..,p(j-1))$ at column j. Without loss of generality we may assume that p was developed before q by the algorithm. Then by step 3.1 first all paths beginning with $(p(1),..,p(j-1),p(j))$ have been developed, and then, according to step 3.3, the row $p(j)$ has been removed from M. After this removal of $p(j)$, however, there is no way to develop the path q, which contains $p(j)$, and this is a contradiction. ■

Thus two different paths produced by the algorithm indeed correspond to two different clauses.

3.2.7 Remark:

Lemmata 2.3 and 2.4 together show that for each complete path p of a matrix M there is a path $q \in \mathcal{P}(M)$ with $q \leq p$.
Step 4 of the algorithm assures that $p \leq q$ implies $p=q$ for arbitrary $p,q \in \mathcal{P}(M)$.

The next lemma justifies step 2 of the transformation algorithm. It says that by developing at first a row with a maximal number of 1-entries, no more paths are produced than by developing any other row that is not maximal. Example 3.2.1 showed that the converse is not true in general: if the S-row is developed at first, the result contains fewer paths than by developing first the P-row.

3.2.8 Lemma:

Let P and Q be rows of an n×k NF-matrix M such that

$$\sum_{i=1}^{k} M(Q,i) < \sum_{i=1}^{k} M(P,i).$$

Suppose $\mathcal{P}$ is the set of paths obtained by developing the row P at first and $\mathcal{Q}$ is the set of paths obtained by developing the row Q at first. Then for any $P \in \mathcal{P}$, there exists some $Q \in \mathcal{Q}$ such that Q is only a permutation of P.

Proof: Without loss of generality we permute the columns of M in such a way, that we have first the set C_1 of columns i with M(p,i)=M(q,i)=1, then C_2 with M(p,i)=1, M(q,i)=0, then C_3 with M(p,i)=0, M(q,i)=1, and last C_4 with M(p,i)=M(q,i)=0. From $\sum_{i=1}^{k} M(Q,i) < \sum_{i=1}^{k} M(P,i)$ follows that $C_2 \neq \emptyset$. Furthermore we assume that subsumed columns in M are already deleted. Then we can write M in the following form:

	C1	C2	C3	C4
P	1...1	1...1	0...0	0...0
Q	1...1	0...0	1...1	0...0
...		.....		

Let P be any path in $\mathcal{P}$.

Case I: P has the form (p,q,....), where p is taken from C_1 and q is taken from C_3.

Exchanging the first two elements of P by taking first q from C_1 and then p from C_2 yields a path $Q \in \mathcal{Q}$ which is only a permutation of P.

Case II: P has the form $(p,x_1,...,x_m,y_1,...,y_n)$, where p is taken from C_1, the x_i are taken from C_3 and the y_j are taken from C_4. We show that the same path P is in $\mathcal{Q}$:

After the development of the p-row this row has been deleted according to step 3. Let M' be the resulting matrix:

M' =

	C1	C2	C3	C4
P	1...1	1...1	0...0	0...0
X_1			1	
:				
X_m			1	
Y_1				1
:				
Y_n				1
...	... ...			

There is only one possibility that the path $P=(p,x_1,...,x_m,y_1,...,y_n)$ is not developed in Q: one column j of the set C_3 or the set C_4, from which one of the x_i or y_j of P has been taken, has been canceled in step 2 of the transform algorithm.
The subsuming column h, against which j has been canceled, must be from C_3 or C_4, since $M(p,j)=0$ and $M(p,i)=1$ for all $i\in C_1\cup C_2$. Suppose that $j\in C_3$. This implies $M(q,j)=1$ and so we have $M(q,j) \geq M(q,h)$. Furthermore we have $M(u,j) \geq M(u,h)$ for all $u\neq q$, since h subsumes j in M'. Together we have $M(u,j) \geq M(u,h)$ for all $u\in \mathbb{L}(M)$, i.e. h subsumes j already in M and this is a contradiction. Thus j must be from C_4 and there must be some y_s with $M(y_s,j)=1$. By an analogous argument h must be from C_3. There must be an x_r such that $M(x_r,h)=M'(x_r,h)=1$. Since j is subsumed by h, we have $M'(x_r,j)=1$, hence also $M(x_r,j)=1$. Thus we have the following situation:

$M=$

	... h...	j..
:		
x_r	1	1
:		
y_s		1
:		

But in this situation the path $P=(....x_r,...,y_s,...)$ could not have been developed according to step 2 of the generate algorithm. Therefore the path P must be in Q. ■

Next we show that our algorithm produces the set of prime implicants of the given formula.

3.2.9 Lemma:
Let p and q be resolvable paths from $\mathcal{P}(M)$ and let R be a nontautological resolvent of p and q.
Then there is an $R'\in \mathcal{P}(M)$ with $R'\leq R$.

Proof: W.l.o.g. let $p=(R,U_1,..,U_i,W_1,..,W_j)$ and $q=(\neg R,U_1,..,U_i,S_1,..,S_k)$, $i,j,k\geq 0$. Then we have $R=(U_1,..,U_i,W_1,..,W_j,S_1,..,S_k)$. We define $U=\{U_1,..,U_i\}$, $S=\{S_1,..,S_k\}$ and $W=\{W_1,..,W_j\}$.
We have to show that there is a path R' in $\mathcal{P}(M)$ such that $\mathbb{L}(R')\subseteq \mathbb{L}(R)=U\cup W\cup S$. Let c be any column of M with $M(R,c)=1$. Then

$M(\neg R,c)=0$, since otherwise the column c would be tautological. Since $q=(\neg R,U_1,..,U_i,S_1,..,S_k)$ is a path through M and $M(\neg R,c)=0$, there must be an $X\in U\cup S$ with $M(X,c)=1$.
Analogously if d is any column of M with with $M(\neg R,d)=1$, then there must be a $Y\in U\cup W$ with $M(Y,d)=1$.
The same argument shows that for any column e with $M(R,e)=M(\neg R,e)=0$ there must be a $Z\in U\cup W$ with $M(Z,e)=1$ and a $Z'\in U\cup S$ with $M(Z',e)=1$.
Now we have shown that for each column c of M there exists a $P\in U\cup W\cup S$ such that $M(P,c)=1$. This implies that there is a complete path p through M with $\mathbb{P}(p)\subseteq U\cup W\cup S$. Together with Remark 3.2.7 this implies that there is a path $R'\in\mathcal{P}$ with $\mathbb{L}(R')\subseteq\mathbb{L}(p)\subseteq U\cup W\cup S$. ■

Lemma 3.2.9 shows that the method of iterated consensus applied to a set $\mathcal{P}(M)$ does not change $\mathcal{P}(M)$. Therefore we have the following

3.2.10 Corollary:

$\mathcal{P}(M)$ is the set of prime implicants of $t_C(M)$. ■

Since the set of prime implicants of an unsatisfiable propositional formula is empty, we have the following

3.2.11 Corollary:

Let $\mathcal{F}$ be an unsatisfiable ground formula and M the NF-matrix of $\mathcal{F}$. Then $\mathcal{P}(M)=\emptyset$. ■

We described the conversion between conjunctive and disjunctive normal form, being the basic step of the clausal normal form algorithm. The transformation of an arbitrary formula $\mathcal{F}$ into clausal normal form (or disjunctive normal form, respectively) starts with the innermost terms of $\mathcal{F}$ and multiplies them by using successively the basic algorithm until the desired normal form is achieved.

The algorithm described in this paper is similar to the algorithm presented in Slagle, Chang & Lee (1970). There are, however, two substantial differences between the two approaches. First, their algorithm's basic data structures are semantic trees, while the data structures used here are matrices. The use of matrices makes our method very suitable for application in the matrix methods for automated theorem proving. All the features described in this paper, like the frequency ordering of literals or the canceling of rows

and columns under certain conditions, directly apply to the matrix approach, resulting in simplification or avoidance of redundancy. Another improvement with regard to the semantic trees method is the cancellation of subsumed columns of the matrix, which is not present in Slagle, Chang & Lee's algorithm and which can save a considerable amount of time.

There is another method to minimize the clausal form of a formula, which is described for instance in Lewis & Papadimitriou (1981) and Gallier (1986). Their method is based on a linear transformation using additional variables, and it avoids the exponential increase in the size of the formula, when multiplied in clausal form. The issue of this thesis, however, is deleting redundancies (i.e. subsumed clauses) rather than minimizing the length of the resulting clause set. The removal of subsumed clauses is usually accomplished in later stages of the proof of a formula, and it is known to be very expensive there. These redundancies, however, are still present in the clause set T that is produced by the linear transformation, since all clauses of the totally multiplied form of a formula can be obtained by resolving clauses of T.

The multiplication algorithm described in this section has been implemented in the MKRP theorem prover (Raph 1984). It performs particularly well for examples with nested equivalences, as it is the case with Andrew's example (Henschen 1980). For instance a formula of the form $f_1 \Leftrightarrow f_2 \Leftrightarrow \ldots \Leftrightarrow f_{n+1}$ with n nested equivalences will result in 2^n clauses in the best case of transformation and in 4^n clauses in the worst case. Our algorithm always produces the minimal number of 2^n clauses.

4 Eliminating Redundant Clauses

4.1 The Use of Subsumption in Automated Reasoning Systems

The derivation of redundant information is one of the greatest obstacles to the efficiency of automated reasoning programs. Even very small problems can become intractable due to the increasing amount of redundant clauses during the proof. The main problem with redundant clauses is their inheritance property. This means that any derivative of a redundant clause itself is redundant. The following example, while being somewhat artificial, is nevertheless very instructive.

4.1.1 Example:

Let n be any even natural number, and let S be the set consisting of the two clauses

C_1: {Px, Pfx}

C_2: {¬Py, ¬Pf^ny}

The set S is unsatisfiable, which is easily proved for n=2 or n=4. Already for n=6, however, many redundant clauses are derived in the proof, and for n>6 the problem is nearly intractable, since the computer resources are swamped with thousands of redundant clauses.

Many other examples show that an approach, which retains all derived information, even the redundant one, usually results in a combinatorial explosion or leads reasoning programs to fruitless paths by concentrating on the redundant information. A means to detect and to discard redundant information thus seems indispensible for solving even small problems[1]. The former task has turned out to be very difficult, and it will be the issue of this chapter. Since in general the test on redundancy must be repeated very often during a refutation, its efficiency is crucial for its use.

Redundant information comes under different logical forms. The two most common types of redundancy are *tautology* and *subsumption*. Since

[1] Overbeek & Wos (1989) provide a broad discussion on the value of procedures removing redundant information, compare also Wos (1988).

detecting tautologies causes no serious problems, we shall deal only with subsumption. The formulation of subsumption is due to J. Robinson (1965a), and it is assessed by Overbek & Wos (1989) to be his most important contribution to automated reasoning, even with regard to his formulation of binary resolution (J. Robinson 1965a) and hyperresolution (J. Robinson 1965b). In fact, practical experience over decades has shown that subsumption, besides demodulation, is the most important means to discard redundant information, such as duplicates or instances of already retained clauses.

The subsumption test has proved to be very expensive in its most general form. In fact, testing subsumption requires a matching process under the theory of associativity, commutativity, and idempotence, which is NP-complete, as Kapur & Narendran (1986) remark. Section 4.4 will give a brief account of several subsumption algorithms that have been developed until now. On account of the subsumption test's complexity, several less complex variants of subsumption can be employed in automated reasoning programs.[1] For instance, the potential subsumers can be restricted to unit clauses. This is a very common variant, which comes under linear complexity. It may also suffice, for instance, to employ the particular kind of subsumption that merely tests whether two clauses are copies of each other. This test will be called *variant test* for short. Being itself NP-complete, the variant test nevertheless allows for refinements that are particularly efficient for the most common cases of clauses with few variables or clauses consisting of few literals. Such a variant test will be presented in the next section. One of our previous results was that two clauses are subsumption equivalent, iff their irreducible factors are variants of each other. In view of this result, it turns out that the variant test in fact amounts to a test on subsumption equivalence, provided that all clauses are completely reduced. The following example shows the effect of the combined reduction to irreducible factors and subsequent elimination of variants.

---

[1] The question, which type of subsumption to select, however, seems by no means to be answered, this is the issue of one of Wos' (1988) *Basic Research Problems*.

4.1.2 Example:

The multiplication of a formula to conjunctive/disjunctive normal form may produce parts of formulae, which are equal up to renaming. In this example the variant test suffices to remove all redundant parts.
The following formula

$\forall x,y,z\ (x \leq y \wedge x \leq z) \vee (y \leq x \wedge y \leq z) \vee (z \leq y \wedge z \leq x)$

is valid in totally ordered sets. Its clausal form consists of the clauses $C_1,\ldots,C_8$ with

$C_1 = \{x \leq y, y \leq x, z \leq y\}$, $C_2 = \{x \leq y, y \leq x, z \leq x\}$, $C_3 = \{x \leq y, y \leq z, z \leq y\}$,
$C_4 = \{x \leq y, y \leq z, z \leq x\}$, $C_5 = \{x \leq z, y \leq x, z \leq y\}$, $C_6 = \{x \leq z, y \leq x, z \leq x\}$,
$C_7 = \{x \leq z, y \leq z, z \leq y\}$, $C_8 = \{x \leq z, y \leq z, z \leq x\}$.

First of all, reducing each C_i to its irreducible factor C_i' yields:

$C_1' = \{x \leq y, y \leq x\}$, $C_2' = \{x \leq y, y \leq x\}$, $C_3' = \{y \leq z, z \leq y\}$,
$C_4' = \{x \leq y, y \leq z, z \leq x\}$, $C_5' = \{x \leq z, y \leq x, z \leq y\}$, $C_6' = \{x \leq z, z \leq x\}$,
$C_7' = \{y \leq z, z \leq y\}$, $C_8' = \{x \leq z, z \leq x\}$.

In the next step, the variant test recognizes C_2', C_3', C_6', C_7', C_8' as variants of C_1', and C_5' as variant of C_4'. Removing the redundant parts results in the clause set

$S = \{\{x \leq y, y \leq x\}, \{x \leq y \vee y \leq z \vee z \leq x\}\}$

An algorithm that accomplishes the task to produce the irreducible factor of a given clause will be given in section 4.3. Finally, in section 4.4 we will show that the notions and methods of 4.2 also prove useful for the general subsumption test.

4.2 A Variant Test Based on Characteristic Matrices

The variant test can be seen as a generalization of the well known (directed) graph isomorphism problem, that is, the problem to decide whether there is a bijective homomorphism mapping one given graph to another. If a directed graph G consisting of k nodes is represented as a set of ordered pairs (i,j), with $i,j \in \{1,\ldots,k\}$, a clause C_G corresponding to G can be constructed which contains the literal Px_ix_j iff G contains the ordered pair (i,j). Then obviously the directed graphs G and G' are isomorphic iff the corresponding clauses C_G and $C_{G'}$ are isomorphic. The most common technique to solve the graph isomorphism problem can be sketched as follows: The basic idea is to test all

bijective mappings $\phi:\{1,\ldots,k\}\rightarrow\{1,\ldots,k\}$, whether they yield a homomorphism $\tilde{\phi}:G\rightarrow G'$, that is, whether $(i,j)\in G$ implies $(i\phi,j\phi)\in G'$. Since the number of those bijective mappings is k!, heuristic techniques have been used to restrict the number of mappings that must be considered in this problem (see, for instance, Unger (1964), or Berztiss (1973)). This techniques all are based on the observation that there are several *invariant* properties of directed graphs that must be preserved by an isomorphism. Unger, for instance, states that the pair (*indegree, outdegree*)[1] is such an invariant property of the nodes.

The analogy between the graph isomorphism problem and the variant test suggests that the well known heuristic techniques to restrict the possible pairings of nodes could also prove useful to limit the number of pairings of variables in the variant test. In the following we shall adopt Unger's invariant to develop an appropriate variant test. Throughout this section, all occurring clauses will be assumed to be irreducible.

Three generalizations have to be made in order to extent the solution of the graph isomorphism problem to an algorithm that detects variants: First, the occurrence of constants and functions has to be considered; second, literals with more than two variables have to be dealt with; and finally, one has to take into consideration that several different predicate symbols can occur in clauses.

Before giving the exact definitions, the basic notions and methods shall be introduced informally by means of an example. Let $C=\{L_1,L_2\}$ with L_1=P(fx,gy) and L_2=Pyx, and let $D=\{K_1,K_2\}$ with K_1=P(fu,gz) and K_2=Puz. If we want to apply the techniques to solve the graph isomorphism problem, we first have to get rid of the function symbols. This can be done by introducing new predicate symbols, say a binary predicate Q, such that Qv_1v_2 stands for $P(fv_1,gv_2)$. The symbol Q represents the "term skeleton" P(f(*),g(*)). Note, however, that the literal P(fx,gx) is transformed into a

[1] Indegree means "number of incoming links", outdegree means "number of outgoing links".

literal R(x) with a unary "skeleton".[1] That this transformation is correct, can be seen from the fact that two literals L and K are equal up to renaming, iff they have the same term skeletons. In our example we obtain $C^* = \{Qxy, Pyx\}$, and $D^* = \{Quz, Puz\}$. These two clauses can be represented by two graphs, whose arcs are labeled with the predicate symbols P and Q (see figure 4.1)

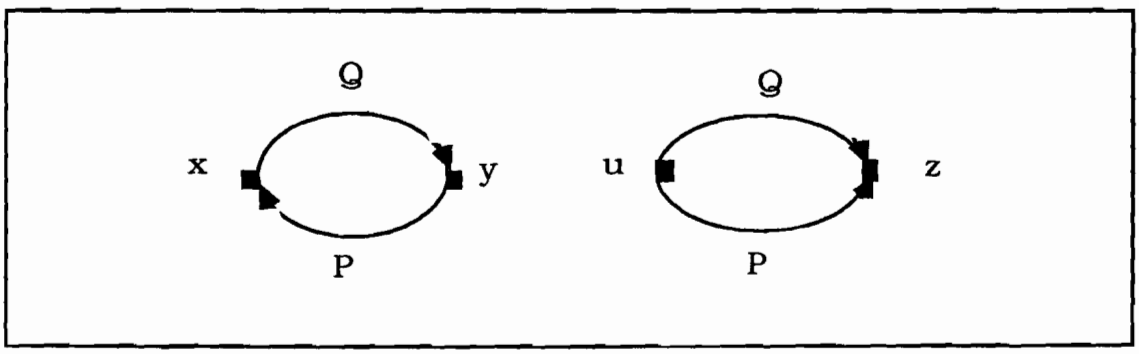


fig. 4.1

Next we compute for each variable $v \in \mathbb{V}(C)$ the (characteristic) pair $\chi_{P,C}(v) = (\text{indegree}(v,P), \text{outdegree}(v,P))$ and $\chi_{Q,C}(v) = (\text{indegree}(v,Q), \text{outdegree}(v,Q))$, where indegree(v,P) denotes the number of P-arcs outgoing from v, and analogously for the other symbols. Since the clauses are variable disjoint, the indices C and D can be omitted. Abbreviating $\chi_{P,C}(x)$ by x_P, we obtain

$x_P = (0,1), x_Q = (1,0), y_P = (1,0), y_Q = (0,1)$

$u_P = (1,0), u_Q = (1,0), z_P = (0,1), z_Q = (0,1)$

Since the two characteristic pairs for x do not match the characteristic pairs of any variable of D, it can already be decided that the variant property does not hold. It can, however, also be useful to define a characteristic for literals, by just joining the characteristic pairs of the corresponding variables, which can be represented by matrices as follows:

---

[1] A similar notion is that of the *rigid part* of a literal, see Nicolaita (1989). The difference to the notion of a term skeleton is that the rigid part does not mirror multiple occurences of variables, whence the rigid part of P(fx,gx) is P(f(*),g(*)).

$$(L_1)_P = (Qxy)_P = [(x_P)^T,(y_P)^T]^1 = \begin{bmatrix}0&1\\1&0\end{bmatrix}, \text{ or}$$

$$(K_1)_P = (Quz)_P = \begin{bmatrix}1&0\\0&1\end{bmatrix} = (K_2)_P$$

$(L_1)_P$ does not match neither the characteristic matrix $(K_1)_P$ nor $(K_2)_P$, which also indicates that the variant property does not hold.

In the following the previous informal description of the algorithm shall be made more precise.

Let $\mathcal{L}$ be a set of literals. For any $L \in \mathcal{L}$, let [L] denote the equivalence class $\{L' \in \mathcal{L} \mid L' \cong L\}$. If Λ is any such equivalence class, we take an arbitrary $L \in \Lambda$. We define the literal L^* by $L^* = P_\Lambda(x_1,\ldots,x_n)$, where $\{x_1,\ldots,x_n\} = \mathbb{V}(L)$. For any $K \in \Lambda$, we define $K^* = P_\Lambda(x_1\rho,\ldots,x_n\rho)$, if ρ is the renaming that maps L to K. This transformation is extended to clauses by $C^* = \{L^* \mid L \in C\}$.

4.2.1 Example:

Let L = P(f(x, g(y)), h(z)), K = P(f(u, g(v)), h(w)), and M = P(f(u, g(v)), h(u)). Since [L] = [K] ≠ [M], we have $L^* = P_{[L]}(x,y,z)$, and $K^* = P_{[L]}(u,v,w)$, where $P_{[L]}$ can be interpreted as the term "skeleton" P(f(*, g(*)), h(*)). For the literal M we obtain $M^* = P_{[M]}(u,v)$.

4.2.2 Lemma:

Let L and K be literals, and let $\rho \in \mathbb{P}$. Then

$L\rho = K$ iff $L^*\rho = K^*$.

Proof: Let $L^* = P_{[L]}(x_1,\ldots,x_n)$. Then $L\rho = K$ iff [L] = [K] and $K^* = P_{[K]}(x_1\rho,\ldots,x_n\rho)$ iff $L^*\rho = K^*$. ■

From the previous lemma follows easily that $C \cong D$ holds for two clauses C and D, if and only if $C^* \cong D^*$ holds. L^* is a positive literal without function and constant symbols. Moreover, it has the additional property that the variables of its argument list are pairwise distinct. Terms or literals of this kind are also called *linear*. The transformation $C \rightarrow C^*$ thus reduces the problem to the elementary case of positive linear literals, and we shall assume in the following that all occurring literals are of this particular form.

---

[1] The operation ^{T} denotes the transposition of matrices. Transposing a row vector yields a column vector.

Some additional assumptions can be made for the test, whether the clause C is a variant of the clause D. $|\mathbb{V}(C)| = |\mathbb{V}(D)|$ as well as $|C| = |D|$ are necessary, and also easy to test, conditions for $C \cong D$. Throughout the rest of this section, it will thus be assumed that both conditions are satisfied. Moreover, we shall assume that $|\{L \in C \mid \mathbb{P}(L) = P\}| = |\{K \in D \mid \mathbb{P}(K) = P\}|$ holds for each predicate symbol P occurring in $C \cup D$.

The following definition introduces the notion of a characteristic function of a clause, which will be the desired invariant analogon to the pair (indegree, outdegree) for directed graphs.

4.2.3 Definition:

Let C and D be (not necessarily distinct) clauses.

a) For any literal $L = Px_1 \ldots x_n \in C$ we define the function

$$\varphi_{L,C} : \mathbb{V}(C) \to \{0,1\}^n \text{ by}$$

$$(\varphi_{L,C}(x))_j = \begin{cases} 1 & \text{if } x = x_j \\ 0 & \text{otherwise}^1 \end{cases}$$

b) For any n-ary predicate symbol P occurring in C we define the function

$$\chi_{P,C} : \mathbb{V}(C) \to \mathbb{N}^n \text{ by}$$

$$\chi_{P,C}(x) = \sum_{\mathbb{P}(L)=P} \varphi_{L,C}(x),$$

where the addition on tuples is defined pointwise.

c) For any n-ary predicate symbol P occurring in C we define the function

$$\chi_{P,C} : C \to \mathbb{N}^{n \times m} \text{ by}$$

$$\chi_{P,C}(L) = (\chi_{P,C}(x_1)^T, \ldots, \chi_{P,C}(x_m)^T),$$

where L is of the form $Qx_1 \ldots x_m$.

d) We define a relation $\approx_{C,D}$ on $\mathbb{V}$ by

$x \approx_{C,D} y$ iff $\chi_{P,C}(x) = \chi_{P,D}(y)$ for all P occurring in $C \cup D$.

The relation $\approx_{C,C}$ will be written $\approx_C$. Analogously, a relation $\approx_{C,D}$ can be defined for literals, with the additional requirement that $\mathbb{P}(L) = \mathbb{P}(K)$ must hold for $L \approx K$.

The function χ_C is called the **characteristic** (function) of C. $\chi_{P,C}(x)$ is an n-tuple of natural numbers, and the k-th component of this n-tuple denotes the number of occurrences of the variable x in the n-th coordinate position

---

[1] Note that a_j denotes the j-th component of the vector a.

in literals with predicate P in the clause C. Suppose the clause C contains only one single predicte symbol P and arity(P)=2 holds. Then the first component of $\chi_{P,C}(x)$ denotes the number of literals of the form P(x,*). In the notation of directed graphs, this is just the number of links outgoing from the node x. Similarly, the second component of $\chi_{P,C}(x)$ denotes the number of links incoming to the node x.

The index referring to a clause will be dropped in the following. The relation $\approx_{C,D}$ is an equivalence relation, and the equivalence class of x modulo $\approx_{C,D}$ will be denoted by $[x]_{C,D}$.

In the following we shall use the abbreviation $\Sigma^* a$ for $\sum_{i=1}^{n} a_i$, if a is an n-tuple $(a_1,\ldots,a_n)$.

4.2.4 Example:

Let $C = \{L_1, L_2, L_3, L_4\}$, where L_1=Pxy, L_2=Pyz, L_3=Pxz, L_4=Qzx.
Then we have

$$\chi_P(x) = \varphi_{L_1}(x) + \varphi_{L_2}(x) + \varphi_{L_3}(x) = (1, 0) + (0, 0) + (1, 0) = (2, 0)$$

and analogously $\chi_P(y) = (1, 1)$, $\chi_P(z) = (0, 2)$, $\chi_Q(x) = (0, 1)$ etc.
The characteristics of literals are represented by matrices, for instance:

$$\chi_P(L_1) = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}, \chi_P(L_2) = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix}, \chi_P(L_3) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

The characteristic function is indeed a property that is invariant under renamings, as the following lemma shows:

4.2.5 Lemma:

Let C and D be clauses and let $\rho \in \mathbb{P}$, such that $C\rho=D$. Then

a) $\varphi_{L,C}(x) = \varphi_{L\rho,D}(x\rho)$ holds for each $L \in C$, $x \in \mathbb{V}(C)$.
b) $x \approx x\rho$ holds for each $x \in \mathbb{V}(C)$.
c) $L \approx L\rho$ holds for each $L \in C$.

Proof: a) Let $L = Px_1\ldots x_n$. Then $L\rho = Px_1\rho\ldots x_n\rho$. We have

$(\varphi_{L,C}(x))_j = 1$ iff $x = x_j$ iff $x\rho = x_j\rho$ iff $(\varphi_{L\rho,D}(x\rho))_j = 1$.

b) follows from a) and the fact that

$$|\{L \in C \mid \mathbb{P}(L)=P\}| = |\{K \in D \mid \mathbb{P}(K)=P\}|$$

holds for each predicate symbol P occurring in $C \cup D$.
c) follows from b) ∎

However, the characteristic function's invariance under the renaming ρ is only a necessary, but not a sufficient condition for the renaming ρ to map C onto D. Consider, for instance, the clauses C = {Pxy, Pyz, Pyu, Pux} and D = {Px'y', Py'x', Pz'u', Pu'z'}. For all $v,w \in \mathbb{V}(C \cup D)$ we have $v \approx_{C,D} w$, but nevertheless there exists no renaming ρ such that $C\rho = D$ holds.

Under certain additional restrictions, however, the characteristic function's invariance is also a sufficient condition for the clauses C and D to be variants. If we require, for instance, that no two variables of C have the same characteristic, then it suffices to compare the characteristics of the variables of C with the characteristics of the variables of D in order to decide the variant property. This requirement can also be formulated in the following form: The equivalence classes of $\mathbb{V}(C)$ modulo $\approx$ have to be singletons. The requirement above can still be weakened, which is shown in the following two lemmata.

For any $x \in \mathbb{V}$, we define O(x,C) to be the total number of occurrences of x in C, that is, $O(x,C) = \sum_{L \in C} \Sigma^* \varphi_L(x)$.

4.2.6 Lemma:

Let C be a clause and let $x \in \mathbb{V}(C)$. Then the total number of occurrences of variables $y \approx x$ in C is

$$|[x]_C| \sum_P \Sigma^* \chi_P(x),$$

where the first sum is taken over all predicate symbols P occurring in C.

Proof: $\Sigma^* \chi_P(x)$ is the number of occurrences of x in literals $L \in C$ with $\mathbb{P}(L)=P$. Then $\sum_P \Sigma^* \chi_P(x)$ is the number of occurrences of x in all literals $L \in C$.

From $\chi_P(y) = \chi_P(x)$ for all y with $y \approx x$ follows $\Sigma^* \chi_P(x) = \Sigma^* \chi_P(y)$ for all $y \in [x]_C$, and for each predicate P. From this follows the assertion of the lemma. ■

4.2.7 Lemma:

Let C and D be irreducible clauses with $|C| = |D|$. Suppose $|[x]_C| = 1$ holds for each $x \in \mathbb{V}(C)$ that satisfies $O(x,C)>1$. Then

$C \cong D$, iff for each $L \in C$ there is a $K \in D$ with $L \approx K$.

Proof: If $C\rho=D$ holds for some renaming ρ, then $L \approx L\rho$ holds for each $L \in C$, according to lemma 4.2.5.

In order to prove the converse direction, we first show that under the assumptions of the lemma the equivalence classes of C modulo $\approx$ are singletons: If this is not the case, then there are $L,K \in C$, with $L \neq K$ but $L \approx K$, which implies $\mathbb{P}(L) = \mathbb{P}(K)$. Let $\mathbb{P}(L) = P$, and let $L = Px_1 \ldots x_n$ and $K = Py_1 \ldots y_n$. Then there is $j \in \{1,\ldots,n\}$ such that $x_j \neq y_j$ but $x_j \approx y_j$. W.l.o.g. we can assume that there is exactly one such $j \in \{1,\ldots,n\}$. From $x_j \approx y_j$ follows that $y_j \in [x_j]_C$, hence $[x_j]_C > 1$. The assumptions of the lemma imply that $O(x_j,C) \leq 1$, and since x_j occurs in L, we have $O(x_j,C) = 1$. Let $\tau = \{x_j \rightarrow y_j\}$. Since $x_i = y_i$ holds for $i \neq j$, we have $L\tau = K$, and $L'\tau = L'$ for each $L' \neq L$, since $x_j \notin \mathbb{V}(L')$ holds for $L' \neq L$. But this implies $C\tau = C \setminus \{L\}$, which contradicts the irreducibility of C. Thus we have proved that $L \approx K$ iff $L = K$.

Next we construct a renaming ρ with $C\rho = D$. Let L be any literal of C. Then there is a unique literal $L' \in D$ with $L' \approx L$, and, conversely, L is the only literal in C with $L \approx L'$. The mapping $L \rightarrow L'$ thus yields a bijective function $\tilde{\rho}: C \rightarrow D$. We show that $\tilde{\rho}$ induces a renaming $\rho: \mathbb{V}(C) \rightarrow \mathbb{V}(D)$. For any variable x we take an arbitrary literal L of C with $x \in \mathbb{V}(L)$, say $L = Px_1 \ldots x_n$ with $x = x_j$. Moreover, let $L\tilde{\rho} = Py_1 \ldots y_n$. Then we define $x_j\rho = y_j$. From the condition $L \approx L\tilde{\rho}$ we obtain $x_i \approx y_i$ for each $i \in \{1,\ldots,n\}$, that is, $\chi_P(x_i) = \chi_P(y_i)$ for each predicate symbol P occurring in C, and each $i \in \{1,\ldots,n\}$. From this follows that for any $x \in \mathbb{V}(C)$

$$|[x]_C \cap \mathbb{V}(L)| = |[x\rho]_D \cap \mathbb{V}(L\tilde{\rho})|$$

that is, the total number of occurrences of variables $y \approx x$ in L equals the total number of occurrences of variables $y' \approx x\rho$ in $L\tilde{\rho}$, and hence

$$\sum_{L \in C} |[x]_C \cap \mathbb{V}(L)| = \sum_{L \in C} |[x\rho]_D \cap \mathbb{V}(L\tilde{\rho})|,$$

that is, the total number of occurrences of variables $y \approx x$ in C equals the total number of occurrences of variables $y' \approx x\rho$ in D. According to the previous lemma, these numbers can also be computed by $|[x]_C| \sum_P \Sigma^* \chi_{P,C}(x)$ and $|[x\rho]_C| \sum_P \Sigma^* \chi_{P,D}(x\rho)$, respectively, from which follows that

$$|[x]_C| \sum_P \Sigma^* \chi_{P,C}(x) = |[x\rho]_C| \sum_P \Sigma^* \chi_{P,D}(x\rho)$$

Since $\Sigma^* \chi_{P,C}(x) = \Sigma^* \chi_{P,D}(x\rho)$ holds for each P, we obtain

$$|[x]_C| = |[x\rho]_D|.$$

Now we show that ρ is well defined. Assume that x occurs in L_1 and in L_2, say $L_1 = Px_1 \ldots x_n$, and $L_2 = Qy_1 \ldots y_m$ with $x = x_j = y_k$. From the construction of ρ we

have $L_1 \approx L_1\rho$ and $L_2 \approx L_2\rho$, hence $x=x_j \approx x_j\rho$ and $x=y_k \approx y_k\rho$, which implies $x_j\rho \approx y_k\rho$.
Case 1: If $|[x_j]_C| = 1$, then also $|[x_j\rho]_D| = 1$. From the previous result follows $x_j\rho = y_k\rho$ and ρ is thus well defined.
Case 2: If $|[x_j]_C| > 1$, then $O(x_j,C) \leq 1$, and, since x_j occurs in L_1, $O(x_j,C) = 1$ must hold. Then $L_1=L_2$ and again ρ is well defined.
Finally we have to show that ρ is a renaming, that is, ρ is bijective on its domain. Since $\mathbb{V}(C)$ and $\mathbb{V}(D)$ are finite sets of equal length, it suffices to show that ρ is surjective. But this is clear, since $\tilde{\rho}$ is a bijective extension of ρ on C, and if ρ were not surjective, then $\tilde{\rho}$ could not be surjective either. ■

Criteria that allow to replace the search for an appropriate variable pairing by a mere comparision of the characteristics are of great value, as the following example shows:

4.2.8 Example:

Let $C=\{L_1,L_2,L_3\}$, with $L_1=Pxy$, $L_2=Pyz$, $L_3=Pxz$, and $D=\{K_1,K_2,K_3\}$, with $K_1=Puv$, $K_2=Pwv$, $K_3=Pwu$. As P is the only predicate symbol, we omit the index P from the characteristics. We have

$$\chi(x) = (2, 0), \chi(y) = (1, 1), \chi(z) = (0, 2),$$

hence the classes $[x]_C$ are singletons for all $x \in \mathbb{V}(C)$. The conditions of the previous lemma thus apply, and from

$$\chi(L_1) = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} = \chi(K_3), \chi(L_2) = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} = \chi(K_1), \chi(L_3) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \chi(K_2)$$

already follows that $C \cong D$. This saves the computation of the 6 possible variable pairings.

In the following it is shown that for clauses with a small number of variables or clauses with few literals the comparision of the characteristic matrices is also sufficient to test the variant property.

4.2.9 Lemma:

Let C and D be clauses. Suppose $|\mathbb{V}(C)| = 2$ and $|\mathbb{V}(D)| = 2$ hold. Then $C \cong D$, iff $\{\chi(x) \mid x \in \mathbb{V}(C)\} = \{\chi(x') \mid x' \in \mathbb{V}(D)\}$ holds.

Proof: Obvious. ■

We will mainly consider the so called **homogeneous** clauses. A clause is homogeneous, if all its literals have the same predicate symbol. Our first

result states that for homogeneous clauses with not more than three variables, the variant property can be decided by merely comparing the characteristics of the variables. Before giving the theorem, we first show some lemmata.

The following assumptions are made throughout the rest of this section: We consider clauses C and D with corresponding variable sets V and W, respectively. If C is homogeneous, then P is the (unique) predicate symbol occurring in C (and also in D), and n is its arity. Since the case $|V|=2$ is trivial, we can assume $|V|\geq 3$. In lemmata 4.2.10 to 4.2.14 we consider the case $|V|=3$, and we shall always assume $V=\{x,y,z\}$.

First, we remark that a homogeneous clause with n=1 is uniquely determined up to renaming. Only the case, where n>1 will thus be considered in the following.

4.2.10 Lemma:

Let $|V|\leq 3$. Then

a) $n\leq 3$

b) $0\leq(\chi_P(x))_i\leq 2$, for each $i\in\{1,\ldots,n\}$.

Proof: a) is clear from $|V|\leq 3$ and the linearity of L.

b) From part a) follows n=2 or n=3. Suppose that $(\chi_P(x))_i\geq 3$ for some $i\in\{1,\ldots,n\}$, say i=1. Then there are different literals L_1, L_2, L_3 all of the form P(x,*,*) or P(x,*). But this is impossible, since there are at most two variables left to fill out the free positions in L_1, L_2, L_3. ■

4.2.11 Lemma:

Suppose $\{\chi(x)\mid x\in V\}=\{\chi(x')\mid x'\in W\}$. If $x\approx y$, and all components of $\chi(z)$ are even, then $C\cong D$.

Proof: Let $W=\{u,v,w\}$, with $u\approx x$, $v\approx y$, $w\approx z$. First we remark that $(\chi(z))_i\in\{0,2\}$ for $0\leq i\leq n$. Let $\rho=\{x\to u, y\to v, z\to w\}$. We have to show that $L\rho\in D$ for each $L\in C$. Let $L=Px_1\ldots x_n\in C$. Furthermore, let

$C'=\{L\in C\mid z\in\mathbb{V}(L)\}$, and $C''=C\setminus C'$,

and let D' and D'' be defined analogously.

a) Suppose $z\in\mathbb{V}(L)$. W.l.o.g. we can assume $z=x_1$. Then $(\chi(z))_1=2$ must hold. Hence there must be some literal $K=Py_1\ldots y_n$, with $K\neq L$ and $z=y_1$. It is easy to see, that there are only the following two possibilities: Either n=2 and L=Pzx

and $K = Pzy$, or n=3 and $L = Pzxy$ and $K = Pzyx$. Since $(\chi(w))_1 = 2$, the same must hold for w, u, and v, that is, either n=2 and $L' = Pwu$ and $K' = Pwv$ are in D, or n=3 and $L' = Pwuv$ and $K' = Pwvu$ are in D. In both cases, $L\rho = L' \in D$.
b) Suppose $z \notin \mathbb{V}(L)$. Then $\mathbb{V}(L)=\{x,y\}$, and, since L is linear, either L=Pxy or L=Pyx. W.l.o.g. let $L = Pxy$. From a) follows that $Pxz \in C$ iff $Pyz \in C$, and $Pzx \in C$ iff $Pzy \in C$, hence $\chi_{C'}(x) = \chi_{C'}(y)$. From $\chi_C(x) = \chi_C(y)$ thus follows $\chi_{C''}(x) = \chi_{C''}(y)$. Now $Pxy \in C''$, hence also $Pyx \in C''$. From a) follows that $|C'| = |D'|$, hence also $|C''| = |D''|$, in particular, there is a literal $K \in D$ with $w \notin \mathbb{V}(K)$. Now the same argument as before applies to D, with the result $Puv \in D$ and $Pvu \in D$, in particular $Puv = L\rho \in D$. ■

4.2.12 Lemma:

Suppose $\{\chi(x) \mid x \in V\} = \{\chi(x') \mid x' \in W\}$. If $(\chi(x))_i = 1$ for all $x \in V$, and all $i \in \{1,\ldots,n\}$, then $C \cong D$.

Proof: Case 1: n=2. It is easy to see that under the assumptions of the lemma either C={Pxy, Pyz, Pzx} or C={Pyx, Pzy, Pxz}, that is, C is determined uniquely up to renaming. The same holds for D, hence $C \cong D$.
Case 2: n=3. It is easy to see that either C={Pxyz, Pyzx, Pzxy} or C={Pzyx, Pxzy, Pyxz}, and again C is determined uniquely up to renaming. ■

4.2.13 Lemma:

Suppose $\{\chi(x) \mid x \in V\} = \{\chi(x') \mid x' \in W\}$. If $|\{\chi(x) \mid x \in V\}| = 3$, that is, if the characteristics of the variables are all distinct, then $C \cong D$.

Proof: Let $W = \{u,v,w\}$ such that $x \approx u$, $y \approx v$, $z \approx w$. Let $\rho = \{x \to u, y \to v, z \to w\}$. We have to show that $L\rho \in D$ for each $L \in C$.
Case 1: n=2. Let L=Pxy, and assume $Puv \notin D$. From $(\chi(x))_1 \geq 1$ and $(\chi(y))_2 \geq 1$ follows $(\chi(u))_1 \geq 1$ and $(\chi(v))_2 \geq 1$. Since $Puv \notin D$, Puw and Pwv must be in D. This implies $(\chi(w))_1 \geq 1$ and $(\chi(w))_2 \geq 1$, hence also $(\chi(z))_1 \geq 1$ and $(\chi(z))_2 \geq 1$. If Pxz would be in C, then $(\chi(u))_1 = (\chi(x))_1 \geq 2$, hence besides Puw there must be another literal $P(u,*) \in D$, but from the other choices $Puv \in D$ contradicts our assumption and Puu is impossible due to the linearity condition. Thus $Pxz \notin C$. Similarly it can be shown that $Pzy \notin D$. Since $z \in \mathbb{V}(C)$, Pzx or Pyz must be in C. In both cases the same argumentation as above applies, yielding $Pyx \notin C$. Altogether we have excluded Pyx, Pxz, and Pzy from occurring in C. Thus either C={Pxy, Pyz}, C={Pxy, Pzx}, or C={Pxy, Pyz, Pzx}. But in the last case we have $\chi(x)=\chi(y)=\chi(z)$, contradicting the assumption $|\{\chi(x) \mid x \in V\}| = 3$.

The first two cases are variants of each other. Hence C is determined uniquely up to renaming.

Case 2: n=3. Let $L = Pxyz$, and assume that $Puvw \notin D$. We have $(\chi(x))_1 \geq 1$, $(\chi(y))_2 \geq 1$, and $(\chi(z))_3 \geq 1$, hence also $(\chi(u))_1 \geq 1$, $(\chi(v))_2 \geq 1$, and $(\chi(w))_3 \geq 1$. Since $Puvw \notin D$, this is only possible, if $Puwv \in D$, $Pwvu \in D$, and $Pvuw \in D$. If D would consist only of these three literals, then we had $u \approx v \approx w$, which contradicts the assumption. Thus there is some other literal, say Pwuv, in D. Then $(\chi(v))_3 \geq 2$, hence also $(\chi(y))_3 \geq 2$. This implies $Pxzy \in C$ and $Pzxy \in C$, hence also $(\chi(x))_1 \geq 2$. From this follows $(\chi(u))_1 \geq 2$, that is, Puvw and Puwv must be in D, which contradicts the assumption. Hence we have proved $Puvw = L\rho \in D$. ■

4.2.14 Theorem:

$C \cong D$ iff $\{\chi(x) \mid x \in V\} = \{\chi(x') \mid x' \in W\}$.

Proof: It suffices to prove that $\{\chi(x) \mid x \in V\} = \{\chi(x') \mid x' \in W\}$ implies $C \cong D$. Let $\xi = \chi(x)$, $\eta = \chi(y)$, and $\zeta = \chi(z)$.

It is easy to verify that the following conditions (1) and (2) must hold:

(1) $\xi_i + \eta_i + \zeta_i = |C|$, for each $i \in \{1,\ldots,n\}$.

(2) $\xi_i + \eta_i - \zeta_k \leq 2$, for each $i,k \in \{1,\ldots,n\}$ such that $i \neq k$.

As to the inequality (2), note that $\xi_i + \eta_i - \zeta_k$ is smaller than the number of literals having x or y in position i, and having z not in position $k \neq i$, that are those literals having x or y in position i and in position k. There are at most two literals of this kind.

With regard to the previous lemma we can assume that $|\{\xi,\eta,\zeta\}| \leq 2$, that is, at least two of these characteristics are equal. W.l.o.g. we assume that $\xi = \eta$.

Case 1: n=2. Then (1) implies

$$2\xi_1 + \zeta_1 = 2\xi_2 + \zeta_2,$$

that is,

$$\zeta_1 - \zeta_2 = 2\xi_2 - 2\xi_1,$$

hence $\zeta_1 - \zeta_2$ is even, which implies that either both ζ_1 and ζ_2 are even or both are odd. If both are even, then we are done by lemma 4.2.11. If both are odd, then $\zeta_1 = \zeta_2 = 1$. Then (2) implies

$$2\xi_1 - 2 \leq 1 \text{ and}$$

$$2\xi_2 - 2 \leq 1.$$

Since $\xi_1 = \xi_2$, ξ_1 cannot be zero. Hence we have

$$\xi_1 = \xi_2 = \eta_1 = \eta_2 = \zeta_1 = \zeta_2 = 1.$$

The assertion of the theorem now follows by lemma 4.2.12.
Case 2: n=3. As in case 1 we obtain that $\zeta_1 - \zeta_2$ is even, $\zeta_2 - \zeta_3$ is even, and $\zeta_1 - \zeta_3$ is even. Hence either ζ_1, ζ_2, and ζ_3 all are even, and then we are done by lemma 4.2.11, or all are odd, which implies $\zeta_1 = \zeta_2 = \zeta_3 = 1$. Again we obtain

$2\xi_1 - 2 \le 1$, $2\xi_1 - 2 \le 1$ and $2\xi_3 - 2 \le 1$,

and the rest is analogous to case 1. ■

The following example shows that an analogous theorem for clauses C with $|\mathbb{V}(C)| = 4$ does not hold.

4.2.15 Example:

Consider the clauses C = {Pxy, Pxz, Pyz, Pzu, Pux} and D = {Pqr, Prq, Ppr, Pps, Psp}. In this example arity(P) = 2 holds, so that the clauses can be depicted by directed graphs, as shown in figure 4.2.

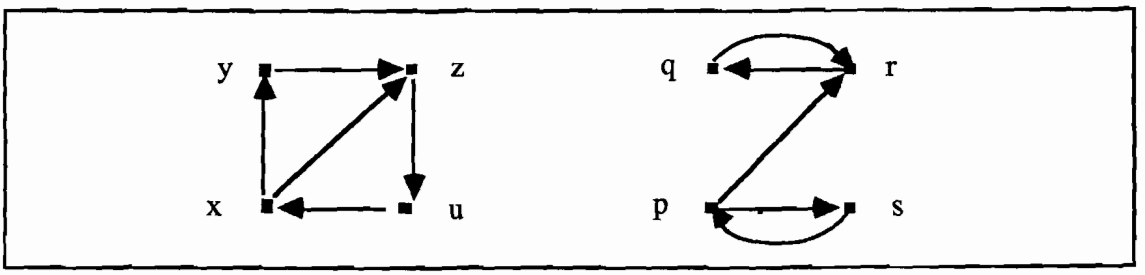


fig. 4.2

It is easy to verify that x≈p, y≈q, z≈r, and u≈s holds. Still, the two graphs are not isomorphic, that is, C≇D.

4.2.16 Theorem:

Let C be a homogeneous clause with $|C| = 3$ and $|[x]_C| < 3$ for all $x \in \mathbb{V}(C)$. Then $C \cong D$ iff $\{\chi(x) \mid x \in V\} = \{\chi(x') \mid x' \in W\}$ and $\{\chi(L) \mid L \in C\} = \{\chi(K) \mid K \in D\}$.

Proof: Let $C = \{L_1, L_2, L_3\}$ and $D = \{K_1, K_2, K_3\}$ such that $L_i \approx K_i$ for i=1,2,3. Let $\tilde{\rho} : C \to D$ be defined by $L_i\rho = K_i$. We show that $\tilde{\rho}$ induces a renaming ρ of $\mathbb{V}(C)$ by $\mathbb{V}(D)$. In analogy to the proof of 4.2.7 it suffices to show that ρ is well defined: Let

$L_1 = Px_1 \ldots x_n$, $L_2 = Py_1 \ldots y_n$, $L_3 = Pz_1 \ldots z_n$,

$K_1 = Px_1' \ldots x_n'$, $K_2 = Py_1' \ldots y_n'$, $K_3 = Pz_1' \ldots z_n'$.

Furthermore, let $\lambda_i = \chi(L_i)$, $\kappa_i = \chi(K_i)$ for i=1,2,3.
Let $x \in V$, let $\xi = \chi(x)$, and suppose $x = x_i = y_j$, for some $i,j \in \{1,\ldots,n\}$. Since $\lambda_1 = \kappa_1$, $x_i' \approx x \approx y_j'$ holds.

If $|[x]_C| = 1$, then from $\{\chi(v) \mid v \in V\} = \{\chi(v') \mid v' \in W\}$ follows that $|[x_i']_D| = 1$, hence $x_i' = y_j'$, that is, ρ is well defined for x.

Now suppose $|[x]_C| \geq 2$, that is, there is $y \in \mathbb{V}(C)$ with $x \approx y$. Then $\chi(y)=\xi$ holds. W.l.o.g. we assume that x (hence also y) occurs in the first three argument positions, that is, $\xi_i=0$ for $i>3$. First we remark that $|C| = 3$ implies $\xi_i+\chi(y)_i \leq 3$, hence $2\xi_i \leq 3$, which in turn implies $\xi_i \leq 1$ for all $i \in \{1,\ldots,n\}$.

Case 1: $O(x,C) = 3$, that is, x occurs three times in C. Together with $\xi_i \leq 1$ for all $i \in \{1,\ldots,n\}$ we obtain $\xi_1=\xi_2=\xi_3=1$. W.l.o.g. we assume $x=x_1=y_2=z_3$. For y we obtain two possibilities, either $y=x_2=y_3=z_1$ or $y=x_3=y_1=z_2$. These two cases are symmetric, and we just assume the first. If $y_1=z_2=x_3$, then $y_1 \in [x]_C$, contradicting the assumption $[x]_C < 3$. Thus

$L_1 = \{P(x,y,x_3,\ldots), L_2 = P(y_1,x,y,\ldots), L_3 = P(y,z_2,x,\ldots),$

with $y_1 \neq z_2$ or $y_1 \neq x_3$ or $z_2 \neq x_3$, and the λ_i have the form

$$\lambda_1 = \begin{bmatrix} 1 & 1 & * & \ldots \\ 1 & 1 & * & \ldots \\ 1 & 1 & * & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix}, \lambda_2 = \begin{bmatrix} * & 1 & 1 & \ldots \\ * & 1 & 1 & \ldots \\ * & 1 & 1 & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix}, \lambda_3 = \begin{bmatrix} 1 & * & 1 & \ldots \\ 1 & * & 1 & \ldots \\ 1 & * & 1 & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix}$$

where in each matrix at least one of the asterisks is equal to 0. Furthermore, $\kappa_i=\lambda_i$ for i=1,2,3. This implies that there are $x',y' \in \mathbb{V}(D)$ such that

$K_1 = P(x',y',x_3',\ldots), K_2 = P(y_1',x',y',\ldots), K_3 = P(y',z_2',x',\ldots).$

with $y_1' \neq z_2'$ or $y_1' \neq x_3'$ or $z_2' \neq x_3'$. This shows that $x\rho=x'$, $y\rho=y'$ and hence ρ is well defined for x (and y).

Case 2: $O(x,C) = 2$. W.l.o.g. we assume that $\xi_1=\xi_2=1$. Then we can distinguish two cases: Either

$L_1 = P(x,y,\ldots), L_2 = P(y,x,\ldots),$ and $x,y \notin \mathbb{V}(L_3)$, or

$L_1 = P(x,y,\ldots), L_2 = P(u,x,\ldots),$ and $L_3 = P(y,v,\ldots).$

In the first case, the λ_i have the form

$$\lambda_1 = \begin{bmatrix} 1 & 1 & \ldots \\ 1 & 1 & \ldots \\ \ldots & \ldots & \ldots \end{bmatrix}, \lambda_2 = \begin{bmatrix} 1 & 1 & \ldots \\ 1 & 1 & \ldots \\ \ldots & \ldots & \ldots \end{bmatrix}, \lambda_3 = \begin{bmatrix} 1 & 0 & \ldots \\ 0 & 1 & \ldots \\ \ldots & \ldots & \ldots \end{bmatrix}$$

Furthermore, $\kappa_i=\lambda_i$ for i=1,2,3. Thus there are $x',y' \in \mathbb{V}(D)$ such that

$K_1 = P(x',y',\ldots), K_2 = P(y',x',\ldots),$ and $x',y' \notin \mathbb{V}(K_3).$

This shows that $x\rho=x'$, $y\rho=y'$ and ρ is well defined for x (and y).

If, on the other hand, $L_1 = P(x,y,\ldots), L_2 = P(u,x,\ldots),$ and $L_3 = P(y,v,\ldots),$ then the λ_i have the form

$$\lambda_1 = \begin{bmatrix} 1\ 1 \ldots \\ 1\ 1 \ldots \\ \ldots\ldots\ldots \end{bmatrix}, \lambda_2 = \begin{bmatrix} 1\ 1 \ldots \\ *\ 1 \ldots \\ \ldots\ldots\ldots \end{bmatrix}, \lambda_3 = \begin{bmatrix} 1\ * \ldots \\ 1\ 1 \ldots \\ \ldots\ldots\ldots \end{bmatrix}$$

where, due to the assumption $[x]_C < 3$, the first two columns of λ_2 (and also of λ_3) are different. Furthermore, $\kappa_i = \lambda_i$ holds for i=1,2,3, hence there are $x', y' \in \mathbb{V}(D)$ such that

$K_1 = P(x',y',\ldots)$, $K_2 = P(*,x',\ldots)$, and $K_2 = P(y',*,\ldots)$,

and again ρ is well defined for x (and y). ■

The following example shows that the condition $|[x]_C|<3$ cannot be dropped from the previous theorem.

4.2.17 Example:

Let C={Pxyzuvw, Pyzxvwu, Pzxywuv} and D={Px'y'z'u'v'w', Py'z'x'w'u'v', Pz'x'y'v'w'u'}. Then

$$\chi(L) = \begin{bmatrix} 111000 \\ 111000 \\ 111000 \\ 000111 \\ 000111 \\ 000111 \end{bmatrix} = \chi(K)$$

holds for each $L \in C$, $K \in D$. We have $\{\chi(x) \mid x \in V\} = \{\chi(x') \mid x' \in W\}$ and $\{\chi(L) \mid L \in C\} = \{\chi(K) \mid K \in D\}$, but $C \cong D$ does not hold.

Of course, theorem 4.2.16 also holds for non homogeneous clauses, possibly containing different predicate symbols. One might suggest that even an anlogon of this theorem holds for clauses, where each homogeneous subclause is of length ≤3. That this is not the case can be seen by the following example:

4.2.18 Example:

Let C={Pxyzw, Pyxwz, Qxyzw, Qyxwz}, and let D={Pxyzw, Pyxwz, Qyxzw, Qxywz}. Then

$$\chi_P(L) = \chi_Q(L) = \begin{bmatrix} 1100 \\ 1100 \\ 0011 \\ 0011 \end{bmatrix} = \chi_P(K) = \chi_Q(K)$$

holds for each $L \in C$, $K \in D$. Still, $C \cong D$ does not hold.

We are now able to use the previous results to define an algorithm that decides whether two given irreducible clauses are variants of each other. Result 4.2.5 shows that the search for the possible matches from variables of

C to variables of D may be restricted to those matches that respect the equivalence $\approx$. Results 4.2.7 to 4.2.16 give sufficient conditions for the variant property for some particular cases.

For the remaining cases we use a constraint propagation procedure: First we determine for each $L \in C$ the set $[L]_{C,D}$ of possible matches in D. Then the following constraint is propagated through C: if two literals L, K share a variable x, say at positions i and j, then the matches L' for L and K' for K must have a common variable at positions i and j. This yields the formal definition of what we call incompatibility of two matches for two different literals:

Let $L=Px_1\ldots x_n$, $K=Qy_1\ldots y_m$ be literals of C and let $L'=Px_1'\ldots x_n'$, $K'=Qy_1'\ldots y_m'$ be possible matches for L, K, respectively. Then we call L' and K' **incompatible**, if the mapping $\{L\rightarrow L', K\rightarrow K'\}$ does not induce a renaming of the corresponding variables, that is, if $x_i=y_j$ for some $i\in\{1,\ldots,n\}$, $j\in\{1,\ldots,m\}$, but $x_i'\neq y_j'$.

Algorithm

VARIANT (C,D).

INPUT: two clauses C and D, with variable sets V and W, respectively.

OUTPUT: TRUE, if $C\cong D$, FALSE otherwise.

0. If $|C|\neq|D|$, $|V|\neq|W|$, $\{\chi(x) \mid x\in V\}\neq\{\chi(x') \mid x'\in W\}$ or $\{\chi(L) \mid L\in C\}\neq\{\chi(K) \mid K\in D\}$, then return FALSE.

1. If for each variable $x\in V$ that occurs more than once, $[x]_C$ is a singleton, then return TRUE.

If C is homogeneous, then

if $|V|\leq 3$, $|C|<3$, or ($|C|=3$ and $[x]_C<3$ for all $x\in V$), then return TRUE.

2. Otherwise form a queue consisting of all literals of C. For each literal L of C compute the set of all neighbourhood literals, i.e. those literals that have at least one variable in common with L.

3. Until the queue is empty:

3.1 Remove the first literal from the queue; this is now the current literal.

3.1.1 Compute, if this has not yet been done, the set of all possible matches for the current literal; i.e. the set of all literals of D with the same characteristic as the current literal.

3.1.2 Remove all literals X from the set of possible matches for the current literal that are incompatible with all possible matches for a literal Y in the neighbourhood of the current literal. If thereby the set of possible matches for the current literal becomes empty, then return False.

3.2 If any change has occurred, then add those literals that are in the neighbourhood of the current literal to the front of the queue.

4. Return True.

To show the termination of the algorithm, let $\Pi = \sum_{L \in C} |[L]_{C,D}|$, where $[L]_{C,D}$ denotes the set of possible matches for L. Let Q denote the queue used in the algorithm. It is easy to see, that the pair $(\Pi, |Q|)$ decreases with respect to the lexicographic order over the natural numbers each time the loop is traversed.

4.2.19 Example:

Let $C = \{K_1, K_2, K_3, K_4\}$ and $D = \{L_1, L_2, L_3, L_4\}$ with

$L_1 = Ptw$, $L_2 = Pst$, $L_3 = Prs$, $L_4 = Pqr$ and

$K_1 = Pxy$, $K_2 = Pyz$, $K_3 = Pzu$, $K_4 = Puv$.

C and D are homogeneous, we thus omit the index P for the characteristics in the following. The following variable characteristics (see figure 4.3) are computed:

V	χ	W
x	(1, 0)	q
y, z, u	(1, 1)	r, s, t
v	(0, 1)	w

fig. 4.3

and the following literal characteristics (see figure 4.4)

C	χ	D
K_1	$\begin{bmatrix}1&1\\0&1\end{bmatrix}$	L_4
K_2, K_3	$\begin{bmatrix}1&1\\1&1\end{bmatrix}$	L_2, L_3
K_4	$\begin{bmatrix}1&0\\1&1\end{bmatrix}$	L_1

fig. 4.4

Thus we have $\{\chi(x) \mid x \in V\} = \{\chi(x') \mid x' \in W\}$ and $\{\chi(L) \mid L \in C\} = \{\chi(K) \mid K \in D\}$. Since $|V|>3$, $|C|>3$, and $|[y]|>1$, none of the sufficient conditions for returning true applies.

We form the queue $Q = (K_1, K_2, K_3, K_4)$ and remove its first literal K_1. The only possible match for K_1 is L_4, that is, $PM(K_1) = \{L_4\}$. All literals in the neighbourhood of K_1 are still in the queue, so we remove the next literal, K_2, from Q. Next we compute $PM(K_2) = \{L_2, L_3\}$.

The only literal in the neighbourhood of K_2, for which the set of possible matches is yet computed, is K_1. The possible match L_2 for K_2 is incompatible with L_4, since it maps y to s. So L_2 is canceled from $PM(K_2)$.

Now we have $PM(K_2) = \{L_3\}$ and $Q = (K_3, K_4)$. Since a change has occurred, we have to put the literal K_1, which is in the neighbourhood to K_2, at the front of the queue.

Next K_1 is removed from the queue and it is easily seen that no change will occur in this step.

The next literal, K_3, is removed from the queue. We compute the set of possible matches for K_3, which is $PM(K_3) = \{L_2, L_3\}$.

The possible match L_3 for K_3 is incompatible with the match L_3 for the neighbourhood literal K_2. Therefore L_3 is removed from $PM(K_3)$. Thus we have $PM(K_3) = \{L_2\}$, and $Q = (K_4)$. Now K_2 has to be added again to the front of the queue, but this will not yield a change.

Now the last element, K_4, is removed from the queue. L_1 is the only possible match for K_4, and this match is compatible with all other matches.

Now the queue is empty; therefore $C \cong D$ holds.

4.3 An Algorithm to Produce the Irreducible Factor of a Clause

This section provides an algorithm that computes the irreducible factor of a given clause. A first approach is suggested by the definition of a subsuming factor. Such an algorithm, which is proposed by Joiner (1973)[1], proceeds by checking the given clause C for unifiable pairs of literals. If such a pair (L,K) is found, and the most general unifier σ of L and K has the property that $C\sigma$ subsumes C, then $C\sigma$ is a subsuming factor of C, and the procedure is repeated with $C\sigma$ instead of C. This is done until no more pair of unifiable literals can be found in the actual clause. This approach's drawback consists in the unification procedure it involves, and, in particular, the number of expensive subsumption tests. It will turn out, however, that the unification operation is not really necessary, and, moreover, that the number of subsumption tests can be significantly reduced.

By lemma 2.2.5, at least one of the irreducible factors of a clause is a subset of this clause. Our algorithm produces such an irreducible factor. Thus C^* will always denote one of the irreducible factors of C that satisfies $C^* \subseteq C$.

Our introductory example 4.1.2 contained the reducible clause $C_1 = \{x \leq y, y \leq x, z \leq y\}$. Here, the fact that the subclause $C' = \{x \leq y, y \leq x\}$ of C is subsumed by $C \setminus C' = \{z \leq y\}$, with the additional property that C' is invariant under the subsumption substitution, accounts for the reducibility of C. The following lemma, which is fundamental for our algorithm, shows that this indeed constitutes a necessary and sufficient condition for reducibility.

4.3.1 Lemma:

The clause C has a subsuming factor C', iff there exists a substitution μ with

(i) $(C \setminus C')\mu \subseteq C'$ and

(ii) $\mathbb{V}(C') \cap \mathrm{dom}(\mu) = \emptyset$

Proof: Suppose that conditions (i) and (ii) are satisfied. Then, by (ii), $C'\mu = C'$ holds, hence from (i) follows $C\mu = C'\mu \cup (C \setminus C')\mu = C'$, which implies that C' is a subsuming factor of C.

---

[1] Joiner calls a subsuming factor of clause C a condensation of C, and the irreducible factor of C the most specific condensation of C.

Conversely, let C be reducible and let C' be a subsuming factor of C. By lemma 2.2.5, we can assume w.l.o.g. that $C' \subseteq C$. Hence, there exists an idempotent substitution μ with $C' = C\mu \subseteq C$. Then we have $(C \setminus C')\mu \subseteq C\mu = C'$.
Finally, if $x \in \mathbb{V}(C')$, then $x = y\mu$ for some $y \in \mathbb{V}(C)$. This implies $x\mu = y\mu^2 = y\mu = x$, that is, $x \notin dom(\mu)$. ■

4.3.2 Definition:

Let C be a clause and $C', C'' \subseteq C$. We write

$$C = C' \oplus C''$$

if $C = C' \cup C''$ and $\mathbb{V}(C') \cap \mathbb{V}(C'') = \emptyset$. C is called **connected**, if it cannot be written in the form $C' \oplus C''$.

4.3.3 Lemma:

If $C = C_1 \oplus C_2$, then C_2 is a subsuming factor of C iff it is subsumed by C_1.

Proof: Observe that $\mathbb{V}(C_1) \cap \mathbb{V}(C_2) = \emptyset$ enforces condition (ii) of lemma 4.3.1, hence $C_1 \leq C_2$ implies that C_2 is a subsuming factor of C. To prove the converse direction, assume that C_2 is a subsuming factor of C, that is, $C_2 = C\mu \subseteq C$ for some idempotent substitution μ. If $x \in \mathbb{V}(C_2)$, then $x = y\mu$ for some $y \in \mathbb{V}(C)$. Hence $x\mu = x$, and we have

$$C_2 = C\mu = C_1\mu \cup C_2\mu = C_1\mu \cup C_2.$$

Hence $C_1\mu \subseteq C_2$ holds. ■

4.3.4 Corollary:

If $C = C_1 \oplus \ldots \oplus C_n$ such that C_i is irreducible and connected for $i \in \{1,\ldots,n\}$, then

$$C^* = M_1 \oplus \ldots \oplus M_k,$$

where $\{M_1,\ldots,M_k\}$ is the set of maximal elements in $\{C_1,\ldots,C_n\}$ w.r.t. the subsumption ordering.

Proof: Follows from the previous lemma. ■

The previous lemmata suggest a proceeding for computing the irreducible factor of a given clause C as follows: First find the connected subsets $\{C_1,\ldots,C_n\}$ of C and reduce each of it to its irreducible factor. Then test each pair of these reduced clauses on subsumption and remove the subsuming clause, if the test was successful. If C is a connected clause, the irreducible factor of C is produced as follows: Test for all subsets C' of C, whether (a) C'

subsumes C\C' and (b) the subsumption substitution satisfies condition (ii) of lemma 4.3.1.

The condition $x\mu=x$ for each $x\in \mathbb{V}(C')\cap\mathbb{V}(C\backslash C')$ provides a restriction for the number of possible matching substitutions μ. This condition can also be expressed as follows: If x is a variable, which is shared by C' and C\C', and L is a literal in C', such that $x\in L$, then the search for the matching literal L' in C\C' can be restricted to those L' that satisfy $x\in L'$. Moreover, we can state that the occurrences[1] of x in L must be the same as those of x in L'. This allows a characterization of subclauses as potential subsumers:

4.3.5 Definition:

Let C be a connected clause, and let $C'\subseteq C$. A variable $x\in \mathbb{V}(C')\cap\mathbb{V}(C\backslash C')$ is called **isolated in C'**, if there is $L\in C'$ with $x\in \mathbb{V}(L)$ and $x\in dom(\sigma)$ holds for each $\sigma\in uni(L,C\backslash C')$.

4.3.6 Example:

Let C = {Pxy, Pzx, Pwy}, and let L=Pxy. Then x is isolated in C' = {L}, since $x\sigma=z$ or $x\sigma=w$ holds for each $\sigma\in uni(L,C\backslash C')$. A similar argument shows that x is also isolated in {Pzx}. Note, however, that x is not isolated in C"={Pxy, Pzx}, since the condition $x\in \mathbb{V}(C'')\cap\mathbb{V}(C\backslash C'')$ is violated. This restricts the potential subsuming factors to C" and C\C".

4.3.7 Lemma:

Let C be a connected clause, let $C'\subseteq C$. If C' is a subsuming factor of C, then C\C' contains no isolated variables.

Proof: Since C' is a subsuming factor of C, there exists a substitution μ with $(C\backslash C')\mu\subseteq C'$, and $\mathbb{V}(C')\cap dom(\mu)=\emptyset$. In particular, $\mu\in uni(L,C')$ holds for each $L\in C\backslash C'$, and if $x\in \mathbb{V}(C')\cap\mathbb{V}(C\backslash C')$, then $x\notin dom(\mu)$ holds, which implies that x is not isolated in C\C'. ■

---

[1] We do not formally define the notion of an occurrence of a variable x in a term t. But it is intuitively clear that this means the *tree address* of x in t. We shall rather denote the fact that x has the same occurrences in literals L and K by $x\notin dom(\sigma)$ for the most general unifier σ of L and K.

According to the previous lemma, the subsets C' may be chosen from those subsets of C that contain no isolated variables.

In particular, if a variable x occurs exactly in the literals L and K, but with different occurrences, then x is L-isolated and also K-isolated. This implies that L is element of a potential subsumer if and only if K is. The variable x will be called (L,K) isolated in this case. For instance, the variable x in example 4.3.6 is (L,K)-isolated with L=Pxy, K=Pzx. The test on isolated variables is mainly understood to be a test on irreducibility of the underlying clause. As in actual problems the clauses most probably are not reducible, a test which early detects failure, should be more valuable than a test which assumes reducibility of the clause. We suggest that for most clauses C arising in practice the reducibility test will fail exactly on account of (L,K)-isolated variables for each pair (L,K) of literals of C.

When checking whether a subclause C' of a connected clause C is a subsuming factor of C, the shared variables are instantiated with "new" constants in order to assure that the subsumption substitution μ with $(C\setminus C')\mu \subseteq C'$ leaves C' invariant.

Algorithm

IRRED_FACTOR (C)
Input: A clause C
Output: The irreducible factor C^* of C

1. Compute all the connected components $C_1,\ldots,C_n$ of C, and let $S=\{C_1,\ldots,C_n\}$.
2. For $i=1,\ldots,n$ do
   $C_i := \text{IRRED\_CONN}\ (C_i)$.
3. For $i=1,\ldots,n$ do
   If C_i subsumes some $C_j \in S\setminus\{C_i\}$, then $S := S\setminus\{C_i\}$.
4. $C^* := \bigcup_{C_i \in S} C_i$.

Function

IRRED_CONN (C)
Input: A connected clause C
Output: The irreducible factor C^* of C

1. Let S = POTENTIAL_SUBSUMERS(C).
2. If $S \neq C$ then for all subsets C' of C containing some element of S do

Let $V = \mathbb{V}(C') \cap \mathbb{V}(C \setminus C')$.

Let $\mu = \{x \to a_x \mid x \in V;\ a_x$ is a constant not occurring in $C\}$.

If $C'\mu$ subsumes $(C \setminus C')\mu$, then return IRRED_CONN$(C \setminus C')$.

3. Return C.

Function

POTENTIAL_SUBSUMERS (C)

Input: A connected clause C

Output: $\{C' \subseteq C \mid C'$ contains no isolated variable$\}$.

1. Let $C = C_1 \cup \ldots \cup C_n$, where (L_i, L_j) with $L_i \in C_i$, $L_j \in C_j$ is unifiable iff i=j.
2. m=1. $S_0 = \emptyset$. For all $x \in \mathbb{V}(C)$ do

   If x is (L,K)-isolated for some literals L,K, then

   if there are $i \leq m$ such that $L \in S_i$ and $j \leq m$ such that $K \in S_j$, then $S_j = S_i \cup S_j$; $S_j = \emptyset$ else

   if there is $i \leq m$ such that $L \in S_i$ then $S_i = S_i \cup \{K\}$ else

   if there is $j \leq m$ such that $K \in S_j$ then $S_j = S_j \cup \{L\}$ else

   $S_m = \{L,K\}$; m=m+1.

   If either of S_i, S_j, or S_m contains some C_k, then return C.
3. Return $\{S_1, \ldots, S_m\}$.

Next, we give some examples to demonstrate how the algorithm works. These are clauses, which frequently occur in axiomatizations of mathematical structures that avoid the use of equality. All these clauses are connected clauses. In all examples the test on isolated variables suffices to recognize irreducibility.

4.3.8 Examples:

a) This is the clause expressing n-fold transitivity of the P-predicate. Such a clause can be obtained, for instance, by n-fold self-resolution of the well-known transitivity clause $Pxy \wedge Pyz \Rightarrow Pxz$. Let n be a natural number, $n \geq 2$, and let $C_n = \{L_1, \ldots, L_n, L_{n+1}\}$ with

$L_1 = \neg Px_1x_2, \ldots, L_n = \neg Px_nx_{n+1}$, $L_{n+1} = Px_1x_{n+1}$.

For each $i \in \{2, \ldots, n\}$, the variable x_i is (L_{i-1}, L_i)-isolated. Thus $C \setminus \{L_{n+1}\}$ is the smallest subset of C containing no isolated variables. Obviously, this subclause can not subsume $\{L_{n+1}\}$, hence C_n is irreducible. For Joiner's algorithm, however, this is nearly the worst case example. Each pair (L_i, L_j)

with $i,j \in \{1,\ldots,n\}$ is unifiable. Thus the algorithm requires $O(n^2)$ subsumption tests with a very unfavourable subsumer/subsumend relation of $n/n+1$[1].
b) The following two clauses represent the axiom of associativity in a notation that avoids the use of equality.[2]

$C_1 = \neg Pxyu\ \neg Pyzv\ \neg Pxvw\ Puzw$

$C_2 = \neg Pxyu\ \neg Pyzv\ \neg Puzw\ Pxvw$

The test on isolated variables yields the following result: The variable y is isolated in the first two literals, v is isolated in $\neg Pyzv$ and $\neg Pxvw$ ($Pxvw$), and u is isolated in $\neg Pxyu$ and $Puzw$ ($\neg Puzw$). Thus each proper subset of C_1 and C_2, respectively, contains isolated variables.

4.4 A Subsumption Algorithm Based on Characteristic Matrices

In this section it is shown that the variant test of section 4.2 can be generalized to a subsumption algorithm that compares favorably with the subsumption tests developed as yet. First, we shall give a brief overview on those algorithms[3]. All these algorithms rely heavily on the fact that the problem to decide whether clause C subsumes clause D is equivalent to the decision whether C subsumes D_{gr}, where D_{gr} is obtained from D by replacing all variables by constants not occurring in $C \cup D$. We shall thus assume throughout this section that D is a ground clause. We shall refer to C as the *subsumer*, and D will be called the *subsumend*.

The search for a matching substitution μ, such that $C\mu \subseteq D$, amounts to a successive computation of matchers for literals pairs (L,K), where $L \in C$, $K \in D$. In analogy to Robinson's (1965a) unification algorithm on the one hand, and the algorithm given by Martelli & Montanari (1982) on the other hand, there are two different ways to find out, whether the computed substitutions are

---

[1] Such a test has a worst case complexity of $O((n+1)^n)$, see for instance Gottlob & Leitsch (1985).

[2] Axiomatizations of some mathematical theories including group theory and ring theory can be found in Wos's (1988) *Basic Research Problems*.

[3] Gottlob & Leitsch (1985) provide a detailed discussion of the existing subsumption tests (except Eisinger's (1981)) and their complexity.

compatible. The first approach, which could be compared with Robinson unification, instantiates the remainder of the given clauses with the substitution just found. This approach is followed by Chang & Lee (1973), by Stillman (1973), and by Gottlob & Leitsch (1985). The first two differ in the search strategy, while the third algorithm improves Stillman's by exploiting the partition of the clause C into connected components, similarly to the algorithm given in section 4.3 of this thesis.

The second approach, followed by Eisinger (1981), first computes for each literal of C a matcher into D and then tests these matchers on compatibility. This subsumption test is mainly designed for use in Kowalski's (1975) connection graph procedure[1], it is yet also applicable for resolution based systems. In the connection graph environment, this subsumption test profits by the explicit representation of unifiers between the literals of C and D by so called S-links[2].

Experience has shown that it is much more likely that subsumption does not take place than the converse. Therefore it seems reasonable to look for some easy to test criteria that preclude subsumption. One such criterion is easily provided by the S-link test. If the subsumer contains a literal, which is not connected to any literal in the subsumend, then subsumption cannot take place.

As our algorithm is based on Eisinger's, we shall provide a short description of the S-link test. The following theorem can be found in Eisinger's (1981) paper.

4.4.1 Theorem:

Let $C = \{L_1,...,L_n\}$ and D be clauses. Then C subsumes D iff $|C| \leq |D|$ and there is an n-tuple $(\sigma_1,...,\sigma_n) \in uni(C,L_1,D_{gr}) \times ... \times uni(C,L_n,D_{gr})$ such that the σ_i are pairwise strongly compatible[3]. ■

---

[1] For a detailed account of graph based reasoning see Eisinger's thesis (1988).

[2] The algorithm is thus also called the *S-link test*.

[3] In our particular case, where D is a ground clause, the notions of *unifiers* and *matchers* coincide, and so do the notions of *compatibility* and *strong* compatibility (see lemma 2.1.10).

4.4.2 Example:

Given the set $\{C,D_1,D_2,D_3\}$ of clauses with

$C=\{Pxy,Qyc\}$, $D_1=\{Pac,Rbc\}$, $D_2=\{Puv,Qvw\}$, $D_3=\{Pab,Pba,Qac\}$

one wants to find out, which clauses are subsumed by C. In fig. 4.5 the unifiable literals are connected with links.

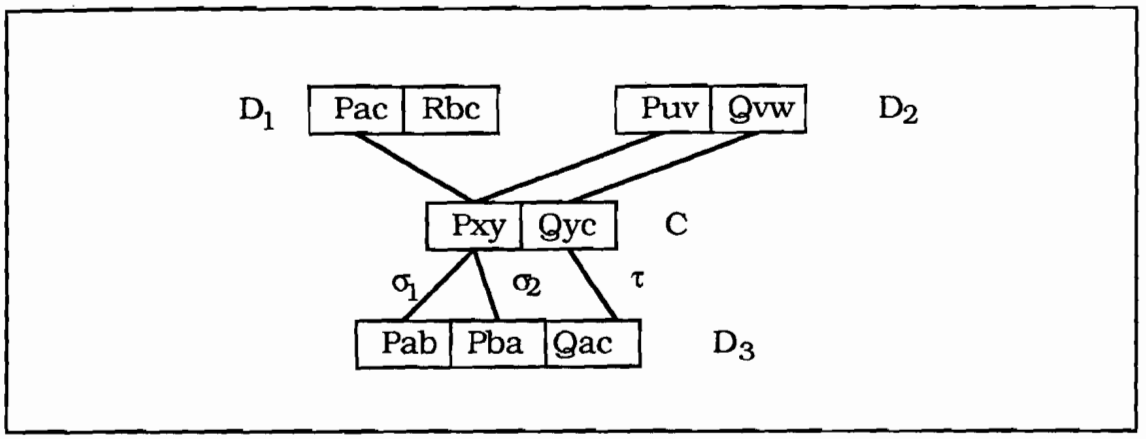


fig. 4.5

D_1 can be excluded, since the literal Qyz from C is not unifiable with any literal in D_1. D_2 cannot be a candidate either, since $uni(C,Qyc,(D_2)_{gr})=\emptyset$. For D_3 we obtain the two pairs (σ_1,τ) and (σ_2,τ), where

$\sigma_1=\{x\rightarrow a, y\rightarrow b\}$, $\sigma_2=\{x\rightarrow b, y\rightarrow a\}$ and $\tau=\{y\rightarrow a\}$.

From these two pairs only (σ_2,τ) is strongly compatible and thus C subsumes D_3.

This example shows that in order to find the clauses that are subsumed by a given clause $C=\{L_1,...,L_n\}$ first there is a preselection of those clauses that are connected to every literal in C by the S-links of the connection graph. For such a candidate clause D the subsumption algorithm is accomplished by a test of all elements of $uni(C,L_1,D)\times...\times uni(C,L_n,D)$ on strong compatibility.

Subsumption tests involving long clauses with more than one matching substitution for each literal may require an expensive search of all elements of the cartesian product. A case in point is the existence of more than one most general unifier for two literals on account of theory unification. In this section we shall provide an algorithm that basically employs the same principles to restrict the search for an appropriate mapping as the variant test of section 4.2. Subsumption can be considered a generalization of graph homomorphism, as is the case for renaming and graph isomorphism. It goes

without saying that the properties invariant under homomorphism are weaker than those invariant under isomorphism. For our purposes, the mere existence of an outgoing (incoming) link for some node will be an appropriate invariant. The extensions of the graph algorithms in order to include constants and functions, however, will turn out to be more complicated than in section 4.2. We thus first deal with the case where the subsumer is a function free clause. Constants and functions in the subsumend do not provide any problems:

4.4.3 Definition:

Let D be a ground clause, and let $T = \mathbb{T}(D)$ be the set of all terms occurring as arguments in literals of D. Let $\theta : T \to \mathbb{V}$ be injective. Then for each predicate occurring in D, the characteristic function $\chi_{P,D}$ is defined by $\chi_{P,D} = \chi_{P,D\theta}$.

4.4.4 Example:

Let $C = \{L_1, L_2\}$ with $L_1 = P(fa,gb)$ and $L_2 = P(gb,a)$. Then the terms occurring as arguments are $t_1 = fa$, $t_2 = gb$, $t_3 = a$. With $\theta = \{t_i \to x_i \mid i=1,2,3\}$ we obtain $C\theta = \{Px_1x_2, Px_2x_3\}$, and thus $\chi_C(L_1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$.

It is obvious that the definition of the characteristic function of D does not depend on the particular choice of θ.

4.4.5 Lemma:

Let C be function free, and let D be a ground clause. Let $\theta : \mathbb{T}(D) \to \mathbb{V} \setminus \mathbb{V}(C)$ be injective. Then C subsumes D, iff C subsumes $D\theta$.

Proof: If $C\sigma \subseteq D$, then $C\sigma\theta \subseteq D\theta$, hence C subsumes $D\theta$. If conversely $C\tau \subseteq D\theta$, then $C\tau\theta^{-1} \subseteq D$, hence C subsumes D. ■

Characteristic matrices encode variable occurrences and the test proposed in section 4.2, whether the renaming σ satisfies $C\sigma = D$ is based on the observation that there must be pairs of variables from the two clauses that match in the number of occurrences, which is expressed by $\chi_C(x) = \chi_D(x\sigma)$ for all $x \in \mathbb{V}(C)$. One might suspect that $\chi_C(x) \leq \chi_D(x\sigma)$[1] for all $x \in \mathbb{V}(C)$ is the

[1] The relation $\leq$ on n-tuples is defined pointwise, that is, $(a_1,\ldots,a_n) \leq (b_1,\ldots,b_n)$, iff $a_i \leq b_i$ for all $i \in \{1,\ldots,n\}$.

appropriate condition for a subsumption substitution σ. That this is not the case, is shown by the clauses $C = \{Pxy, Pxz, Pyz\}$ and $D = \{Puu\}$. The substitution $\sigma = \{v \to u \mid v \in \mathbb{V}(C)\}$ satisfies $C\sigma \subseteq D$, but $\chi_C(x) = (2,0) \not\leq (1,1) = \chi_D(x\sigma)$. Only the following weaker condition is necessary for a subsumption substitution σ: If the variable x occurs at least once at argument position k, that is, if $(\chi_C(x))_k \geq 1$, then $x\sigma$ must occur at the same argument position, that is, $(\chi_D(x\sigma))_k \geq 1$. This can also be expressed by $\text{sign}(\chi_C(x))_k \leq \text{sign}(\chi_D(x\sigma))_k$. In the following we shall denote sign(n) by $\bar{n}$.

4.4.6 Definition:

a) We define a relation $\lesssim$ on $\mathbb{N}\times\mathbb{N}$ by $n \lesssim m$, iff $\bar{n} \leq \bar{m}$. This relation is extended to n-tuples and matrices over $\mathbb{N}$ in an obvious way.

b) We define a relation $\lesssim$ on clauses C, D by
$L \lesssim K$ iff (i) $\mathbb{P}(L) = \mathbb{P}(K)$ and (ii) $\chi_{P,C}(L) \lesssim \chi_{P,D}(K)$ holds for each predicate symbol P occurring in $C \cup D$.

4.4.7 Lemma:

Let C be a function free clause and let D be a ground clause. If there is a substitution σ, such that $C\sigma \subseteq D$, then $L \lesssim L\sigma$ holds for each $L \in C$.

Proof: Suppose $C\sigma \subseteq D$ and $L \in C$. Let $A = \bar{\chi}_{P,C}(L)$ and $B = \bar{\chi}_{P,D}(L\sigma)$. Let $L = Px_1 \ldots x_n$, and take any $i,j \in \{1,\ldots,n\}$. If $A_{i,j} = 0$, then obviously $A_{i,j} \leq B_{i,j}$. If $A_{ij}=1$, then there is some $K \in C$ with $K = Py_1 \ldots y_n$ such that $x_i = y_j$. Since $K\sigma \in D$, we have $x_i\sigma = y_j\sigma$, hence $B_{i,j}=1$. ∎

The previous lemma provides a restriction on the possible matching substitutions that suffices in many cases to exclude subsumption alltogether: If there is some literal L in C such that there exists no literal K in D with $L \lesssim K$, then C cannot subsume D. The following example, taken from (Gottlob & Leitsch 1985), illustrates this enhancement of the subsumption test.

4.4.8 Example:

For any $m \in \mathbb{N}$, let

$C_m = \{Pxy_1z_1, Pz_1y_2z_2, \ldots, Pz_{m-2}y_{m-1}z_{m-1}, Pz_{m-1}xz_m\}$ and

$D_k = \{Pab_1a, \ldots, Pab_ka\}$.

Each pair $(L_i,K_j) \in C_m \times D_k$ is unifiable. Let σ_{ij} be the unifier of (L_i,K_j).

a) The S-link test for subsumption needs k^m steps in the worst case and k^2 steps in the best case to detect that C does not subsume D:

For each literal $L_i \in C_m$ we have $uni(L_i,D) = \{\sigma_{ij} \mid 1 \le j \le k\}$ whence $|uni(L_i,D)| = k$. Therefore

$$\left| \bigtimes_{i=1}^{m} uni(L_i,D) \right| = k^m.$$

The number of steps needed for the search of a strongly consistent m-tuple $(\sigma_1,..,\sigma_m) \in \bigtimes_{i=1}^{m} uni(L_i,D)$ depends on the ordering of the literals in C_m and on the search strategy. Depth-first search (with uncontrolled backtracking) will always yield the worst case with complexity k^m. Breadth-first search will yield the best case when one starts with the literals L_1 and L_m and the worst case when one ends with one of the literals L_1 or L_m.

b) The test provided by lemma 4.4.7 needs at most 2 steps to establish non-subsumption.

The characteristic matrices of the literals in C are computed as follows:

$$\chi_P(L_1) = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \chi_P(L_i) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \text{ for } i \in \{2,\ldots,m-1\}, \text{ and } \chi_P(L_m) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix},$$

and for the clause D

$$\chi_P(K) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix},$$

holds for each $K \in D$. We have to test if $\chi(L_i) \lesssim \chi(K)$ holds for the three matrices above. After at most two steps one can recognize that $\chi(L_1) \lesssim \chi(K)$ (respectively $\chi(L_m) \lesssim \chi(K)$) does not hold.

In order to extend the characteristic to arbitrary literals, we define the function free form C^* of a clause C as it was defined in section 4.2.

The computation of the characteristic of the clause D requires only the consideration of those literals in D that are instantiations of some literal in C. We write those literals with the predicate symbols occurring in C^*.

4.4.9 Definition:

Let C and D be clauses. Then

$$M(D,C) := \{L^*\mu \mid L \in C, \mu \in uni(C,L,D)\}.$$

4.4.10 Example:

Let $C = \{L_1, L_2, L_3\}$ with

$L_1 = P(x,y)$, $L_2 = P(f(y),d)$, $L_3 = R(a,g(x))$ and

$D = \{P(a,b), P(f(b),d), R(a,g(b))\}$.

For the construction of C^* we introduce the new predicate symbols Q_1, Q_2, Q_3. Then $C^*=\{Q_1xy, Q_2y, Q_3x\}$. We have the following matchers:

$uni(C,L_1,D) = \{\mu_{11}, \mu_{12}\}$, with $\mu_{11}=\{x\to a, y\to b\}$, $\mu_{12}=\{x\to f(b), y\to d\}$.

$uni(C,L_2,D) = \{\mu_{22}\}$, with $\mu_{22}=\{y\to b\}$.

$uni(C,L_3,D) = \{\mu_{33}\}$, with $\mu_{33}=\{x\to b\}$.

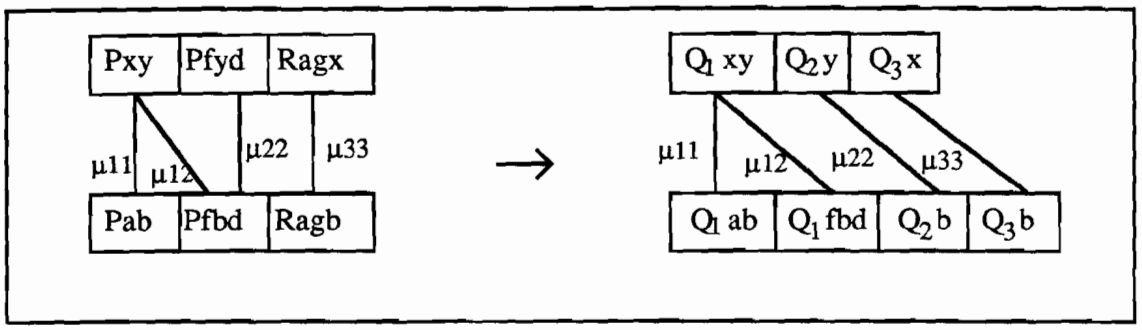


fig. 4.6

This yields

$M(D,C) = \{Q_1ab, Q_1fbd, Q_2b, Q_3b\}$

4.4.11 Lemma:

Let C and D be clauses. C subsumes D iff C^* subsumes M(D,C).

Proof: Assume there is a substitution σ with $C\sigma\subseteq D$. Let $K\in C^*$. Then there is an $L\in C$ with $L^*=K$. We have $K\sigma=L^*\sigma\in M(D,C)$, since $\sigma\in uni(C,L,D)$. Hence $C^*\sigma\subseteq M(D,C)$.

Now suppose there is a substitution σ with $C^*\sigma\subseteq M(D,C)$. This implies that for each $L\in C$ there is some K^* with $K\in C$ and some $\tau\in uni(C,K,D)$ with $L^*\sigma=K^*\tau$. This obviously implies $L\sigma=K\tau$, which proves $C\sigma\subseteq D$. ■

4.4.12 Corollary:

Let C and D be arbitrary clauses. If $C\sigma\subseteq D$, then

$$\chi_{P,C^*}(L) \le \chi_{P,M(D,C)}(L\sigma)$$

holds for each predicate symbol P in C, and each $L\in C^*$.

Now we can formulate the algorithm that improves the S-link test for C subsumes D:

Algorithm

SUBSUMPTION (C,D)

Input: Clauses $C = \{L_1,...,L_n\}$ and D.

Output: True, if C subsumes D and False otherwise.

1. If there is a literal L in C that does not possess an S-link to a literal in D, then return False.
2. For each literal L of C do begin
   Compute uni(C,L,D) . If uni(C,L,D)=∅, then return False end.
3. Compute C^* and M(D,C) and the set of all characteristic matrices of all literals.
4. For all $L \in C$ do begin
   for all $\sigma \in$ uni(C,L,D): If $\chi(L^*) \not\leq \chi(L\sigma)$, then discard σ from uni(C,L,D);
   if uni(C,L,D) = ∅, return False end
5. If there is some $(\sigma_1,...,\sigma_n) \in \bigtimes_{i=1}^{m}$ uni(L_i,D) such that the σ_i are pairwise strongly compatible then return True, otherwise False.

In automated theorem proving subsumption tests are usually repeated very often. Therefore the characteristic matrices for a clause C can be computed once for many subsumption tests.

Of course one has to realize that the computation of the set S of characteristic matrices for the clause C is an additional effort that has to be performed by the subsumption algorithm. Thus one has to weigh the costs of the method against its possible gains.

Finally, it should be pointed out that the occurrence of theory unifiers causes no change in the subsumption algorithm. Theory subsumption in an equational theory $\mathcal{E}$ ($\mathcal{E}$-subsumption) is defined by the existence of a substitution σ, such that $C\sigma =_{\mathcal{E}} D' \subseteq D$. Lemma 4.4.11 applies to $\mathcal{E}$-subsumption in the following form: C $\mathcal{E}$-subsumes D, iff C^* subsumes

$M_{\mathcal{E}}(D,C) = \{L^*\mu \mid L \in C, \mu \in \text{uni}_{\mathcal{E}}(C,L,D)\}$,

where $\text{uni}_{\mathcal{E}}(C,L,D)$ denotes the set of $\mathcal{E}$-unifiers of L and literals in D. Thus E-subsumption is reduced to ordinary subsumption.

4.4.13 Example:

Let C = {Pf(x,y), Qxz, Qzx}, and D = {Pf(a,b), Qba}, where the function symbol f is commutative. We introduce the new predicate symbol P_1 for Pf(*,*), and obtain $C^* = \{P_1xy, Qxz, Qzx\}$. Since f is commutative,

$uni_{\mathcal{E}}(C,Pf(x,y),D) = \{\sigma_1,\sigma_2\}$ with

$\sigma_1=\{x\rightarrow a, y\rightarrow b\}$ and $\sigma_2=\{x\rightarrow b, y\rightarrow a\}$.

Thus $M_{\mathcal{E}}(D,C) = \{P_1ab, P_1ba, Qba\}$, and it remains to test whether $\{P_1xy, Qxz, Qzx\}$ subsumes $\{P_1ab, P_1ba, Qba\}$, which is performed with the algorithm above.

4.5 Concluding Remarks

Section 4 provides several techniques to detect redundant information, which come under subsumption. Both the variant test and the subsumption test are based on the notion of a characteristic matrix of a literal, which, roughly speaking, encodes the occurrences of the literal's variables in the literals of the whole clause. These tests proceed from the assumption that most probably a given clause does not subsume another one (and, a fortiori, it is not a variant of the other), and that failure of the subsumption test often finds is expression by clashes in variable occurrences. Thus our techniques proceed in the spirit of Eisinger's (1981) S-link test, by "filtering out" the potential subsumers according to fast and successively stronger preselections and thus postponing the expensive merging of substitutions. The particular siginificance of the variant test is due to the fact that for many "simple" clauses, which nevertheless frequently occur in practice, there is an algorithmic solution which requires no merging at all. This solution simply amounts to a test on equality of two sets of matrices. In the case of "simple" clauses, the corresponding set of matrices yields a unique representation. Therefore the characteristic matrix of a clause can be used just as an indexing scheme, that is, a single data element suffices to represent two clauses, which are equal up to renaming.

5 Eliminating the Derivation of Redundant Clauses

Not only the mere presence, but also the additional derivation of redundant information is one of the greatest obstacles to the efficiency of reasoning programs, as redundant clauses generate further redundancy. Wos (1988) reports an attempt to prove SAM's lemma (see Guard 1969) using hyper-resolution, where 6000 clauses identical to retained clauses and 5000 clauses being proper instances of retained clauses were generated. Even if these redundant clauses can be removed after their generation, they must be processed with demodulation, subsumption, and other standard procedures. Moreover, the test on subsumption is rather expensive (cf section 1.2 of this thesis). A strategy to prevent the generation of redundant clauses, or at least to reduce the number of newly generated unneeded clauses, would thus prove very useful for increasing the power of a reasoning system. In sections 5.3 and 5.4 we shall characterize two clause structures that admit the systematic derivation of subsumed clauses, and in section 5.5 we show how to cope with these structures in order to prohibit the derivation of subsumed clauses.

Another question addressed in this chapter is closely related to the quest for a strategy to decrease the derivation of redundant clauses. If we had a reasonable means to completely prohibit the derivation of redundant clauses, we could also decrease the number of necessary subsumption tests in a significant way, since use of this means would automatically exclude the occurrence of forward subsumed clauses. As the subsumption test is rather expensive, it would be of great value to have a means to restrict the number of required tests. Even if we cannot expect to find such a strategy that overcomes the need for forward subsumption tests, the results of sections 5.2 and 5.3 will show that we can at least exclude a great part of the present clauses from being subsumers of the newly generated clause.

The structures, which are to a part responsible for the derivation of redundant clauses, will be described as particular *clause graph* structures. Section 5.1 serves as a short introduction into the clause graph terminology.

5.1 Clause Graphs

In the following we shall deal with finite graphs, whose nodes are labelled with literals and whose links are R-links (i.e. links joining resolvable literals) or S-links (i.e. links joining unifiable literals) labelled with substitutions.

5.1.1 Definition:

An (un)directed **clause graph** is a 5-tuple $G = (N,\Lambda,[N],£,\Sigma^*)$, where

(i) N is a finite set of *literal nodes*.

(ii) $\Lambda \subseteq N\times N$ is a (symmetric) relation. The elements (L,K) of Λ are called *links* and written in the form LK.

(iii) $[N] \subseteq 2^N$ is a partition of the literal nodes. The elements of [N] are called the *clause nodes* of G. The clause of a literal node L is denoted by [L].

(iv) $£:N\rightarrow\mathbb{L}$ is a mapping, which labels each literal node L with a literal denoted by $\mathbb{L}L$ such that $LK\in\Lambda$ implies that the atoms of $\mathbb{L}L$ and $\mathbb{L}K$ are weakly unifiable.

(v) $\Sigma^*:\Lambda\rightarrow 2^\Sigma$ is a mapping, which labels each link with a set of substitutions, such that $LK\in\Lambda$ implies $\Sigma^*(LK)$ is the set of most general unifiers of the atoms of $\mathbb{L}L$ and $\mathbb{L}K$.

We do not distinguish between literal and clause nodes on the one hand and literals and clauses on the other hand. We make the additional (purely technical) requirement that $\mathbb{V}(C)\cap\mathbb{V}(D)=\emptyset$ for different clause nodes C and D of G. The standard graph theory terminology applies to clause graphs: a link $LK\in\Lambda$ *joins* the literal nodes L and K; the link LK is *incident* with the literal nodes L and K and also with the clause nodes [L] and [K]; two links are *adjacent*, if they are incident with a common literal node.

The following definitions and results hold equally for undirected as for directed clause graphs. Note that according to definition 5.1.1, undirected graphs are considered a special form of directed ones with symmetric link relation.

5.1.2 Definition:

For a clause graph $G = (N,\Lambda,[N],£,\Sigma^*)$ we use the following notation:
$\mathbb{N}(G) = N$, $\mathbb{\Lambda}(G) = \Lambda$, $\mathbb{C}(G) = [N]$

$O(L) = (\{L\}\times N)\cap\Lambda$ is the set of links outgoing from the literal L,
$I(L) = (N\times\{L\})\cap\Lambda$ is the set of links incoming to the literal L,
$\mathbb{A}(L) = I(L)\cup O(L)$ is the set of links incident with L, and likewise for clause nodes.
A link LK is called an **R-link**, if the signs of L and K are different, and an **S-link** otherwise.

The notions O(L), I(L), and $\mathbb{A}(L)$ coincide for undirected graphs. Two links with compatible substitutions will also be called **compatible**. For any clause node C, E(C) denotes the literals of C which are not incident with any $\lambda\in\mathbb{A}(C)$. If $O(C)\cap I(D)\neq\emptyset$, then D is called a **successor** of C, and C is a **predecessor** of D.

5.1.3 Definition:

A clause graph G' is a subgraph of a clause graph G, if each link of G' is a link of G, and each clause node of G' is a clause node of G, and the labeling function of G' is the appropriate restriction of the labeling function of G.

Two frequently occurring types of subgraphs are obtained from a graph G by removing a subset of $\mathbb{A}(G)$, or by removing a clause node of G together with the links incident with this node, respectively:

5.1.4 Definition:

Let G be a clause graph, let $\Lambda\subseteq\mathbb{A}(G)$, and let $C\in\mathbb{C}(G)$.

a) The subgraph G^{Λ} of G is defined by
$\mathbb{C}(G^{\Lambda}) = \mathbb{C}(G)$, and $\mathbb{A}(G^{\Lambda}) = \mathbb{A}(G)\setminus\Lambda$

b) The subgraph G^{C} of G is defined by
$\mathbb{C}(G^{C}) = \mathbb{C}(G)\setminus\{C\}$, and $\mathbb{A}(G^{C}) = \mathbb{A}(G)\setminus\mathbb{A}(C)$

Let λ=LK be an R-link with $L\in C$ and $K\in D$, and let $\sigma\in\Sigma^*(\lambda)$. Then the *resolvent along* λ is the resolvent $R = (C\setminus\{L\})\sigma\cup(D\setminus\{K\})\sigma$ of C and D. Let $\lambda_1 = MM'$ be an R-link with $M\in C\setminus\{L\}$ and $M'\in E$. If λ_1 is compatible with λ, then $M\sigma$ and M' are also resolvable. The link $\lambda_1' = M\sigma M'$ between the literal $M\sigma$ in R and the literal M' is called the link **inherited** from λ_1 **qua** λ (see figure 5.1). For any link λ, we define the relation $\rightarrow_\lambda$ by $C\rightarrow_{D,\lambda} R$, iff $\lambda\in O(C)\cap I(D)$, and R is the resolvent along λ.

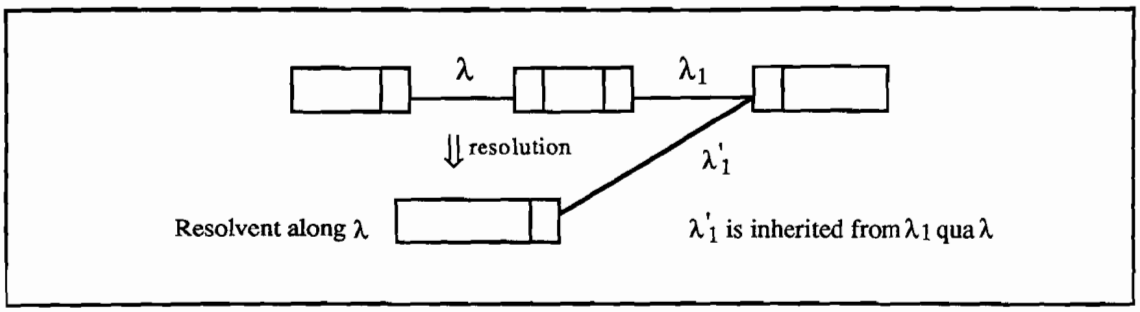


Fig. 5.1: Link Inheritance

If $\Lambda = \{\lambda_1,\ldots,\lambda_n\}$ is any finite set of links, then the merge of Λ is defined by $\Sigma^*(\Lambda) = \Sigma^*(\lambda_1)*\ldots*\Sigma^*(\lambda_n)$, that is the set of most general common instances of substitutions belonging to Λ. Note that in particular $\Sigma^*(\emptyset) = \{id\}$.

5.1.5 Definition:

Let G be a clause graph and let $\lambda \in \mathbb{L}(G)$. Then the derivation relation $G \rightarrow_\lambda G'$ is defined by $\mathbb{C}(G')=\mathbb{C}(G)\cup\{R\}$, where R is the resolvent along λ, and $\mathbb{L}(G')=\mathbb{L}(G)\cup\Lambda'$, where Λ' is the set of links, which are inherited qua λ.

A **path** from node C_1 to C_n, $n\geq 1$, in a clause graph is an alternating sequence $(C_1, \lambda_1, \ldots, C_{n-1}, \lambda_{n-1}, C_n)$ of clause nodes and links, such that $\lambda_i \in O(C_i)\cap I(C_{i+1})$ for $i \in \{1,\ldots,n-1\}$, no two links are adjacent, and the links $\lambda_1,\ldots,\lambda_{n-1}$ are pairwise compatible. The latter condition implies that $\Sigma^*(\{\lambda_1,\ldots,\lambda_{n-1}\}) \neq \emptyset$.

The path from C_1 to C_n is called **weakly cyclic**, if there is also a link λ_n from C_n to C_1. A weakly cyclic path is called **cyclic**, if $\Sigma^*(\{\lambda_1,\ldots,\lambda_{n-1},\lambda_n\}) \neq \emptyset$. A clause graph is called cyclic, if it contains a (weakly) cyclic path.

5.2 Redundancy Caused by Cyclic Structures

On closer inspection of the proof of SAM's lemma it turns out that many of the duplicates are generated by twofold application of the symmetry clause $S = \neg Pxy\ Pyx$, like[1]

$$Pab \rightarrow_S Pba \rightarrow_S Pab$$

[1] Remember the notation $C \rightarrow_D E$ for the resolution step yielding resolvent E (definition 2.4.7).

Obviously, only the symmetry clause Pxy ¬Pyx accounts for the derivation of the duplicate clause Pab in this example. No matter what the other clause looks like, the result after the second resolution step is always identical to the original clause. This should be compared with the following two resolution steps with the clause C = ¬Px Pa:

$Pa \rightarrow_C Pa$ and $Pb \rightarrow_C Pa$

In this example, the derivation of a subsumed clause depends not only on the clause C, but also on the choice of the other clause. In the following section, we shall be concerned with structures of the first type. In this symmetry example the resolvent is identical to its own "grandfather" in the resolution derivation. More generally, we shall deal with resolvents that are subsumed by some of their ancestors in a linear resolution derivation, a phenomenon, which we shall call *ancestor subsumption*. Ancestor subsumption is a particular kind of *forward subsumption* (Overbeek 1975), that is the subsumption of a newly deduced clause by another, already present clause. One of this section's objectives is to characterize clause sets that admit ancestor subsumption. This approach is based on the following observation: A resolvent of two ground clauses cannot be subsumed by one of its parent clauses (a situation, which could be called *parent subsumption*), unless the other parent is a tautology. This can easily be seen: let $C = L_1L_2...L_n$ and $D = \neg L_1K_2...K_n$ be ground clauses and assume, C subsumes the resolvent $R = K_2...K_nL_2...L_n$. Then $L_1 \in R$ must hold and from $L_1 \notin L_2...L_n$ now follows $L_1 \in K_2...K_n$. Hence D is a tautology. It will turn out in this section that *cycles* are for ancestor subsumption what *tautologies* are for parent subsumption. This means that a resolvent R cannot be subsumed by some ancestor C, unless the set of ancestors of R contains a cycle[1]. Noncyclic clause sets thus have the nice property of excluding ancestor subsumption. A prominent example for this class of clause sets is Schubert's Steamroller (see Stickel 1986).

The concept of cyclic clause sets, which accounts for ancestor subsumption, proves also very useful in the context of decreasing the number of subsumption tests. Such a restriction can be achieved as follows: Having

---

[1] The notion of a cycle was introduced by Shostak (1976).

identified the cycles in a given clause set, a comparision of the ancestors of a given resolvent with the set of cycles suffices to possibly exclude these ancestors from being subsumers. Applied to the simple example from above, where R is the resolvent of two ground clauses C and D, we can state that the tests C subsumes R and D subsumes R both are superfluous, since these tests fail, except one of the clauses is a tautology. (We can assume, of course, that tautologies are always removed).

In the following we give a syntactical characterization of clause sets admitting ancestor subsumption. Our main result is as follows: Clause sets admitting ancestor subsumption possess *cycles*, whose elements are the *far parents* of the subsumed clause. First, we establish some results for the particular case of *parent subsumption*.

5.2.1 Definition:

A clause is **self-resolving**, if it resolves with a copy of itself.

The following lemma determines those clauses that possibly produce subsumed resolvents.

5.2.2 Lemma:

Let C, D, and R be clauses with $C \rightarrow_D R$. If R is a variant of C, then D is self-resolving.

Proof: For sake of simplicity we shall assume that $|C| = |D| = 2$. Let $C = NM$ and let $D = LK$ and let σ be a unifier of M and $\neg L$. From the assumption follows the existence of a renaming ρ with $R\rho = (NK)\sigma\rho = NM$. Now either

$N\sigma\rho = N$ and $K\sigma\rho = M$ or

$N\sigma\rho = M$ and $K\sigma\rho = N$.

Case 1: If $K\sigma\rho = M$, then $K\sigma\rho\sigma = M\sigma = \neg L\sigma$. Let $\varphi = \sigma\rho\sigma$. We show that there is a renaming substitution ρ' and a substitution ψ, such that $K\rho'\psi = \neg L\psi$. Let ρ' be a renaming substitution with

$dom(\rho') = dom(\sigma) \cap dom(\varphi)$ and $cod(\rho') \cap \mathbb{V}(L,K) = \emptyset$.

Define the substitution ψ with $dom(\psi) = dom(\sigma) \cup dom(\varphi) \cup cod(\rho')$ by

$\psi|_{dom(\sigma)} = \sigma$,

$\psi|_{dom(\varphi)\setminus dom(\sigma)} = \varphi$ and

$(x\rho')\psi = x\varphi$ for $x\rho' \in cod(\rho')$.

Then we have $x\psi = x\sigma$ for $x \in \mathbb{V}(L)$ and $y\rho'\psi = y\varphi$ for $y \in \mathbb{V}(K)$, hence $L\psi = L\sigma = K\varphi = K\rho'\psi$.

Case 2: Now we have $K\sigma\rho\sigma\rho = M$, and the lemma is proved analogously to case 1. ■

The next lemma shows that clauses, which only produce parent subsumed clauses, are tautologies. A tautology D with $|D|=2$, with linear literals[1], and without function symbols[2] will be called an **elementary tautology**.

5.2.3 Lemma:

Let D be a clause.

a) Suppose for all clauses C the following holds: $C \rightarrow_D R$ implies that C and R are variants. Then D is an elementary tautology.

b) Suppose for all clauses C the following holds: $C \rightarrow_D R$ implies that R is an instance of C. Then D is a tautology and $|D| = 2$.

c) Suppose for all clauses C the following holds: $C \rightarrow_D R$ implies that C subsumes R. Then D is a tautology.

Proof: a) Let $L = Pt_1...t_n$ be an arbitrary literal of D. Let C be the unit clause consisting of the literal $M=\neg L\rho$, for some variable renaming substitution ρ. Then there is a resolvent R of C and D and R is a subset of D. From the assumption follows that C is a variant of the resolvent R, that is, there is a renaming substitution σ, such that $R=M\sigma =\neg L\rho\sigma$. Thus C is the binary clause LR. Let μ be the renaming substitution $\rho\sigma$. If μ is not the identity, then there is some $x\in \mathbb{V}(L)$ such that $x\mu=x'$ with $x\neq x'$. Let a be an arbitrary constant not occurring in C nor D and let C' be the clause consisting of the literal $M' = \neg L\{x\rightarrow a\}$. Let R' be the resolvent of C' and D. Then, as a occurs in M', but not in $R'=\neg L\mu\{x\rightarrow a\}=\neg L\mu$, we obtain $R'\neq M'$, which is a contradiction. Hence μ is the identity and $R=\neg L$. If D contains a function symbol g, then g occurs in both literals of D, hence also in any resolvent of D. Let C_1 be the unit clause consisting of the literal $Px_1..x_n$, with $x_i\notin \mathbb{V}(D)$ for all $i\in\{1..n\}$. Then C_1 has a resolvent R_1 with D, however, as g occurs in R_1, but not in C_1, the clauses R_1 and C_1 cannot be variants, which is a contradiction to the assumption. Thus D cannot contain function symbols. In a similar way it can be shown that the literals of D are linear.

---

[1] Remember that a literal L is called *linear*, iff each variable occurs at most once in L.

[2] Note that this condition also precludes the occurence of constant symbols.

b) The same argument as for the first part of a) applies, except that μ is now an arbitrary substitution, and x' must be replaced by a term t.
c) If C is any clause resolving with D to some resolvent R, then C subsumes R, that is, there exists some subset R' of R, which is an instance of C. The assertion now follows from part b). ■

Our goal now is to generalize lemmata 5.2.2 and 5.2.3 to ancestor subsumption instead of parent subsumption, that is, we want to determine those clause sets $\mathcal{D}$, which - possibly or only - produce (ancestor) subsumed clauses. It will turn out that the appropriate generalization of self-resolving and tautologous clauses are *cyclic* clause sets in directed clause graphs.

In the following we shall deal with linear derivation without repetitions, that are derivations Δ of the form

$$\Delta = C \rightarrow_{D_1,\lambda_1} C_1 \rightarrow_{D_2,\lambda_2} \cdots \rightarrow_{D_n,\lambda_n} C_n,$$

where the D_i are pairwise distinct.

The number n is called the length of Δ. A derivation of length 0 is said to be *trivial*. It will prove useful to determine the clause graph structure for such a linear derivation. A linear derivation implies a certain direction of resolution steps, and this direction is reflected in the representation in form of a directed clause graph. We are thus interested in the structure of the directed clause graph G with $\mathbb{C}(G) = \{D_1,\ldots,D_n\}$, where each $\lambda \in \mathbb{A}(G)$ represents a link between some clause nodes of G, which is inherited to some λ_j. Note that the links in G, which inherit to some λ_j are uniquely determined.

5.2.4 Definition:

Let

$$\Delta = C \rightarrow_{D_1,\lambda_1} C_1 \rightarrow_{D_2,\lambda_2} \cdots \rightarrow_{D_n,\lambda_n} C_n$$

be a linear derivation without repetition. The **associated clause graph G(Δ)** is the graph G with

$\mathbb{C}(G) = \{C, D_1,\ldots,D_n\}$ and

$\Lambda(G) = \{\lambda \in \mathbb{N}(G)\times\mathbb{N}(G) \mid$ there is $i \in \{1,\ldots,n\}$, such that λ is inherited to λ_i qua $\lambda_1,\ldots,\lambda_{i-1}\}$

5.2.5 Example:

Consider the derivation $\Delta = C \rightarrow_{D_1,\lambda_1} C_1 \rightarrow_{D_2,\lambda_2} C_2$, which is shown in the left half of figure 5.2. The associated clause graph G(Δ) is shown in the right half of figure 5.2. Both links λ_2' and λ_2" of this graph are inherited to the link λ_2

of Δ. This example shows that there can be several links in G(Δ), which are inherited to the same link in the derivation Δ.

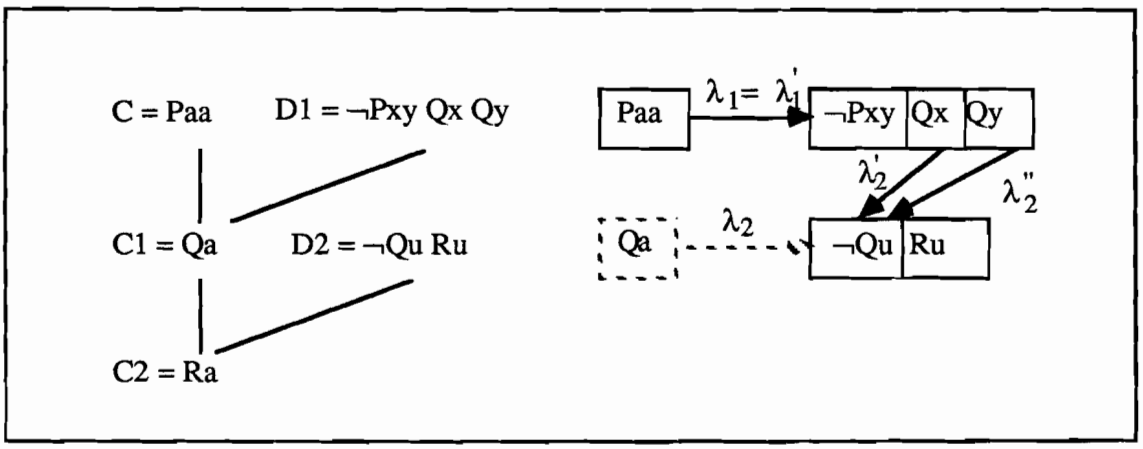


Fig. 5.2

In the sequel we shall make frequent use of the following lemma:

5.2.6 Lemma:

Let

$$\Delta = C \rightarrow_{D_1, \lambda_1} C_1 \rightarrow_{D_2, \lambda_2} \cdots \rightarrow_{D_n, \lambda_n} C_n$$

be a linear derivation, and let G be its associated clause graph. Let $\sigma \in \Sigma^*(\mathbb{A}(G))$. Then there is a derivation

$$\Delta' = C\sigma \rightarrow_{D_1\sigma, \lambda_1'} C_1' \rightarrow_{D_2\sigma, \lambda_2'} \cdots \rightarrow_{D_n\sigma, \lambda_n'} C_n'$$

with $C_n' \cong C_n$.

Proof: Follows by an induction argument on n from lemma IV.9 in Herold (1983). ■

In the rest of section 5.2 all occurring clause graphs will be directed clause graphs, unless stated otherwise. The following lemma shows that the structure of the clause graph associated to a linear derivation is a tree-like structure.

5.2.7 Lemma:

Let

$$\Delta = D_0 \rightarrow_{D_1, \lambda_1} C_1 \rightarrow_{D_2, \lambda_2} \cdots \rightarrow_{D_n, \lambda_n} C_n$$

be a linear derivation without repetition. Let G be the clause graph associated with Δ. Then G is an acyclic clause graph, which satisfies:

(i) Each clause node C of G is of the following form (see figure 5.3):

- If $C=D_0$, then $I(C)=\emptyset$
- If $C\neq D_0$, there is a literal $L\in C$ such that $I(L)\neq\emptyset$, $O(L)=\emptyset$, and $I(C)=I(L)$
- No two links in O(C) are adjacent

(ii) $\Sigma^*(\Lambda)\neq\emptyset$

Proof: (i) Let $C\in\mathbb{C}(G)$. If $C=D_0$, then obviously $I(C)=\emptyset$. If $C=D_j$ with $1\leq j\leq n$, then from the assumptions it is clear that there is some $\lambda\in\mathbb{A}(G)$, which is inherited to λ_j. Then $\lambda\in I(D_j)$, that is, $I(C)\neq\emptyset$. If there are two links λ_j' and λ_j'' in I(C), then both are inherited qua $\{\lambda_1,\ldots,\lambda_{j-1}\}$ to the same link λ_j with $D_{j-1}\rightarrow_{C,\lambda_j} D_j$. But this implies that λ_j' and λ_j'' are incident with the same literal L in C, that is, $\lambda_j',\lambda_j''\in I(L)$. Since λ_j' and λ_j'' were arbitrary, this proves $I(C)=I(L)$. If there were any link $\lambda\in O(L)$, then there would be some $k>j$ with $\lambda=\lambda_k$. But λ could not be inherited to the step k, since the literal L, which is incident with λ, is resolved away in the step j, and this is a contradiction. This proves $O(L)=\emptyset$. Suppose there are two links λ_j and λ_k incident with the same literal $K\in C$, and w.l.o.g. assume that $k>j$. But then again the link λ_k could not be inherited since the literal K is resolved away in step j.
(ii) is obvious (see, for instance, Herold (1983)). ■

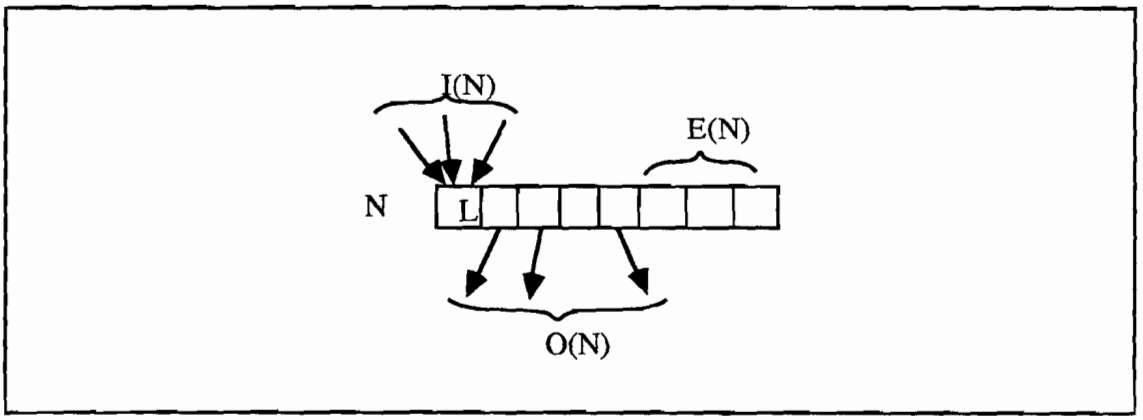


Fig. 5.3: Structure of a Branching Node

A clause node, which satisfies condition (i) of the previous lemma, will be called a **branching node** in the following. The literal L of the branching node C, which satisfies I(C)=I(L) is called the **I-literal**, each literal of C incident with a link in O(C) is an **O-literal** of C. A clause graph G satisfying conditions (i) and (ii) of the previous lemma will be called a **branching tree** with **root** D_1.

5.2.8 Definition:

Let G be a branching tree and let $\sigma \in \Sigma^*(\mathbb{A}(G))$. Then

$$res(G) = \bigcup_{D \in \mathbb{C}(G)} E(D)\sigma$$

is the **residue** of G.

In the following we shall prove that the residue of a branching tree G(Δ) is just the last clause C_n of the corresponding linear derivation Δ.

5.2.9 Lemma:

Let

$$\Delta = C \rightarrow_{D_1,\lambda_1} C_1 \rightarrow_{D_2,\lambda_2} \cdots \rightarrow_{D_n,\lambda_n} C_n$$

be a linear derivation without repetition, and let $\Delta' = C_1 \rightarrow_{D_2} \cdots \rightarrow_{D_n} C_n$. Then

$$res(G(\Delta)) = res(G(\Delta')).$$

Proof: Let G=G(Δ), G'=G(Δ'), and let $G \rightarrow_{\lambda_1} G''$. Then $G' = (G'')^{\{C,D_1\}}$, and $\Sigma^*(\mathbb{A}(G)) = \Sigma^*(\mathbb{A}(G'))$ holds. We have to show that

$$\bigcup_{D \in \mathbb{C}(G)} E(D)\sigma = \bigcup_{D' \in \mathbb{C}(G')} E(D')\sigma,$$

Let $L \in E(D)$ in G for some $D \in \mathbb{C}(G)$, that is, L is not incident with any link in Λ. Suppose there is some $\lambda' \in \mathbb{A}(G')$, which is incident with L. Then λ' must be inherited from some $\lambda \in \mathbb{A}(G)$, which is also incident with L. This is a contradiction, hence L is not incident with any link in $\mathbb{A}(G')$ either, which implies $L \in E(D)$ in G'. This proves $\bigcup_{D \in \mathbb{C}(G)} E(D)\sigma \subseteq \bigcup_{D' \in \mathbb{C}(G')} E(D')\sigma$.

Conversely, we can also show that $\bigcup_{D' \in \mathbb{C}(G')} E(D')\sigma \subseteq \bigcup_{D \in \mathbb{C}(G)} E(D)\sigma$, which implies the assertion of the lemma. ∎

5.2.10 Example:

Let

$$\Delta = PR\neg S \rightarrow_{\neg RW} PW\neg S \rightarrow_{SW} PW \rightarrow_{\neg WQ} PQ.$$

Then $\Delta' = PW\neg S \rightarrow_{SW} PW \rightarrow_{\neg WQ} PQ$. The corresponding branching trees G(Δ) and G(Δ') are shown in figure 5.4. The links λ_i^* are inherited from the corresponding link λ_i qua λ_1.

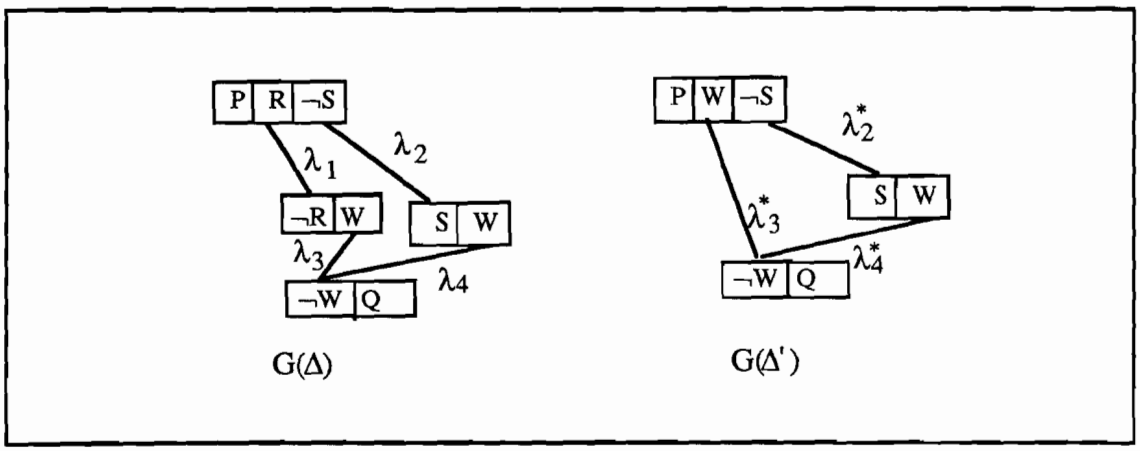


Fig. 5.4

5.2.11 Corollary:

Let

$$\Delta = C \rightarrow_{D_1} C_1 \rightarrow_{D_2} \dots \rightarrow_{D_n} C_n$$

be a linear derivation without repetition. Then

$$C_n = \text{res}(G(\Delta)).$$

Proof: An induction argument using the previous lemma shows that res(G(Δ)) = res(G(Δ^n)), where Δ^n is the trivial derivation C_n. Thus $\mathbb{C}(G(\Delta^n)) = \{C_n\}$ and $\mathbb{A}(G(\Delta^n)) = \emptyset$), which implies res(G($\Delta^n$)) = E($C_n$) = C_n. ■

Next we shall give the characterization of ancestor subsumption for the ground case. Ancestor subsumption is expressed by a linear derivation $\Delta = C \rightarrow_{D_1} C_1 \rightarrow_{D_2} \dots \rightarrow_{D_n} C_n$, where C subsumes C_n. From lemma 5.2.7 it follows that the associated clause graph is a branching tree. In particular, if C is a unit clause, then there is a linear derivation $D_1 \rightarrow_{D_2} C_2' \rightarrow_{D_3} \dots \rightarrow_{D_n} C_n'$, and the branching tree associated with this derivation is $G(\Delta)^C$.

A **semicycle** G is a clause graph, which satisfies:

(i) Each node $C \in \mathbb{C}(G)$ is a branching node with $|I(C)| > 0$ and $E(C) = \emptyset$.

(ii) There is a **special node** $C_0 \in \mathbb{C}(G)$ such that each cyclic path of G passes C_0.

(iii) All occurring substitutions are compatible, that is, $\Sigma^*(\mathbb{A}(G)) \neq \emptyset$.

It is obvious that the subgraph $G^{I(C_0)}$ of a semicycle G with special node C_0 is a branching tree with root C_0. The residue of the semicycle G is defined by the residue of the branching tree $G^{I(C_0)}$.

The fact that each node C of a semicycle has an incoming link guarantees the existence of a cyclic path in a semicycle. The concept of semicycles is easily seen to be a generalization of self-resolving clauses, for a semicycle consisting of only one clause node C forces this clause to be self-resolving. The semicycle G is called a **cycle**, if $|O(C)| = 1$ holds for each $C \in \mathbb{C}(G)$. It is easy to see that a cycle is just a single cyclic path with nodes of length two, whence condition b) of the definition of a semicycle is satisfied for each node of a cycle.

A cycle is just what Shostak (1976) and (1979) calls a *loop*. This notion also corresponds to the notion of *recursive predicates* in the terminology of deductive databases (Vieille 1987, Ohlbach 1988) and logic programming.

5.2.12 Example:

The graphs G_1 and G_2, which are shown in figure 5.5, are semicycles; G_1 is a cycle. It can be seen that each node of a cycle may be chosen to be the "special node" C_0. As to semicycles, still several, but in general not all nodes have this property. The clauses ¬R¬W and ¬UV, for instance, cannot be chosen to be the special node of G_2.

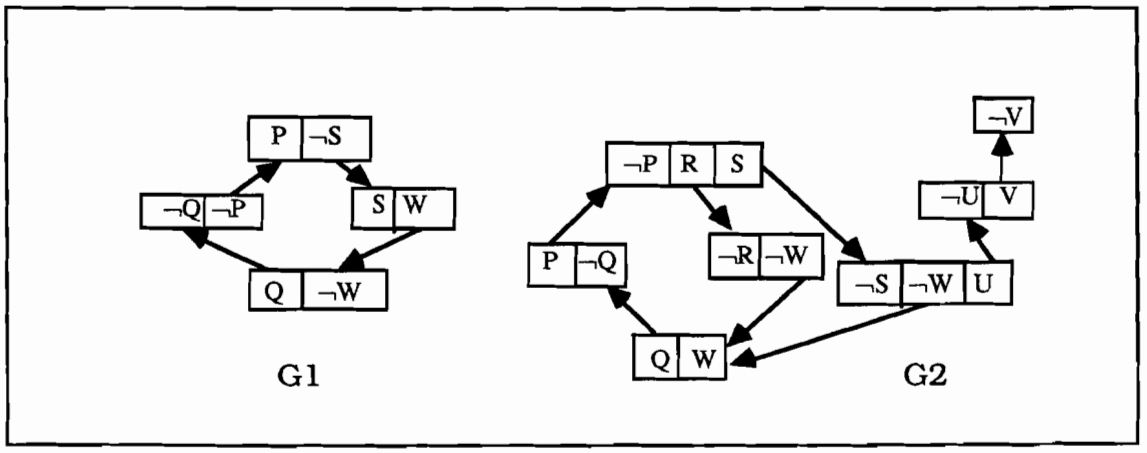


Fig. 5.5

5.2.13 Theorem:

Let $\Delta = C \rightarrow_{D_1} C_1 \rightarrow_{D_2} \ldots \rightarrow_{D_n} C_n$ be a ground derivation with unit clause C, and let $G=G(\Delta)^C$.

a) If $C_n = C$, then G is contained in a semicycle G' with special node D_1 and $G = (G')^{I(D_1)}$.

b) If $C_n \subseteq C$, then G is contained in a cyclic graph G' with $\mathbb{C}(G)=\mathbb{C}(G')$ and $\mathbb{\Lambda}(G)\subseteq\mathbb{\Lambda}(G')$.

Proof: a) Let $C=C_n=\{L\}$. From the branching tree G we construct a semicycle in the following way: From $res(G)=\{L\}=\bigcup_{C\in\mathbb{C}(G)} E(C)$ we can conclude that for each $C\in\mathbb{C}(G)$ either $E(C)=\emptyset$ or $E(C)=\{L\}$. Moreover, since C is a unit clause, the I-literal of D_1 must be $\neg L$. For each $C\in\mathbb{C}(G)$ with $E(C)=\{L\}$ let $\lambda_C = L\neg L \in O(C)\cap I(D_1)$. Let

$$\Lambda' = \Lambda \cup \bigcup_{C\in\mathbb{C}(G)} \lambda_C$$

Let G' be the graph with $\mathbb{C}(G')=\mathbb{C}(G)$ and $\mathbb{\Lambda}(G')=\Lambda'$. From the construction it is clear that G' satisfies the conditions of a semicycle and $G = (G')^{I(D_1)}$.

b) Let $C=\{L\}$. In a way similar to a) we conclude that $\{L\}\subseteq \bigcup_{C\in\mathbb{C}(G)} E(C)$, that is, there exists a node $C\in\mathbb{C}(G)$ with $L\in E(C)$. Let $\lambda = L\neg L \in O(C)\cap I(C)$, and define the graph G' by $\mathbb{C}(G')=\mathbb{C}(G)$ and $\mathbb{\Lambda}(G')=\Lambda\cup\{\lambda\}$. Then G' is a cyclic graph with $\mathbb{\Lambda}(G)\subseteq\mathbb{\Lambda}(G')$. ■

In order to generalize the previous lemma to the general nonground case, we first have to prove some properties of substitutions.

5.2.14 Lemma:

Let L and K be literals and σ be a substitution.

a) If $L\sigma$ and $K\sigma$ are weakly unifiable, then so are L and K. Moreover, there is a substitution θ and a renaming ρ, such that
$L\sigma\theta = K\rho\sigma\theta$ and
$dom(\sigma) \cap \mathbb{V}(cod(\rho)) = dom(\rho) \cap \mathbb{V}(cod(\sigma)) = \emptyset$.

b) If there is a substitution φ and a renaming ρ, such that $L\rho\varphi = K\varphi$ holds, then $L\rho'\varphi = K\rho'\varphi$ for some renaming ρ'.

Proof: a) see (Herold 1983), lemma III.9.
b) see (Herold 1983), lemma III.8. ■

5.2.15 Definition:

Let $\lambda,\lambda_1,\lambda_1'$ be R-links, such that λ_1 is inherited qua λ to λ_1'. We say that λ_1' is inherited qua λ **solely** from λ_1, iff there is no link $\lambda_2\neq\lambda_1$, which inherits qua λ to λ_1'.

5.2.16 Lemma:

Suppose we have clauses C, D, and E, and undirected, compatible links $\lambda_1 \in \mathbb{A}(C) \cap \mathbb{A}(D)$, $\lambda_2 \in \mathbb{A}(D) \cap \mathbb{A}(E)$, which are not adjacent. Let λ_2' be a link inherited solely from λ_2 qua λ_1. If there are derivations

$C \to_{\lambda_1} C_1 \to_{\lambda_2} C'$ and

$D \to_{\lambda_2} D_1 \to_{\lambda_1} D'$,

then $D' \lesssim C'$ holds.

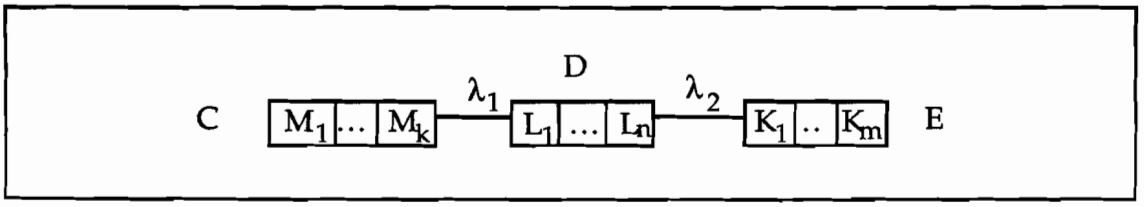


Fig. 5.6

Proof: Let C, D, E be as in figure 5.6. For i=1,2, let $\sigma_i \in \Sigma^*(\lambda_i)$, and let $\sigma \in \sigma_1 * \sigma_2$. Obviously, C' and D' can also be derived as resolvents of the clauses $C\sigma$, $D\sigma$, and $E\sigma$. Let C_1'' be the resolvent of $C\sigma$ and $D\sigma$, then $C_1'' \cong (M_1 \ldots M_{k-1} L_2 \ldots L_n)\sigma$. From the assumption of the lemma follows that $M_i\sigma_1 \neq L_j\sigma_1$ for any appropriate i,j. Suppose $L_j\sigma = \ldots = L_n\sigma$, $2 \leq j \leq n$. Then

$C' \cong (M_1 \ldots M_{k-1} L_2 \ldots L_{j-1} K_2 \ldots K_m)\sigma$

If D_1'' is the resolvent of $D\sigma$ and $E\sigma$, then $D_1'' = (L_1 \ldots L_{j-1} K_2 \ldots K_m)\sigma$. Suppose $K_j\sigma = L_1\sigma$ for some $j \in \{2,\ldots,m\}$. W.l.o.g let j=m. Then

$D' \cong (M_1 \ldots M_{k-1} L_2 \ldots L_{j-1} K_2 \ldots K_{m-1})\sigma \lesssim C'$.

Otherwise, if $K_j\sigma \neq L_1\sigma$ for all $j \in \{2,\ldots,m\}$, we have $D' \cong C'$. In either case $D' \lesssim C'$ holds. ■

In particular, if C is a unit clause, there can be no link in $\mathbb{A}(C)$, which is inherited to λ_2', and so the last assumption of the lemma is trivially satisfied.

5.2.17 Theorem:

Let $\Delta = C \to_{D_1} C_1 \to_{D_2} \ldots \to_{D_n} C_n$ be a derivation with unit clause C, and let $G = G(\Delta)^C$.

a) If $C_n \cong C$, then G is contained in a semicycle G' with special node D_1 and $G = (G')^{I(D_1)}$.

b) If C_n is subsumed by C, then G is contained in a weakly cyclic graph G' with $\mathbb{C}(G) = \mathbb{C}(G')$ and $\mathbb{A}(G) \subseteq \mathbb{A}(G')$.

Proof: a) In view of lemma 5.2.13 the only thing we have to prove is the compatibility of the substitutions in $\Lambda=\mathbb{\Lambda}(G')$, that is, $\Sigma^*(\Lambda)\neq\emptyset$. Let

$$\Delta = C \rightarrow_{D_1} C_1 \rightarrow_{D_2} \dots \rightarrow_{D_n} C_n.$$

Since C is a unit clause, there is a derivation

$$\Delta' = D_1 \rightarrow_{D_2} C_2' \dots \rightarrow_{D_n} C_n'$$

Let $D=C_n'$, and let $C \rightarrow_D C'$. Since C is a unit clause, the previous lemma implies that $C' \subsetneq C_n$, and since $C'\neq\square$, $C' \cong C_n \cong C$ holds (see figure 5.7).
Thus, according to lemma 5.2.2, D is a self-resolving clause of the form $L'K_1'..K_n'$, with a renaming ρ and a substitution τ, such that $L'\tau = \neg K_i'\rho\tau$ for each $i\in\{1,..,n\}$ holds. Moreover, since $D=res(G(\Delta))$, we have $L'=L\sigma$ and $K_i'=K_i\sigma$, where L is the I-literal of D_1 and the K_i are the O-literals of the predecessors of D_1. Take any $K\in\{K_1,..,K_n\}$. We have $L\sigma\tau = \neg K\sigma\rho\tau$ and from lemma 5.2.14 a) follows that there is some substitution θ with $L\sigma\theta = \neg K\rho'\sigma\theta$ for some renaming ρ', and from part b) of the same lemma follows that $L\rho''\sigma\theta = \neg K\rho''\sigma\theta$ for some renaming ρ''. Let $\lambda = \rho''\sigma\theta$, then λ is a unifier of L and $\neg K$, hence λ is an instance of the most general unifier φ of L and $\neg K$. We show that σ and φ are compatible substitutions. Let $\lambda = \varphi\lambda'$. The renaming ρ'' can be chosen, such that $dom(\sigma) \cap \mathbb{V}(cod(\rho'')) = dom(\rho'') \cap \mathbb{V}(cod(\sigma)) = \emptyset$, that is, $\sigma\rho'' = \rho''\sigma$. This implies

$$\sigma\lambda = \sigma\rho''\sigma\theta = \sigma\sigma\rho''\theta = \lambda = \varphi\lambda' = \varphi\varphi\lambda' = \varphi\lambda,$$

hence φ and σ are compatible. We have shown that σ is compatible with the unifier of an arbitrary link in $I(D_1)$, that is, all occurring unifiers are compatible and condition (iii) of the definition of a semicycle is satisfied.

b) is proved analogously to a) ■

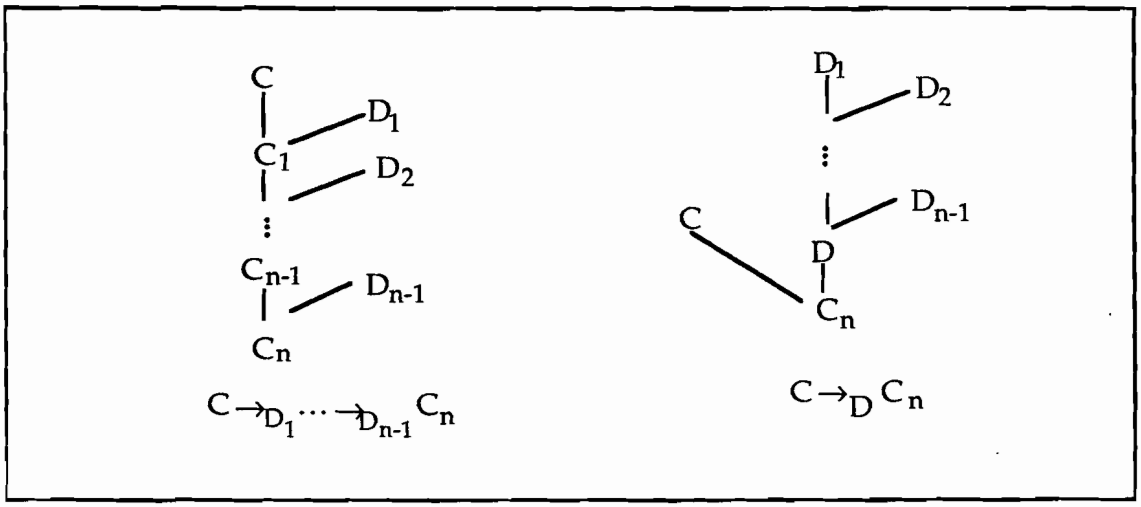


Fig. 5.7

Our next goal is to identify those graphs that allow only derivations with ancestor subsumption, corresponding to lemma 5.2.3. The possible derivations from a graph G are those derivations Δ, for which the associated graph $G(\Delta)$ is the branching tree of any cyclic subgraph of G.

5.2.18 Theorem:

The semicycle G is a cycle, iff each cyclic subgraph of G is a semicycle.

Proof: Obviously, if G is a cycle, then each cyclic subgraph of G is also a cycle. To prove the converse, we choose a special node C_0 from G, and construct inductively a subgraph of G as follows: If we have already constructed $(C_0,\lambda_0,\ldots,C_i)$, then take arbitrarily any $\lambda \in O(C_i)$ and let C be the node of G which satisfies $\lambda \in I(C)$. Set $\lambda_i=\lambda$, and $C_{i+1}=C$. Since G is cyclic, there must be $j,k \in \mathbb{N}$ with $j<k$ and $C_k=C_j$, that is, we have constructed a cyclic path. Since each cyclic path must pass C_0, we can w.l.o.g. assume that $C_k=C_0$. Moreover, we take the smallest such k. Let $G' = (\{C_i \mid i=1,\ldots,k\},\{\lambda_i \mid i=1,\ldots,k\})$. Assume G is not a cycle. Then there exists a branching node $C \in \mathbb{C}(G)$ with $|O(C)|>1$. W.l.o.g. we assume that this node is C_0. Since $|O(C_0)|>1$, there is a link $\lambda \in O(C_0)$ with $\lambda \neq \lambda_0$ and a $L \in C_0$ with $\lambda \in O(L)$. Since $C_j \neq C_0$ for $0<j<k$, the link λ_0 is the only link in $O(C_0) \cap \mathbb{A}(G')$, which implies $L \in E(C_0)$ in G'. This violates the condition $E(C)=\emptyset$ for semicycles, which contradicts the assumption of the lemma. Thus G must be a cycle. ■

5.2.19 Corollary:

Let G be a semicycle consisting of ground clauses. G is a cycle, iff for any derivation $\Delta = C \rightarrow_{D_1} C_1 \rightarrow_{D_2} \ldots \rightarrow_{D_n} C_n$, where $\{D_1,\ldots,D_n\}$ are the nodes of a cyclic subgraph of G, $C = C_n$ holds.

Proof: The assumption of the theorem can be stated as follows: Each cyclic subgraph of G is a semicycle. The assertion now follows from the previous theorem. ■

Thus cycles are the only clause sets with the following property: Take any clause node D of the cycle, and an arbitrary clause C, which resolves with D. Construct derivations $\Delta_k = C \rightarrow_{D_1} C_1 \rightarrow_{D_2} \ldots \rightarrow_{D_k} C_k$, where $D_1 = D$, and D_i is the successor of D_{i-1} in the cycle. Then there is a $k \in \mathbb{N}$, such that $C_k = C$ holds.

The proof of theorem 5.2.12 showed that semicycles are structures that shrink to self-resolving clauses. Those particular semicycles that can be reduced to tautologies, are characterized by a particular property of their substitutions:

5.2.20 Definition:

Let G be a semicycle with special node C_0. Let $\sigma \in \Sigma^*(\Lambda \setminus I(C_0))$, and let $\tau \in \Sigma^*(I(C_0))$. The semicycle G is called **C_0-complete**, if σ is an instance of τ, that is $L\sigma = K\sigma$ holds for all literals L and K joined by a link in $I(C_0)$. G is called **complete**, iff it is complete for each choice of C_0.

Let G be a complete semicycle. Let $D = (LK_1..K_m)\sigma$ be the residue of G. As for each $i \in \{1,..,m\}$, K_i and L are complementary unifiable under τ, and σ is an instance of τ, we have the following corollary:

5.2.21 Corollary:

The semicycle G is complete, iff res(G) is a tautology for any choice of the special node of G.

Complete cycles can also be characterized in the following way: There exists a clause C and a substitution φ, which is the identity on C, such that all links in the cycle become complementary pairs of literals. This is the way Bibel (1987) defines what he calls *tautological* cycles.

In analogy to elementary tautologies, complete cycles, all literals of which are linear and function free, will be called **elementary cycles**.

5.2.22 Theorem:

Let G be a semicycle.

a) G is an elementary cycle, iff $C \cong C_n$ holds for any derivation $\Delta = C \rightarrow_{D_1} \ldots \rightarrow_{D_n} C_n$, for which $\{D_1,\ldots,D_n\}$ are the nodes of a cyclic subgraph of G.

b) G is a complete cycle, iff C_n is an instance of C for any derivation $\Delta = C \rightarrow_{D_1} \ldots \rightarrow_{D_n} C_n$, for which $\{D_1,\ldots,D_n\}$ are the nodes of a cyclic subgraph of G.

c) G is complete, iff C subsumes C_n for any derivation $\Delta = C \rightarrow_{D_1} \ldots \rightarrow_{D_n} C_n$, for which $\{D_1,\ldots,D_n\}$ are the nodes of a cyclic subgraph of G.

Proof: a) The cyclic structure of G is provided by corollary 5.2.19. Choose any C to be the special node of G. As in the proof of theorem 5.2.17, let $D = (LK)\sigma$ be the residue of G. Then the relation $C \rightarrow_D C_n$ implies $C_n \cong C$ for each clause C that resolves with D. From lemma 5.2.3 follows that D is a tautology, which is function and constant free, which shows that the same holds for L and K. As C was chosen arbitrary, all elements of $\mathbb{C}(G)$ are function and constant free. Since D is a tautology, the cycle G is complete.
b) and c) are proved analogously. ■

Note that the definition of the derivation relation $\rightarrow_D$ does not capture ancestor resolution, which is part of complete linear strategies.

5.2.23 Example:

Let

$$\Delta = P \rightarrow_{\neg PRS} RS \rightarrow_{\neg SR} R \rightarrow_{\neg RT} T \rightarrow_{\neg T \neg RP} \neg RP \rightarrow_R P$$

be a linear derivation with ancestor resolution and ancestor subsumption. The corresponding graph G is shown in figure 5.8. It is easy to see that this graph is not a semicycle. The bold faced links represent the "ancestor resolution links".

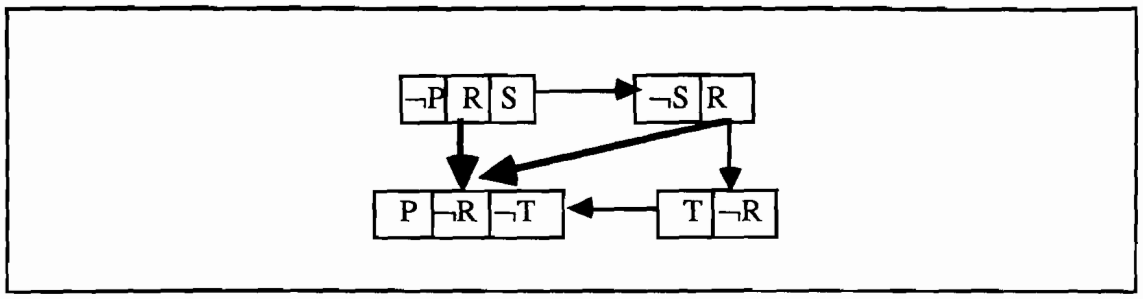


Fig. 5.8

The following lemma shows that it is sufficient to look for cycles in an initial clause set during a resolution refutation, as "new" cycles cannot be generated by resolution. For any clause set S, $\Re(S)$ denotes the resolution closure of S, that is the smallest clause set containing S and closed under the resolution operation.

5.2.24 Lemma:

The clause set S contains a semicycle, iff $\Re(S)$ contains a semicycle.

Proof: Let R be a resolvent of clauses C and D with C, D$\in$ S and assume that R is a node of a semicycle G. We show that C and D are also nodes of a semicycle.

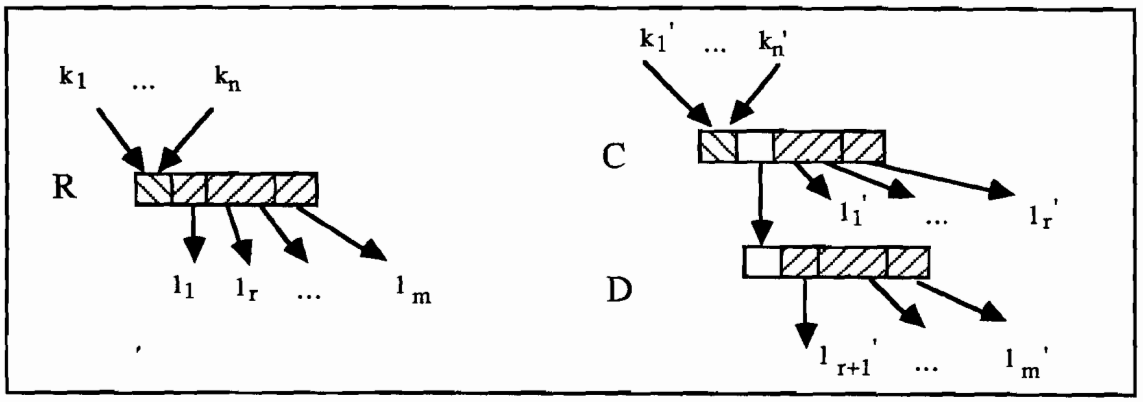


Fig.5.9

Each link k_i and l_j of G going into or coming from the node R must be inherited from some link k_i' and l_j', respectively, as shown in figure 5.9. The diagram illustrates that these links are links of a semicycle G', which contains nodes C and D. ■

5.3 Redundancy Caused by Subsumed Links

On closer inspection of proofs using a deduction system consisting solely of the resolution rule, it turns out that a lot of redundancy is produced by permuting the order of resolution steps. For instance, it does not matter, whether the clause ¬P¬QR is first resolved with the clause P and then the resolvent ¬QR is resolved with the clause Q, or vice versa; the result always is the clause R. This observation has led to the hyperresolution strategy (Robinson 1965) and the UR-resolution strategy (McCharen 1976), which both combine several resolution steps, thus abstracting from their order. Clause graph resolution too avoids this multiple derivation of identical clauses. There, it is a cooperation between *link deletion* and *link inheritance*, which precludes one of the two resolution sequences. Link inheritance denotes the fact that each resolvent inherits all its links by its parent clauses. Deleted links thus cannot be inherited, and link deletion thus not only prevents the single resolution step connected with the removed link, but it also accounts for the exclusion of subsequent resolution steps connected with inherited links. The following example illustrates this effect of link deletion and link inheritance.

5.3.1 Example:

Consider the clauses C = PQR, D_1 = ¬PS, and D_2 = ¬QW (see figure 5.10). Resolving first C with D_1 along λ_1 yields the resolvent C_1 = QRS, and the link λ_1 is removed afterwards. The resolvent C_2 = PRW of C and D_2 thus cannot inherit λ_1, and there remains only one possibility to derive the resolvent C_{12} = WRS.

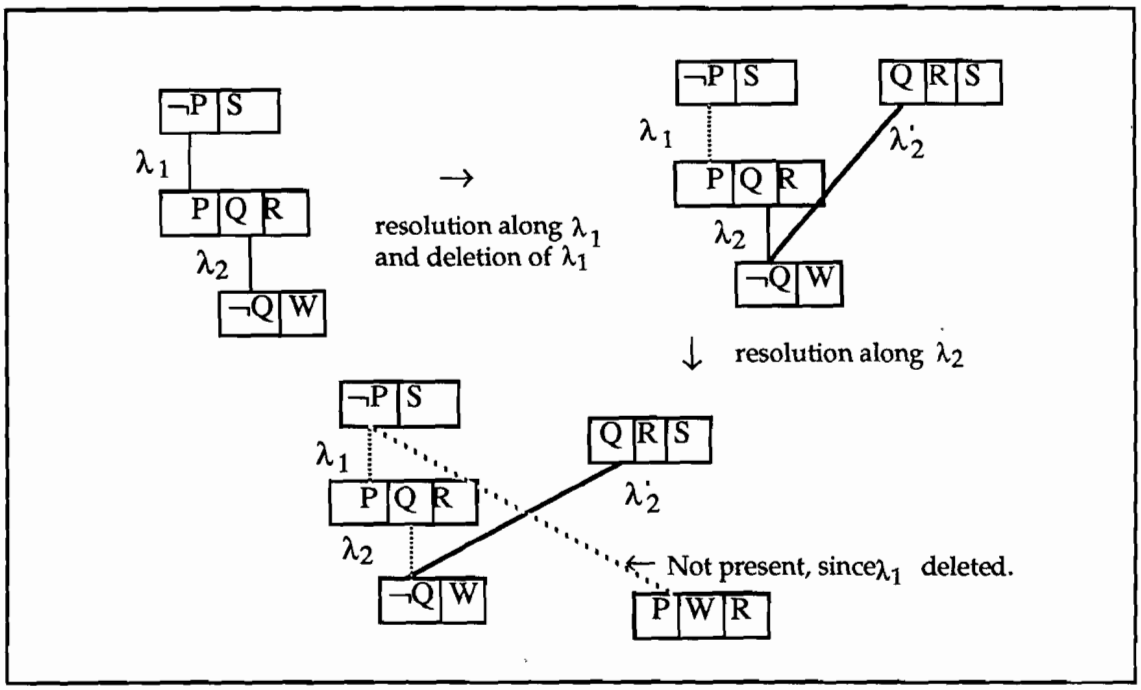


Fig. 5.10: Link Inheritance and Link Deletion

In general, however, the order, in which clauses are processed in a sequence of resolution steps, cannot simply be permuted without changing the result. Consider for instance the clauses Pc, ¬Px Pa , and ¬Px Pb. If Pc is first resolved with ¬Px Pa and the resulting clause Pa is resolved with ¬Px Pb, the clause Pb is obtained. By resolving first Pc with ¬Px Pb and then the result with ¬Px Pa, the clause Pa is derived.

There are, however, still other clause structures, which are invariant under the permutation of the resolution steps' order, and to which hyperresolution does not apply. The transitivity clause Pxy & Pyz ⇒ Pxz is a well-known example for such structures. Suppose, we are given the unit clauses Pab, Pbc and Pcd and the transitivity axiom holds for the predicate P. There are two different ways to deduce the clause Pad by two applications of the transitivity axiom, which differ only in the resolution steps' order. Still worse, given n clauses Pa_1a_2, Pa_2a_3, ..., $Pa_{n-1}a_n$, the number of different ways to derive the clause Pa_1a_n amounts to $2^{n-1}(n!)^{-1}\prod_{i=1}^{n-1}(2i - 1)$, which is of order $O(4^n)$ (see Lüneburg 1976). It is also easy to see that the clause graph mechanisms of link inheritance and link deletion do not prevent the multiple derivation of identical resolvents in this case.

Another example for the permutability of resolution steps consists of the clauses Pa, $C_1 = \neg Px\ Pf^2x$ and $C_2 = \neg Px\ Pf^3x$. There are two ways to deduce the clause Pf^5a, which differ only in the choice of resolving Pa first with C_1 or with C_2.

The difference between this example and the hyperresolution example is revealed by considering the clause graph structure (Figure 5.11):

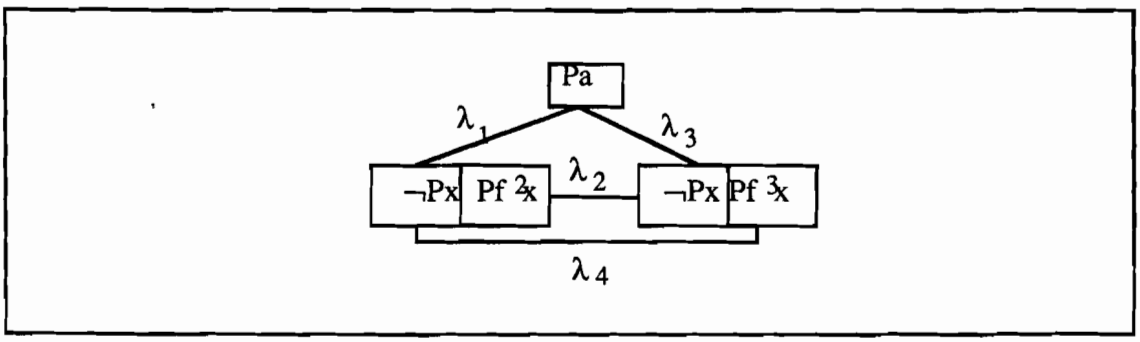


Fig. 5.11

In the hyperresolution example the order, in which the electrons are resolved against the nucleus, corresponds to the order of selecting the links that join the nucleus with the electrons. In the last example, however, the two different ways to produce the clause Pf^5x correspond to different pairs of links: One deduction proceeds on links λ_1 and λ_2, while the other uses links λ_3 and λ_4. The very reason for the permutability of the resolution steps in this particular example becomes clear, when we take into account that processing link λ_2 or processing link λ_4 yields the same resolvent up to renaming, namely $\neg Px\, Pf^5x$. As a natural generalization of the variant notion these will be called variants.

More general, the subsumption relation between clauses will be extended to links by calling those links subsumed that produce a subsumed resolvent. In the sequel it will be shown that the type of redundancy, which is inherent in subsumed links, is inherited via the usual link inheritance. In particular, it will be shown that the inheritance of subsumed links accounts for the redundancies occurring in the two introductory examples.

The syntactic description of the permutability of resolution steps by means of subsumed links is but the first step to overcome the difficulty with the multiple derivation of identical clauses. In this particular example it is

rather obvious, how to cope with the problem of deducing redundant clauses: The solution, very simply, consists in the appropriate selection of links to process. The suitable order to proceed is resolving first on link 2 (or 4, respectively), obtaining the clause $\neg PxPf^5x$. Both links 2 and 4 can now be removed, which inhibits further derivations starting from Pf^2a or from Pf^3a. The only way to deduce the clause Pf^5a now consists in resolving Pa with $PxPf^5x$. This approach is based on the observation that redundant information, if not removed, usually causes an inflation of redundancy. This is the case for subsumed clauses as well as for tautologous clauses: Each descendant of such a clause is redundant, too. In the following it will be shown that the same holds for the redundancy inherent in subsumed links. In other words, each link, which is inherited from a subsumed link, is again subsumed.

We shall also consider a very particular kind of subsumed links, which will be defined by *symmetric* clauses. A clause is symmetric, if it has literals L_1 and L_2 such that resolving on L_1 always yields the same resolvent as resolving on L_2. For instance, the clause Pxy Pyx is symmetric. Symmetric clauses are a source of redundancy very similar to clauses with subsumed links. This section provides a complete syntactic description of symmetry, which is the basis for an appropriate reduction consisting in the restriction of resolution to one of the arbitrarily chosen symmetry literals.

Note that throughout this section all clause graphs are undirected graphs.

5.3.2 Definition:

Let λ_1, λ_2 be R-links, and for i=1,2, let C_i be the resolvent along λ_i. The subsumption and variant relations between links are defined by

$\lambda_1 \leq \lambda_2$ (λ_1 subsumes λ_2), iff $C_1 \leq C_2$

$\lambda_1 \cong \lambda_2$ (λ_1 is a variant of λ_2), iff $C_1 \cong C_2$.

Although a resolution step using a subsumed link is obviously unnecessary, it is not clear that the subsumed link itself can be removed, since such removal implies the deletion of all inherited links. Our main theorem will justify the removal of subsumed links, by showing that each link λ', which is inherited *solely* from a subsumed link λ, is itself subsumed. If there are, however, other (non-subsumed) links besides λ, which also inherit to λ', then λ' is not subsumed. But also in this case the removal of λ causes no problems, since generation of λ' is guaranteed by inheritance.

5.3.3 Lemma:

Let C and D be clauses with $C \leq D$. If R is a resolvent of D with any other clause, then there is a clause C' with $C' \leq R$.

Proof: Since D is subsumed by C, it can be written in the form $D = C\theta \cup D'$. Let E be a clause resolving with D, and let L',K be literals with most general unifier σ', such that $R' = (D \setminus \{L'\})\sigma' \cup (E \setminus \{K\})\sigma'$. If $L' \in D'$, then

$$R' = (C\theta \cup D' \setminus \{L'\})\sigma' \cup (E \setminus K)\sigma' = C\theta\sigma' \cup (D' \setminus \{L'\})\sigma' \cup (E \setminus K)\sigma'$$

is subsumed by C. Otherwise, if $L' \in C\theta$, then there is a literal $L \in C$ with $L\theta = L'$, and $R' = (C \setminus \{L\})\theta\sigma' \cup D'\sigma' \cup (E \setminus K)\sigma'$. L' and K are unifiable, hence so are L and K. Let σ be a most general unifier of L and K. Since different clauses are assumed to be variable disjoint, $\mathbb{V}(E) \cap \mathrm{dom}(\theta) = \emptyset$. Hence $L\theta\sigma' = L'\sigma' = K\sigma' = K\theta\sigma'$, that is, $\theta\sigma' \in \mathrm{uni}(L,K)$, which implies $\sigma \leq \theta\sigma'$. Consider the resolvent $R = (C \setminus \{L\})\sigma \cup (E \setminus K)\sigma$ of C and E. We have

$$(C \setminus \{L\})\sigma \cup (E \setminus K)\sigma \leq (C \setminus \{L\})\theta\sigma' \cup (E \setminus K)\theta\sigma' = (C \setminus \{L\})\theta\sigma' \cup (E \setminus K)\sigma' = E,$$

and E is subsumed by R. ■

5.3.5 Theorem:

Let λ, λ_1 and λ_2 be R-links, such that $\lambda_1 \leq \lambda_2$ holds, and λ is compatible with λ_2. If λ_2' is inherited solely from λ_2 qua λ, then there is a link λ^*, which is not inherited from λ_2 neither inherited qua λ_2, with $\lambda^* \leq \lambda_2'$.

Proof: For i=1,2, let C_i and D_i be the clauses incident with λ_i, let E_i be the resolvent along λ_i, and let C be incident with λ (see figure 5.12). Note that the clauses C_1, C_2, as well as the clauses D_1, D_2 need not be distinct. Then $E_1 \leq E_2$ holds by the assumption of the lemma. Let λ' be the link inherited from λ qua λ_2. From lemma 5.2.16 follows that the resolvent E_2' of E_2 with C subsumes the resolvent along λ_2', that is, $\lambda' \leq \lambda_2'$. From the previous lemma follows that E_2' itself is subsumed either by E_1 or by the resolvent of E_1 with C. In the latter case there exists a link $\kappa' \in \mathbb{A}(E_1) \cap \mathbb{A}(C)$, which is inherited from some link $\kappa \in (\mathbb{A}(C_1) \cup \mathbb{A}(D_1)) \cap \mathbb{A}(C)$. Hence either $\lambda_1 \leq \lambda' \leq \lambda_2'$ or $\kappa' \leq \lambda' \leq \lambda_2'$, and both λ_1 and κ' are not inherited from λ_2 neither inherited qua λ_2. ■

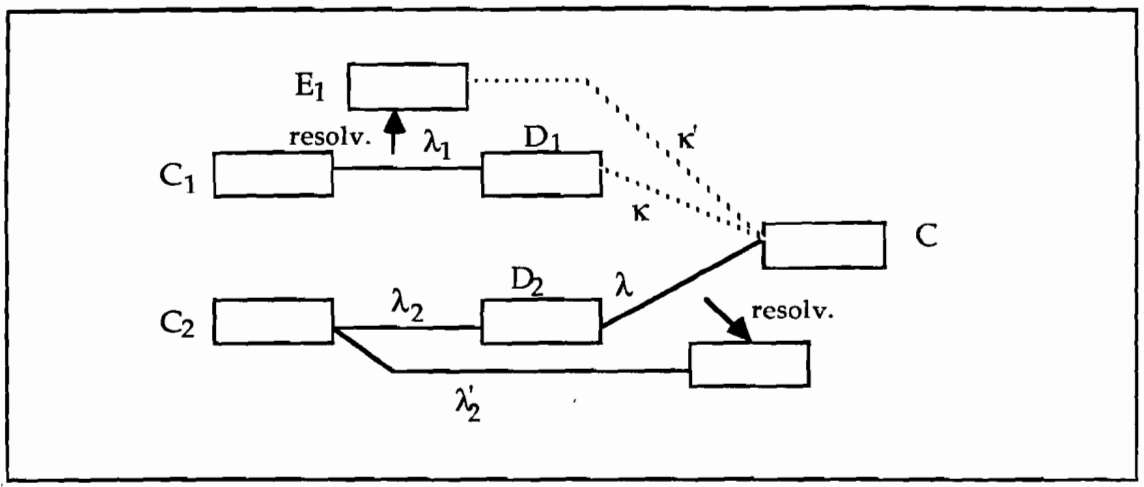


Fig. 5.12: λ_2' is subsumed either by λ_1 or by κ'

5.3.6 Example:

In our introductory example the multiple derivation of identical resolvents by the transitivity clause results from the inheritance of two internal links $\lambda_1 \cong \lambda_2$ in the transitivity clause ¬Pxy ¬Pyz Pxz. Both resolvents are copies of the 3-transitivity clause ¬Pxy ¬Pyz ¬Pzw Pxw. Figure 5.13 shows the effect of inheriting these variants.

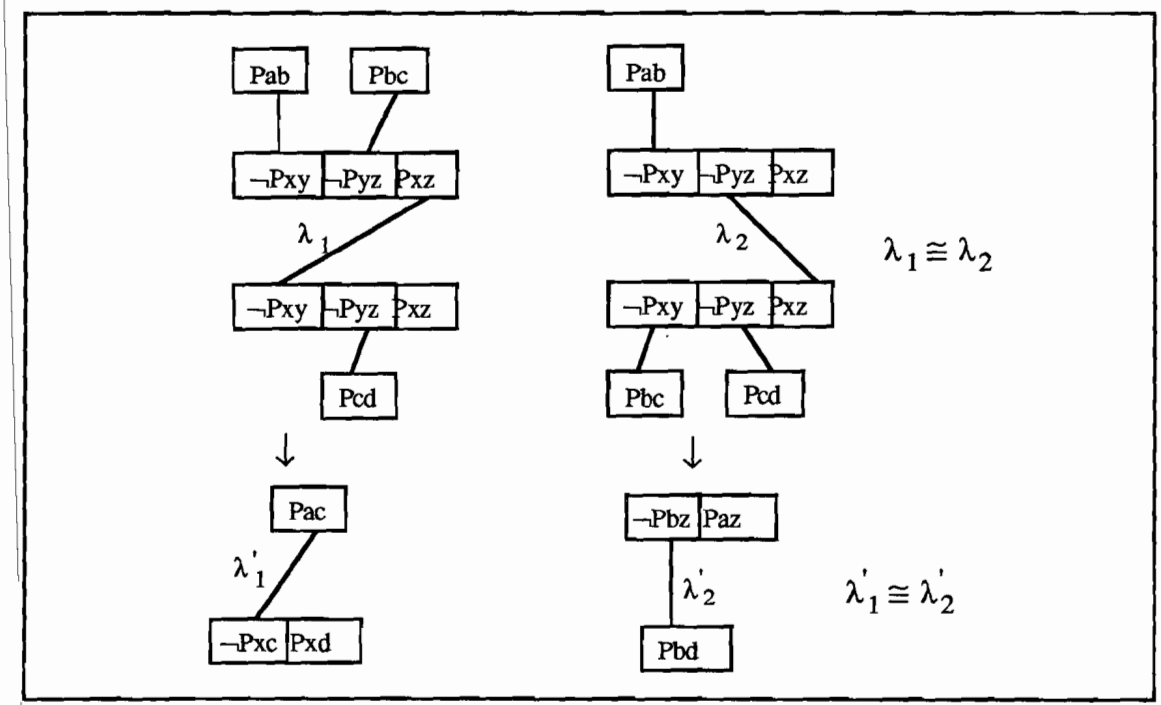


Fig. 5.13: Link Inheritance From the Transitivity Clause

A particular kind of link subsumption is symmetry. A clause D with distinct literals L_1, L_2 is called **symmetric (on L_1, L_2)**, iff there is a bijective mapping $\rho : \mathbb{V}(D) \to \mathbb{V}(D)$ with $L_1\rho = L_2$, $L_2\rho = L_1$ and $D\rho = D$.

5.3.6 Theorem:

Let D be a clause with literals L_1 and L_2. Then the following conditions are equivalent:

a) D is symmetric on L_1, L_2.

b) L_1 and L_2 have the same predicate symbol and the same sign and $\lambda_1 \cong \lambda_2$ holds for each pair (λ_1, λ_2) of adjacent R-links with $\lambda_i \in \mathbb{A}(L_i)$.

Proof: Suppose there is a renaming ρ satisfying the symmetry condition for D. Let C be a clause and $L \in C$ a literal, such that L and L_1 are unifiable. Since C and D are assumed to be variable disjoint, $\mathbb{V}(D) \cap \mathrm{dom}(\rho) = \emptyset$. Then L and L_2 are unifiable, and $\rho\sigma$ is a most general unifier of (L_2, L) for any most general unifier σ of (L_1, L). For i=1,2, let R_i be the resolvent of D with C on L_i. Then

$$R_2 = (D \setminus \{L_2\} \cup C \setminus \{L\})\rho\sigma = (D \setminus \{L_1\} \cup C \setminus \{L\})\sigma = R_1.$$

Conversely, suppose that b) is satisfied. Let $D = L_1 L_2 D'$. First we show that L_1 and L_2 are variants: W.l.o.g we assume that L_1 is a positive literal. Let $L_1 = Pt_1 \ldots t_n$. Since L_2 and L_1 have the same predicate symbol and the same sign, there are $s_1, \ldots, s_n$, such that $L_2 = Ps_1 \ldots s_n$. Let $C = \neg P x_1 \ldots x_n$, where $x_1, \ldots, x_n \notin \mathbb{V}(D)$. For i=1,2, let $\lambda_i \in \mathbb{A}(C) \cap \mathbb{A}(L_i)$, and let R_i be the resolvent along λ_i. Then $R_1 = L_2\sigma_1\, D'\sigma_1$ and $R_2 = L_1\sigma_2\, D'\sigma_2$, where $\sigma_i \in \Sigma^*(\lambda_i)$. Both σ_1 and σ_2 are renamings, thus $R_1 \cong L_2 D' = D \setminus L_1$ and $R_2 \cong L_1 D' = D \setminus L_2$ holds. If L_1 were not a variant of L_2, then also R_1 could not be a variant of R_2, which contradicts $\lambda_1 \cong \lambda_2$.

We write L_1 in a form $P'(x_1, \ldots, x_n)$, with $\{x_1, \ldots, x_n\} = \mathbb{V}(L_1)$. As L_1 and L_2 are variants of each other, L_2 can be written in the form $P'(y_1, \ldots, y_n)$.

Let C be a unit clause $\neg P'(c_1, \ldots, c_n)$ with $c_1, \ldots, c_n$ are constants not occurring in D. For i=1,2, let $\lambda_i \in \mathbb{A}(C) \cap \mathbb{A}(L_i)$ with $\sigma_i \in \Sigma^*(\lambda_i)$. Then $\sigma_1 = \{x_i \to c_i \mid i=1,\ldots,n\}$ and $\sigma_2 = \{y_i \to c_i \mid i=1,\ldots,n\}$. Define $\rho : \mathbb{V}(D) \to \mathbb{V}(D)$ by

$z\rho = z\sigma_1\sigma_2^{-1}$ if $z \in \mathbb{V}(L_1)$,

$z\rho = z\sigma_2\sigma_1^{-1}$ if $z \in \mathbb{V}(L_2)$, and

$z\rho = z$ otherwise.

First we have to show that ρ is well defined: Let $z \in \mathbb{V}(L_1) \cap \mathbb{V}(L_2)$. Then there are $x_i \in \mathbb{V}(L_1)$ and $y_j \in \mathbb{V}(L_2)$ with $z = x_i = y_j$. From the assumption follows that

the resolvent along λ_2, which is $L_1\sigma_2D'\sigma_2$, is a renaming of the resolvent along λ_1, which is $L_2\sigma_1D'\sigma_1$. Thus we have $L_2\sigma_1 \cong L_1\sigma_2$ and $D'\sigma_2 = D'\sigma_1$. As $x_i=y_j$, $x_i\sigma_2$ is the constant c_j, which implies $x_i\sigma_2 = y_i\sigma_1$. Moreover, we have $L_1\sigma_1=L_2\sigma_2=C$, which implies $x_i\sigma_1=y_i\sigma_2$. Hence we have

$x_i\sigma_2\sigma_1^{-1} = y_i\sigma_1\sigma_1^{-1} = y_i$ and

$x_i\sigma_1\sigma_2^{-1} = y_i\sigma_2\sigma_2^{-1} = y_i$.

Next we have to show that $L_1\rho = L_2$ and $L_2\rho = L_1$, and that $D'\rho = D'$. The latter follows from $D'\sigma_2 = D'\sigma_1$, and the first equations are obtained by $L_1\rho = L_1\sigma_1\sigma_2^{-1} = L_2\sigma_2\sigma_2^{-1} = L_2$, and $L_2\rho = L_2\sigma_2\sigma_1^{-1} = L_1\sigma_1\sigma_1^{-1} = L_1$. ■

5.3.7 Example:

The clause $C = Px\ Pyx$ is symmetric, since the renaming $\rho = \{x\rightarrow y, y\rightarrow x\}$ satisfies $(Pxy)\rho = Pyx$, $(Pyx)\rho = Pxy$ and $C\rho = C$. Resolving on the literal Pxy always yields the same result as resolving on the literal Pyx, that is, if λ_1 (λ_2) is a link joining the clause D with Pxy (Pyx) in C, then $\lambda_1 \cong \lambda_2$ holds.

A clause thus is symmetric, if it contains two literals, such that resolving on one of the literals always yields the same resolvent as resolving on the other literal. The condition that the two literals have the same predicate symbol and polarity, guarantees the existence of a clause resolving on both literals. Since symmetric clauses are a source of redundancy, such a characterization proves useful in order to recognize symmetric clauses and to restrict unneeded resolution steps.

Clause graph resolution provides an obvious means to cope with the kind of redundancy, which results from subsumed links. However, the context of this chapter is a resolution based reasoning system rather than a clause graph based system. In the following it will be shown how the results of section 5.2 and 5.3 can be used to reduce the derivation of redundant clauses, without referring to link inheritance and link deletion.

5.4 Removing Cycles and Subsumed Links

As cycles and subsumed links in clause sets are responsible for the generation of redundant information, a technique to remove such structures would prove very useful. As to cycles, a first approach to this question is due to Bibel (1981). He showed that under certain conditions, similar to those

allowing the deletion of tautologies in clause graphs, cycles can be removed from clause sets. In this section, a uniform approach to cope with both cycles and subsumed links, will be considered.

The method's basic idea is to preclude certain resolution steps by "compiling" the "critical structures" into a theory serving as the basis for theory resolution. A rather similar approach is pursued in equational reasoning, where certain equations, like that defining commutativity, are removed from the derivation and put into the unification procedure. Theory resolution, first proposed by Stickel (1985), is a generalization of ordinary resolution. The literals resolved upon need not be syntactically complementary, it is sufficient for them to be complementary under some theory. For instance, while the literals a<b, b<c, and c<a are not syntactically complementary, they are complementary under the theory of transitivity of the symbol <. Similarly, the literals Pab and ¬Pba are complementary under the theory of symmetry for the predicate P. For our purpose, the basic theory is always given by a set S of clauses, namely cyclic clause sets, or clauses containing subsumed links. The following example illustrates that this approach precludes the "critical" derivations.

5.4.1 Example:

Let S = {C_1,...,C_6}, with C_1=PQ, C_2=¬PR, C_3=RS, C_4=QS¬W, C_5=¬RW, C_6=¬SU (see figure 5.14).

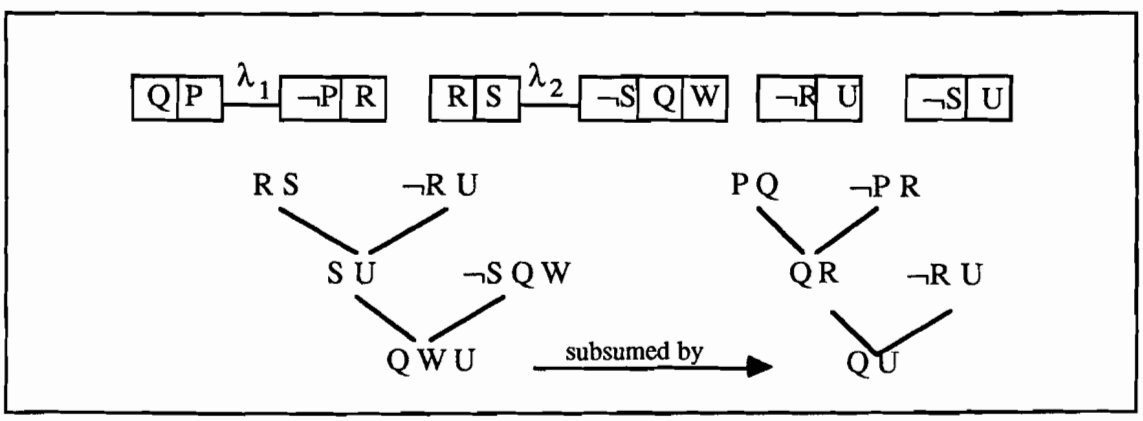


Fig. 5.14

The resolvent of C_3 and C_4 is subsumed by the resolvent of C_1 and C_2, that is, $\lambda_1 \leq \lambda_2$ holds for the appropriate links. Removing only the subsumed clause does not preclude further derivations of subsumed clauses by

inherited subsumed links. A case in point is the derivation of the clause QWU (see figure 5.14).

Such redundant derivations can be avoided by using the "critical clauses" $\{C_3, C_4\}$ for theory resolution. Obviously, a set of literals complementary under $\{C_3, C_4\}$ must contain the literals $\neg R$ and $\neg S$, or the literals $\neg Q$, S, and $\neg W$. This enables, for instance, a resolution step between C_5 and C_6 yielding the resolvent U (see figure 5.15). Resolution steps involving a clause of the theory are not allowed, hence no more derivations yielding subsumed clauses are possible. Figure 5.15 also illustrates why the particular kind of theory resolution, which is defined by clause sets, is called "link resolution" by Ohlbach & Siekmann (1988)[1]. The link between the literals $\neg R$ and $\neg S$ in the middle of the diagram can be seen as a *theory link*, under the theory generated by the clause RS.

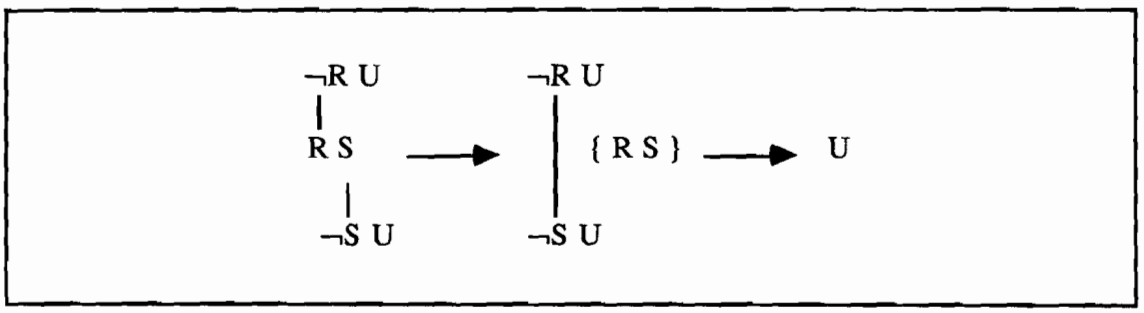


Fig. 5.15

In the following, we shall deal with so called *S-theories*, that are theories generated by a finite set S of clauses. If S is any set of clauses, then the *theory of S* is just the semantic closure $\langle S \rangle$ of S.

5.4.2 Definition:
Let S be a set of clauses, and let $C=\{L_1,\ldots,L_n\}\in\langle S\rangle$. Let $C_1,\ldots,C_n$ be clauses with $K_i\in C_i$ for $i\in\{1,\ldots,n\}$. If there is a most general simultaneous unifier σ of $\{(L_i,\neg K_i) \mid i=1,\ldots,n\}$, then the clause $\bigcup_{i=1}^{n} (C_i\setminus\{L_i\})\sigma$ is an **S-resolvent** of $C_1,\ldots,C_n$ (using C).

---

[1] The ancestor of the link resolution principle seems to be Wos' et al (1984) *linked inference* rules.

Note that if the clause C in the previous definition is an elementary tautology, then the S-resolution step is an ordinary resolution step. Since elementary tautologies are contained in each S-theory, the concept of S-resolution includes ordinary resolution. If the clause C is the empty clause, then the S-resolvent using C is also the empty clause.

The definitions of the derived notions, like S-resolution refutation, are straightforward. It is easy to see that the concept of S-resolvents is an instance of total narrow theory resolution, where $\langle S \rangle$ is the theory, and the L_i are the key literals[1]. Since theory resolution was shown to be correct and complete, this result implies soundness and completeness of S-resolution:

5.4.3 Corollary:
Let S be a clause set, and $S' \subseteq S$. Then S is unsatisfiable, iff $S \setminus S'$ admits an S'-resolution refutation.

In general, it is not decidable, whether a given clause C lies in the theory $\langle S \rangle$, even if S consists of a single clause only (Schmidt-Schauß 1986). However, it is not necessary to use all clauses in $\langle S \rangle$ for S-resolution. S-resolution remains complete, if only the *prime implicants* of $\langle S \rangle$, that is, the minimal clauses in $\langle S \rangle$ w.r.t. the subsumption order, are employed in the S-resolution steps[2]. Of course, even the set of prime implicants of $\langle S \rangle$ may be an infinite set.

5.4.4 Lemma:
Let S be a clause set, and let $D \in \langle S \rangle$. If there is a clause $C \in \langle S \rangle$ with $C \leq D$, then each S-resolvent using D is subsumed by an S-resolvent using C.

Proof: Obvious. ■

---

[1] As to the notions of key literals and total vs. partial, and narrow vs. wide theory resolution, see Stickel (1985).

[2] Note that the restriction to prime implicants corresponds to the requirement that the key literals of a theory resolution step have to be *minimally* unsatisfiable in the basic theory.

5.4.5 Definition:

Let S be a clause set. Then the set [S] is the set of prime implicants of ⟨S⟩, that is, the set of nontautologous clauses, which are minimal in ⟨S⟩ w.r.t. the subsumption order.

In contrast to the closed set ⟨S⟩, the set [S] of prime implicants does not contain any tautologies. In order to remain complete, when the used clauses are restricted to the prime implicants, the ordinary resolution steps, which are considered S-resolution steps using an elementary tautology, have to be explicitely admitted. Thus an [S]-resolution derivation (refutation) will be regarded as a derivation (refutation) consisting of S-resolution steps using clauses in [S] and of ordinary resolution steps.

5.4.6 Corollary:

Let S be a clause set, and $S' \subseteq S$. Then S is unsatisfiable, iff $S \setminus S'$ admits an [S']-resolution refutation.

For any clause set S, the set [S] of prime implicants can be generated in the usual way[1]: Form resolvents and remove subsumed clauses and tautologies.

For the class of complete semicycles, which produce only subsumed resolvents, the set of prime implicants is finite. This is easy to see: Suppose there is a resolution derivation from the set S of nodes of a complete semicycle G. W.l.o.g the derivation can be assumed to be minimal, that is, it produces only non-subsumed resolvents. If the set of prime implicants were infinite, then we had an infinite resolution derivation from the nodes of a complete semicycle G. This derivation would obviously involve a cyclic subset of G. According to the results of the previous section, such a derivation would produce a subsumed resolvent, which contradicts the assumption.

5.4.7 Examples:

a) Let S = {¬Pxy Qxy, ¬Quv Puv}. Together with the appropriate links, S forms an elementary cycle G, and [ℂ(G)] = S holds, since no non-tautological resolvents can be formed from S.

---

[1] As in chapter 3 of this thesis. Note, however, that chapter 3 deals only with ground clauses, where the set of prime implicants is always finite.

b) Let S = {¬Pxy Pyx}. Together with the internal link, S forms a function free, but not complete, cycle G. However, adding a copy of ¬Pxy Pyx yields the elementary cycle {¬Pxy Pyx, ¬Puv Pvu}. Thus [ℂ(G)] is finite and again [ℂ(G)] = S holds.
c) Let S = {¬Pf(f(x,y), z) Pf(x,f(y,z))}. Together with the internal link, S forms a non-complete cycle G. Then [ℂ(G)]= S ∪ {¬Pf(f(f(x,y),z),u) Pf(x,f(y,f(z,u))), ...} is infinite.

In section 5.2, it was shown that cyclic structures in clause sets account for ancestor subsumption. The strongest forms of ancestor subsumption are caused by complete semicycles, which correspond to finite theories. These can easily be employed for S-resolution, which thus in part overcomes the problem with the derivation of redundant clauses.

The situation is somewhat different, however, for the problem with subsumed links. The theory generated by clauses containing subsumed links, need not be finite, as the transitivity example shows:

5.4.8 Example:

Let T_3 be the transitivity clause $\neg Px_1x_2\ \neg Px_2x_3\ Px_1x_3$. T_3 possesses two internal links λ_1 and λ_2 with $\lambda_1 \equiv \lambda_2$. Resolving on λ_1 (or, equivalently, on λ_2) yields the clause $T_4 = \neg Px_1x_2\ \neg Px_2x_3\ \neg Px_3x_4\ Px_1x_4$. This clause again possesses internal links, and the process continues. Proceeding this way, we obtain the set

$$\mathcal{T} = \{\neg Px_1x_2\ \neg Px_2x_3 \ldots \neg Px_{n-1}x_n\ Px_1x_n \mid n \geq 3\}$$

This admits for instance the following sequence (figure 5.16):

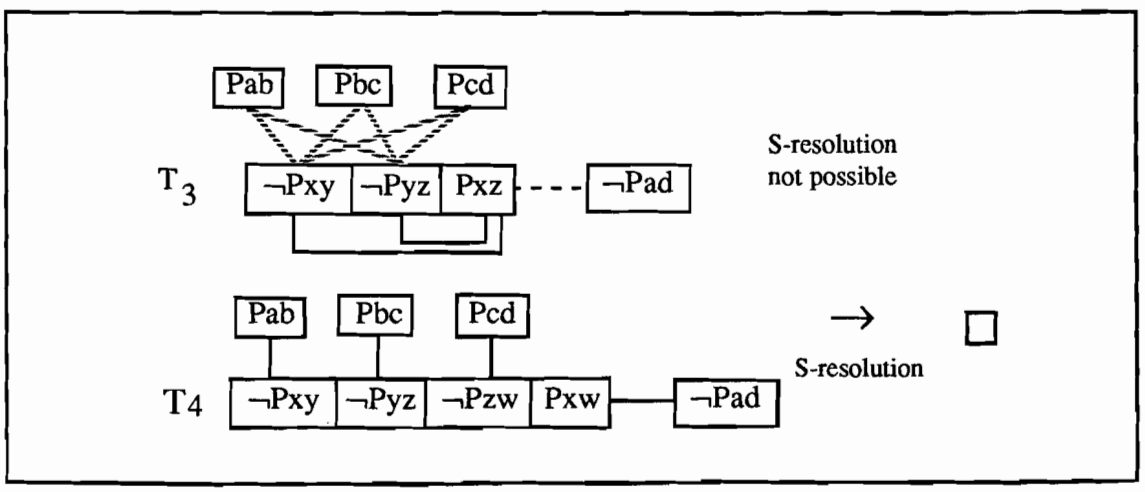


Fig. 5.16

Next we show by an example, how the information concerning cycles in clause graphs can be used to reduce the search space in resolution theorem proving. This example illustrates the application of S- resolution to problems containing cyclic structures.

5.4.9 Example

Translating SAM's lemma into a clause set, avoiding the use of the equality predicate (see Wos 1988), results in a clause set S that consists of the following 11 units

$\min(0\ x\ 0)$, $\max(x\ 0\ x)$, $\max(a\ b\ d_1)$, $\max(c_1\ e\ g)$, $\max(c_1\ f\ h)$, $\min(c_2\ b\ e)$
$\min(d_2\ c_2\ 0)$, $\min(c_2\ a\ f)$, $\min(d_1\ c_1\ 0)$, $\min(a\ b\ d_2)$, $\neg\min(g\ h\ c_1)$

and 6 non unit clauses:

(C_1) $\neg\min(x\ y\ z)\ \min(y\ x\ z)$

(C_2) $\neg\max(x\ y\ z)\ \max(y\ x\ z)$

(C_3) $\neg\min(x\ y\ u)\ \neg\min(y\ z\ v)\ \neg\min(x\ v\ w)\ \min(u\ z\ w)$

(C_4) $\neg\min(x\ y\ u)\ \neg\min(y\ z\ v)\ \neg\min(u\ z\ w)\ \min(x\ v\ w)$

(C_5) $\neg\max(x\ y\ z)\ \min(x\ z\ x)$

(C_6) $\neg\min(x\ z\ x)\ \neg\max(x\ y\ x_1)\ \neg\min(y\ z\ y_1)\ \neg\max(x\ y_1\ z_1)\ \min(z\ x_1\ z_1)$

In the following we describe a refutation of this clause set using positive hyperresolution together with theory resolution. The cyclic clauses C_1 and C_2, describing the symmetry of the min and max predicates, can be used for S-resolution, as in example 5.4.8.b). The same holds for the clause C_5, which is neither self-resolving nor a member of a cycle. The clauses C_3 and C_4, describing the associativity of the min predicate, form a cyclic structure, which produces ancestor subsumed clauses in the following way: Let D_1, D_2, D_3 be clauses that resolve with C_3 to the unit clause D_4. Then (D_1, D_2, D_4) resolves with C_4 to a copy of D_3. The resolution closure of the set $\{C_3, C_4\}$ is not finite, it thus cannot be used directly for theory resolution. But C_3, C_4 can be used to derive complete cycles in a way, as shown in figure 5.17:

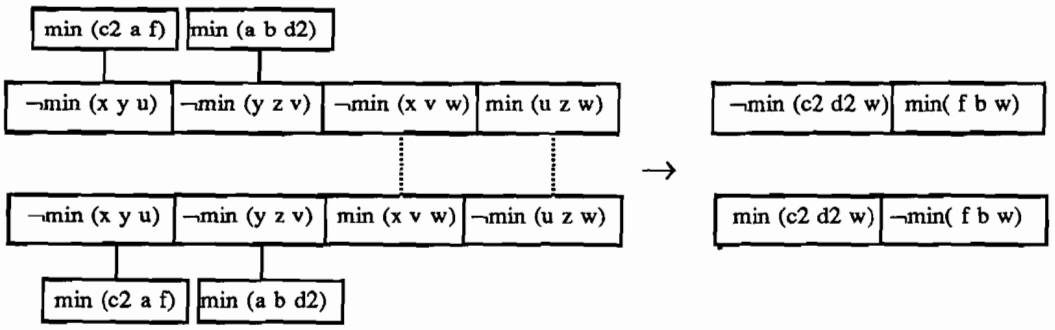


Fig. 5.17

Only the relevant links are shown in this figure. The dotted links denote the cyclic structure. The result of resolving $\{C_3,C_4\}$ with the two units is a complete cycle, which is added to the theory box. A resolution step of this particular kind will be abbreviated by the following diagram (figure 5.18), where a cycle is represented by an equivalence of the form A≈B.

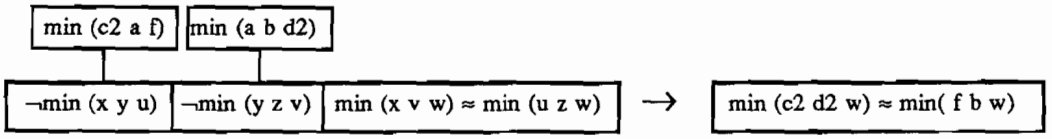
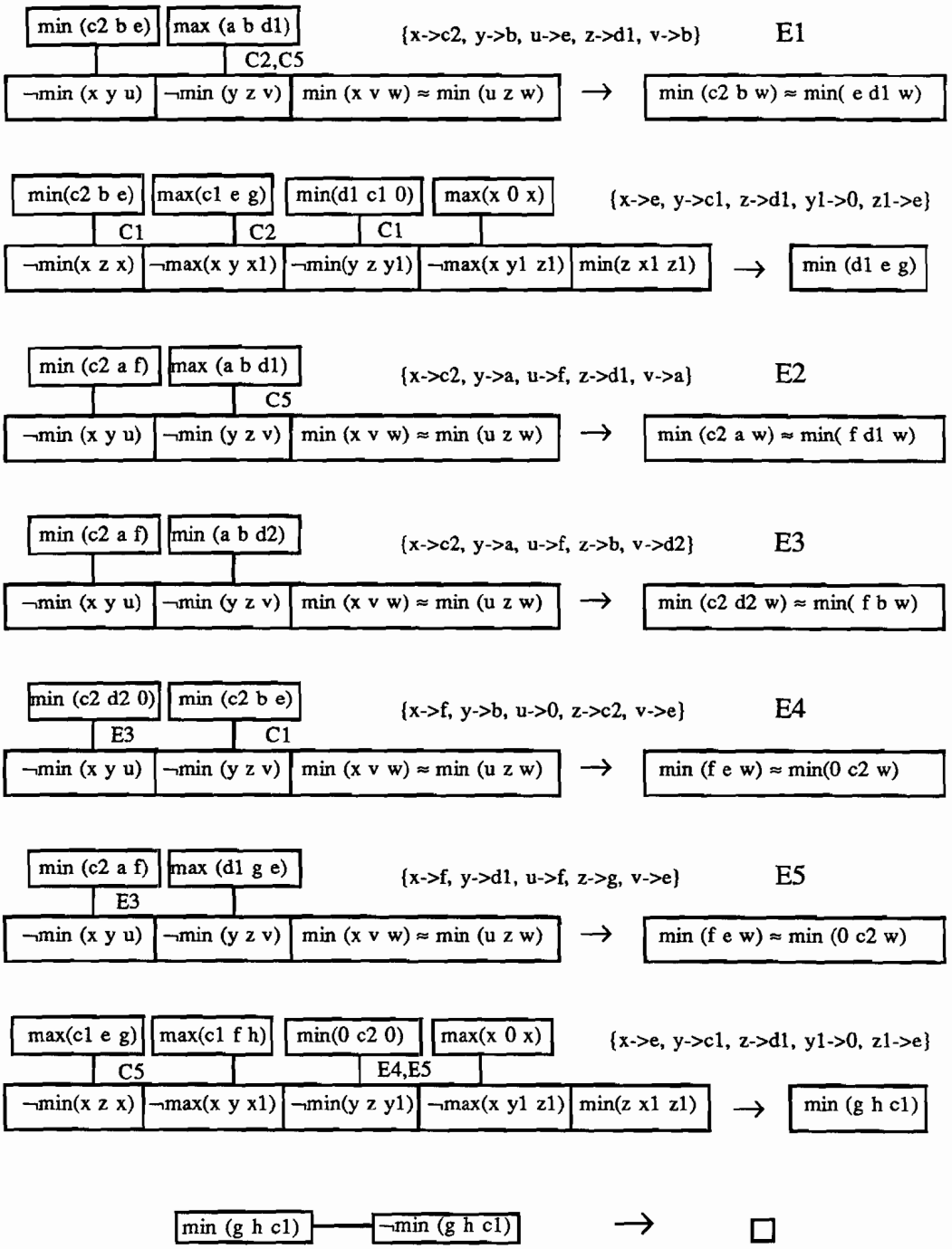


Fig. 5.18

Proceeding this way, only the clause C_6 is needed to produce "ordinary" resolvents. Of course, there remains the possibility to produce copies of already retained cycles. Taking into account these redundancies, a total 660 copies or instances of already present clauses are generated in the proof.

The proof of SAM's lemma, as illustrated below, consists of 8 steps. The bold faced links denote theory links, the used theories are denoted by their clauses. For instance, a link numbered with C_1 denotes a theory link under the theory of C_1, that is the commutativity of *min*.


min (c2 b e)
max (a b d1)
{x->c2, y->b, u->e, z->d1, v->b}
E1
C2,C5
¬min (x y u)
¬min (y z v)
min (x v w) ≈ min (u z w)
min (c2 b w) ≈ min(e d1 w)
min(c2 b e)
max(c1 e g)
min(d1 c1 0)
max(x 0 x)
{x->e, y->c1, z->d1, y1->0, z1->e}
C1
C2
C1
¬min(x z x)
¬max(x y x1)
¬min(y z y1)
¬max(x y1 z1)
min(z x1 z1)
min (d1 e g)
min (c2 a f)
max (a b d1)
{x->c2, y->a, u->f, z->d1, v->a}
E2
C5
¬min (x y u)
¬min (y z v)
min (x v w) ≈ min (u z w)
min (c2 a w) ≈ min(f d1 w)
min (c2 a f)
min (a b d2)
{x->c2, y->a, u->f, z->b, v->d2}
E3
¬min (x y u)
¬min (y z v)
min (x v w) ≈ min (u z w)
min (c2 d2 w) ≈ min(f b w)
min (c2 d2 0)
min (c2 b e)
{x->f, y->b, u->0, z->c2, v->e}
E4
E3
C1
¬min (x y u)
¬min (y z v)
min (x v w) ≈ min (u z w)
min (f e w) ≈ min(0 c2 w)
min (c2 a f)
max (d1 g e)
{x->f, y->d1, u->f, z->g, v->e}
E5
E3
¬min (x y u)
¬min (y z v)
min (x v w) ≈ min (u z w)
min (f e w) ≈ min (0 c2 w)
max(c1 e g)
max(c1 f h)
min(0 c2 0)
max(x 0 x)
{x->e, y->c1, z->d1, y1->0, z1->e}
C5
E4,E5
¬min(x z x)
¬max(x y x1)
¬min(y z y1)
¬max(x y1 z1)
min(z x1 z1)
min (g h c1)
min (g h c1)
¬min (g h c1)
□

6 Resolution with Equivalence

As in most languages, there are many ways to represent information to an automated reasoning program. One has, for instance, the choice between the predicative notation $Iyx \Rightarrow Pxye$ and the functional notation $P\,x\,inv(x)\,e$, when encoding that the product of x and its inverse is e. Similarly, the fact that z is the product of x and y can be expressed using a predicative notation Pxyz, or using an equality notation $fxy = z$. Usually the performance of a reasoning system strongly depends on the particular choice of notation, together with the appropriate choice of inference rule. For instance, if resolution is the chosen inference rule, while the problem is represented in the equality formulation, even simple problems become difficult to solve. Wos (1988) reports a proof of the commutator theorem of group theory, which was solved in 2 CPU seconds employing the inference rule paramodulation (Robinson & Wos 1969). The same problem took some 100 CPU seconds, when hyperresolution was used instead of paramodulation. In general, there is a number of advantages of using an equality oriented notation, together with the appropriate choice of the inference rule. It is not only that these proofs are shorter, in particular when demodulation (Wos 1967) is employed. Moreover, these proofs are often much more natural and easier to read. The performance of reasoning systems could thus be considerably increased, if the inference rules of paramodulation and demodulation could be made applicable also to problems that are represented in an equality-free notation. In this chapter it will be shown that the incorporation of logical equivalence into clause resolution is such a way to enhance the efficiency of automated reasoning systems.

With regard to the fact that a *cycle* of the form $\{\neg L_1L_2, \neg L_2L_3,...,\neg L_nL_1\}$ represents a set $\{L_1 \Rightarrow L_2, L_2 \Rightarrow L_3,..,L_n \Rightarrow L_1\}$ of implications, which expresses that the literals L_i are pairwise logically equivalent, the incorporation of logical equivalence thus also presents, besides the approach described in the previous chapter, another means to overcome the problems with the derivation of redundant clauses caused by cyclic structures in clause sets.

The approach described in this chapter is based on the correspondence between equivalence of literals and the equality of terms. This similarity finds expression in the fact that translating the functional expression

fxy = fuv into predicate notation yields the logical equivalence Pxyz ≡ Puvz. This can be taken as a hint that equality problems, which are denoted in an unsuited, predicative notation, admit a translation back into a better suited equivalence notation. Moreover, a resolution calculus that is extended by logical equivalence provides the power of paramodulation and demodulation also for problems that cannot be represented in an equality notation. For instance, the problem consisting of the axiom $\forall x\, Px \equiv \neg Pfx$ and the theorem $\exists x\, Px \,\&\, \neg Pfx$ (see Pelletier 1986) could be equally well handled with demodulation, even if it does not possess any corresponding equality formulation.

What would such a resolution calculus with equivalence look like? Consider for instance the clause set $S = \{C_1, C_2, C_3, C_4\}$, with $C_1 = PQ$, $C_2 = \neg P\neg Q$, $C_3 = P\neg Q$, and $C_4 = \neg PQ$. In a pure resolution calculus, resolving C_2 with C_3 is unnecessary, since the resolvents are tautologies, and likewise for C_1 and C_4. In an extended resolution calculus, however, there is an inference rule that derives the equivalence $P \equiv Q$ from C_2 and C_3, and the negated equivalence $\neg(P \equiv Q)$ from C_1 and C_4. These equivalences can be treated like ordinary clauses, that is, the empty clause can be derived from them. This example expresses the fact that a single equivalence $P \equiv Q$ or a negated equivalence can be treated like a single literal (or a unit clause). For instance, $P \equiv Q$ and $\neg(P \equiv Q)$ can be resolved to the empty clause, which is possible only for unit clauses. The equivalence $P \equiv Q$, however, might also be used as a demodulator[1] (Wos 1967) in the form $P \rightarrow Q$ or $Q \rightarrow P$. Directing this (ground) equivalence is arbitrary, one might employ for instance a well founded ordering on the predicate symbols. Thus one could "normalize" the actual clause set, after having deduced the rule $P \rightarrow Q$. Normalizing the parent clauses $\neg PQ$ and $P\neg Q$ yields the tautologies $\neg QQ$ and $Q\neg Q$, which can thus be removed. In fact, this is but another way to describe that $P \equiv Q$ *subsumes* its parent clauses $P \Rightarrow Q$ and $Q \Rightarrow P$. Note, however, that this description of subsumption applies only to equivalences that can be directed to rules. Normalizing the other clauses yields Q and $\neg Q$, and the next step

[1] A demodulator is the same as a rewrite rule. The notion of a demodulator, however, is more commonly used in the context of resolution theorem proving.

leads to the empty clause. This derivation has an analogon on the "resolution side", as figure 6.1 shows:

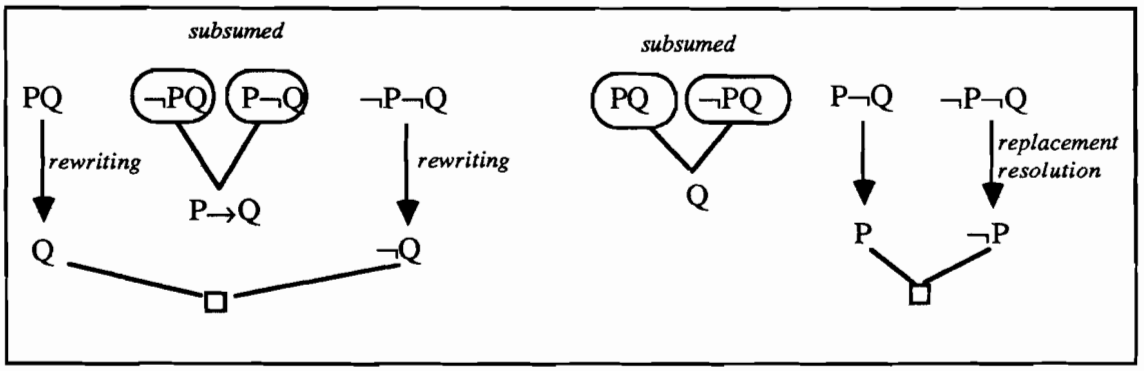


Fig. 6.1

Replacement resolution (Markgraf 1984), is a combined resolution and subsumption deletion step. In this example, resolving P¬Q and Q yields the unit clause P. This step is combined with the deletion of the subsumed parent clause P¬Q. In fact, this step can be considered a *reduction* step rather than a *deduction* step, since it derives no new clause, but instead removes a literal of the clause P¬Q. Replacement resolution can thus be considered a reduction, very similar to normalization by means of the rule P→Q.

One might argue that this example contains nothing really new. The derivations employing the equivalence clause are rather similar to the other ones and there seems to be no advantage in using them. In the following example (see figure 6.2), however, the rewriting approach surpasses the pure resolution refutation. The clause set in point is S = {¬PQ, P¬Q, ¬PR, P¬R, QR, ¬Q¬R}. While figure 6.2 shows a pure reduction refutation based on demodulators, no resolution reduction rule is applicable to the initial clause set.

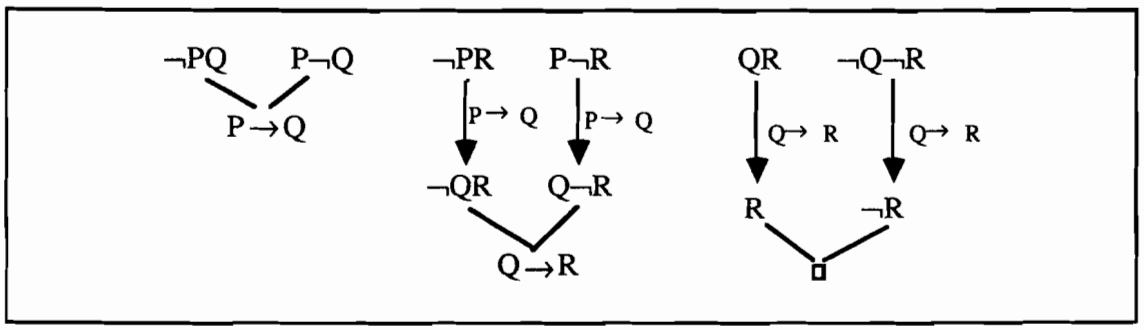


Fig. 6.2

While the previous example showed that literal demodulation enhances the reduction potential of resolution inference systems, another example demonstrates an effect of concentrating dispersed information. Refuting the set $S = \{C_1, C_2\}$ with $C_1 = \neg Pax \neg Pyb$ and $C_2 = PaxPyb$ requires either resolution with factoring or otherwise six binary resolution steps. Factoring, being necessary to guarantee completeness, is also a highly undesirable inference rule, since its unconstrained use can lead to a growing number of redundant clauses. (See Noll (1980) or Rabinov (1988)). Computing with equivalence clauses condenses the information about the necessary factoring step in the clause $\neg(Pax \equiv Pyb)$. Now only the instantiation $\{x \rightarrow b, y \rightarrow a\}$ is needed for the refutation.

It should, however, be mentioned that there is a certain tradeoff between supplying more power to reduction by the new rewrite rules and enlarging pure resolution's search space. New resolution possibilities (the ones resulting in equivalence literals) have to be considered besides the old ones, yet their successful application is not warranted. Restricting the unlimited derivation of equivalences could thus be appropriate. There are several choices for such a restriction. One could for instance derive equivalence clauses only if they subsume their parents (or at least one parent), such that this operation does not increase the size of the actual data base. We will adopt another restriction, which is based on the assumption that conditional rewriting (see for instance Kaplan (1984) or Zhang (1984)) is far more intricate than the unconditional version. In order to avoid a derivation like the one that computes the conditional equivalence $(P \equiv Q)RS$ from the two clauses $\neg PQR$ and $P \neg QS$, we will allow only unit (that is, unconditional) equivalence clauses. Recently, Zhang & Kapur (1988) have developed a first order calculus using conditional rewrite rules. According to their calculus, one literal of each clause is transformed into a conditional rewrite rule, where the condition consists of the remaining literals of the clause.

6.1 The Calculus

The resolution calculus with logical equivalence requires a slight extension of the syntax. In addition to the standard definitions of atomic formulae and

literals, given in chapter 2 of this thesis, we will deal with atomic formulae that consist of an equivalence.

6.1.1 Definition:

An atomic formula is either a **P-atom** $Pt_1...t_n$, with terms $t_1,...,t_n$ and a predicate symbol P of arity n, or it is an **E-atom** consisting of a pair (A,B) of P-atoms A and B. If the pair (A,B) is ordered, then the E-atom (A,B) is also called a **rule**, and written in the form $A \rightarrow B$. If it is unordered, then it is written in the form $A \equiv B$[1]. An **E-literal** is a literal, whose atom is an E-atom, and an **E-clause** is - due to the restriction mentioned above - a unit clause consisting of an E-literal.

The extensions of the semantic notions are straightforward:

6.1.2 Definition:

An interpretation $\mathfrak{I}$ **satisfies** a ground E-clause (L,K), iff either $\mathfrak{I}$ satisfies both L and K, or $\mathfrak{I}$ falsifies both L and K, and it satisfies an arbitrary E-clause E, iff it satisfies all its ground instances.

From now on a clause set will be understood as consisting of usual and/or E-clauses. The previous definition implies that the notions $\neg(A,B)$, $(\neg A,B)$ and $(A,\neg B)$ are equivalent. An E-literal that is used to paramodulate on some literal L, may thus be assumed in a form (K_1,K_2), where the literals L and K_1 have the same sign. This avoids an awkward case analysis in the definition of the derivation rules. For instance, the paramodulation step between the E-clause $P \equiv Q$ and the clause $\neg PR$, yielding the clause $\neg QR$, will be described with the modified E-literal $\neg P \equiv \neg Q$. Thus in the following E-literals will always be assumed to be in a form (L,K) with literals L and K. The set $\mathbb{A}(E)$ of atoms of the E-literal E = (L,K) is the set $\{\mathbb{A}(L),\mathbb{A}(K)\}$.

6.1.3 Definition:

For any E-clause E = (L,K), the **expanded form** E^* of E is defined as the clause set $\{\neg LK, L\neg K\}$.

---

[1] We shall, however, still write (A,B) for the E-atom to include both possibilities $A \equiv B$ and $A \rightarrow B$.

6.1.4 Corollary:

The interpretation $\mathfrak{I}$ satisfies the E-clause E, iff it satisfies E^*.

Proof: Let $E^*=\{C_1,C_2\}$. $\mathfrak{I}$ satisfies (L,K), iff $\mathfrak{I}$ satisfies each ground instance of $L\equiv K$, iff for each ground substitution θ, either $\mathfrak{I}$ satisfies both $L\theta$ and $K\theta$ or $\mathfrak{I}$ falsifies both $L\theta$ and $K\theta$, iff for each ground substitution θ, $\mathfrak{I}$ satisfies both $C_1\theta$ and $C_2\theta$, iff for each pair (θ_1,θ_2) of ground substitutions, $\mathfrak{I}$ satisfies both $C_1\theta_1$ and $C_2\theta_2$, iff $\mathfrak{I}$ satisfies both clauses C_1 and C_2. ■

6.1.5 Definition:

A **strong reduction ordering** is a well-founded[1] ordering $\sqsubset$ on terms and atoms, such that for all $s_i,t_i\in\mathbb{T}$, $s,t\in\mathbb{T}\cup\mathbb{A}$, $\sigma\in\Sigma$, $f\in\mathbb{F}_n\cup\mathbb{P}_n$:

a) if $s\sqsubset t$, then $s\sigma\sqsubset t\sigma$ ($\sqsubset$ is stable),

b) if $s_i\sqsubset t_i$, then $f(s_1,\ldots,s_i,\ldots,s_n)\sqsubset f(s_1,\ldots,t_i,\ldots,s_n)$ ($\sqsubset$ is monotonic),

c) $s\sqsubset t$ holds, if t contains s as a subterm ($\sqsubset$ has the subterm property),

d) $\sqsubset$ is total on ground terms.

For any objects a,b, we define $a\sqsubseteq b$, iff a=b, or $a\sqsubset b$. The ordering $\sqsubset$ can be extended to literals and to ground clauses in a canonical way:

6.1.6 Definition:

Let $\sqsubset$ be a strong reduction ordering on $\mathbb{T}\cup\mathbb{A}$. For all non-E-literals L,K, and for all (arbitrary) ground clauses C and D:

a) $L\sqsubset K$, iff $\mathbb{A}(L)\sqsubset\mathbb{A}(K)$

b) $C\sqsubset D$, iff $C\neq D$ and for each $L\in C$ there is a $K\in D$ with $L\sqsubseteq K$.

Note that this definition implies that E-clauses are incomparable by $\sqsubset$. From the definition it is clear that the strong reduction ordering $\sqsubset$ on ground clauses is stronger than the subsumption ordering and in particular, $\square\sqsubset C$ holds for arbitrary (E and non-E) ground clauses. In the sequel we shall assume a strong reduction ordering on the set of terms and atoms. Such an ordering allows to direct equivalences to rewrite rules. The E-clause (L,K) will be assumed to be ordered (that is, it is a rule $L\rightarrow K$), iff $K\sqsubset L$ holds.

Such an ordering could also be used to restrict the other inference rules like resolution by requiring that at least one of the literals involved must be

---

[1] An ordering $\sqsubset$ is well-founded, iff it admits no infinite descending chains $t_1\sqsupset t_2\sqsupset\ldots$

maximal in its clause w.r.t. the ordering. This proceeding results in a variant of ordered resolution, as it is described for instance by Rusinowitch (1987). The calculus will be formulated, however, using rewrite rules only for reduction.

An E-clause of the form A≡A is called *tautologous*. It is obvious that tautologous E-clauses represent tautologies and thus are redundant.

Besides the well-known resolution rule (with factoring), the calculus has four additional deduction rules.

6.1.7 Definition:

Let C be a clause, let E' be an E-clause, and let E be an E-clause of the form (L,K), with $\mathbb{V}(C)\cap\mathbb{V}(E)=\emptyset$, and $\mathbb{V}(E)\cap\mathbb{V}(E')=\emptyset$.

a) Let $L'\in C$ be unifiable with L with most general unifier σ. Then $(C\setminus\{L'\}\cup K)\sigma$ is an **ER-paramodulant** of C and E.[1]

b) Let E' = (L',K'), and let L' be unifiable with L with most general unifier σ. Then the E-clause $(K\sigma,K'\sigma)$ is an **ER-paramodulant** of E and E'.[2,3]

c) If there is a substitution μ with $L\mu = \neg K\mu$, then the empty clause $\square$ is an ER-paramodulant of E.

The following lemma establishes the relation between ER-paramodulation with an E-clause on the one hand and resolution with the clauses of the expanded form on the other hand.

6.1.8 Lemma:

Let E, E', and C be as in the previous definition.

a) Each ER-paramodulant of C and E is a resolvent of C with a clause in E^*.

b) Let E'' be an ER-paramodulant of E and E'. Let $D''\in (E'')^*$. Then D'' is resolvent of clauses D and D' with $D\in E^*$, $D'\in (E')^*$.

c) If the empty clause $\square$ is an ER-paramodulant of E, then $\square$ is a resolvent of the clauses in E^*.

---

[1] ER-Resolution stands for resolution with *equivalence and rewriting*.

[2] In the terminology of rewriting, rule a) would be called narrowing, rule b) superposition.

[3] Note that ER-paramodulation is ordered, if E is a rule.

Proof: Obvious. ■

6.1.9 Definition:

Let $C = L_1K_1$ and let $D = L_2K_2$ be variable disjoint clauses, and let σ be a simultaneous most general unifier of $(L_1, \neg L_2)$ and $(K_1, \neg K_2)$. Then the E-clause $(L_2\sigma, K_1\sigma)$ is an **ER-resolvent** of C and D.

From this definition follows that $\{C\sigma, D\sigma\}$ is the expanded form of the ER-resolvent of C and D.

An **ER-deduction** step is either a usual resolution step, or an ER-paramodulation step or an ER-resolution step. In addition to the common resolution reduction rules like tautology removal and subsumption deletion there are the following reduction rules.

6.1.10 Definition:

Let E and E' be E-clauses, and let C be a non-E-clause.

a) Then $E \leq C$ (E **subsumes** C) holds, iff $D \leq C$ holds for some $D \in E^*$.

b) Then $E \leq E'$ (E **subsumes** E') holds, iff there is a substitution μ with $E\mu = E'$.

6.1.11 Definition:

Let $R = L \rightarrow K$ be a rule, let E be an E-clause of the form (L',K') with $R \neq E$, and let $C = L'\ C'$ be a clause.

a) R **reduces** C to $K\mu\ C'$, if there is a substitution μ with $L\mu = L'$.

b) R **reduces** E to the E-clause $(K\mu, K')$, if there is a substitution μ with $L\mu = L'$.

A clause or rule C is called *irreducible* w.r.t. the set $\mathfrak{R}$ of rules, iff no rule of $\mathfrak{R}$ is applicable to C. An **ER-derivation** is a sequence $(S_1,\ldots,S_n)$ of irreducible clause sets, where S_{i+1} is obtained by reducing $S_i \cup \{R_i\}$, where R_i is an ER-resolvent or ER-paramodulant of clauses of S_i.

According to corollary 6.1.4 and lemma 6.1.8, it is clear that all the deduction and reduction rules of ER-resolution are sound. At the first glance, also the completeness of the calculus (without reductions) seems trivial, since ER-resolution is an extension of the resolution calculus, which itself is known to be complete. This argument, however, applies only for clause sets without E-clauses. For general clause sets, the completeness of the calculus has to be proved, and this is done traditionally in two steps. First complete-

ness for ground clauses is shown, then a lifting lemma transfers this result to the general case.

6.1.12 Lemma:
Let S be an unsatisfiable set of irreducible ground clauses. If $D \in S$ and $D \neq \square$, then there is an ER-derivation of a non-E clause C with $C \sqsubset D$ from S.

Proof: Let n be the number of atoms occurring in S. We proceed by induction on n. If n=1, then S must consist of two complementary unit clauses, which yields a resolution derivation of $\square$ and thus proves the assertion. Now let n>1.
Case 1: D is a non-E-clause. Let $L \in D$ and let $S(\neg L)$ be the set obtained from S by removing all non-E-clauses containing the literal $\neg L$, deleting the literal L from the remaining non-E-clauses, and replacing each E-clause of the form (L,K) by the unit clause $\{\neg K\}$. Then $S(\neg L)$ is also unsatisfiable,[1] and contains n-1 atoms. By the induction hypothesis, $S(\neg L)$ admits an ER-derivation $\mathcal{D}$ of a clause $C' \sqsubset D \setminus \{L\}$. From $\mathcal{D}$ we construct a derivation $\mathcal{D}'$ in the following way: we adjoin the literals L back to all clauses which were used in the derivation $\mathcal{D}$. Suppose, $\mathcal{D}$ contains a resolution step between C and $\neg K$ yielding the resolvent $C \setminus \{K\}$, where the clause $\neg K$ was generated from the E-clause (L,K). The ordering $\sqsubset$ is total on ground terms, hence either $L \sqsubset K$, or $K \sqsubset L$ holds. If $L \sqsubset K$, then this step is replaced by a reduction/paramodulation step between C and (K,L) yielding $C \setminus \{K\} \cup \{L\}$. If $K \sqsubset L$, then this resolution step is simply dropped. In the derivation $\mathcal{D}$, all literals but L are reduced after each derivation step. In the derivation $\mathcal{D}'$ we additionally reduce the literal L.
It is easy to see that $\mathcal{D}'$ results either in the clause $C=C'$, or in a clause $C=C' \cup \{K\}$ with $K \sqsubseteq L$. In both cases, $C \sqsubset D$ holds.
Case 2: D is an E-clause of the form (L,K). W.l.o.g. we assume that $K \sqsubset L$ holds. Let S' be the clause set obtained from S by removing D and replacing all occurrences of L ($\neg L$) by K ($\neg K$). Then S' is also unsatisfiable, and contains n-1 atoms. By induction hypothesis, there is an ER-derivation of the empty

[1] Compare the analogous construction in section 2.4 of this thesis; the argument that S(L) is also unsatisfiable is literally the same.

clause from S'. Since S' is obtained from S only by ER-reduction steps with the rule D, we have an ER-derivation of □ from S. ■

The completeness theorem for ground clauses now immediately follows from the previous lemma by an induction argument (note that the empty clause is a minimal element in the set of clauses w.r.t. ⊏).

6.1.13 Theorem:
Let S be an unsatisfiable set of ground clauses. Then there is an ER-derivation of the empty clause □ from S. ■

6.1.14 Lemma (Lifting lemma):
Let C' and D' be instances of (E-)clauses C and D, respectively, and assume R' is derived by an ER-derivation step from C' and D'. Then there is some clause R, which can be derived by the same type of ER-derivation step from C and D. Moreover, R' is an instance of R. ■

The proof of this lemma is analogous to the corresponding proof for resolution. The completeness theorem for ground ER-resolution together with the lifting lemma proves the completeness of the ER-calculus.

6.1.15 Theorem:
S is unsatisfiable, iff it admits an ER-derivation of the empty clause □. ■

We have shown that the resolution calculus allows for a sound and complete extension by equivalence literals. This extension provides part of the power of equality reasoning techniques also for resolution provers. In particular, it enhances the reduction part of resolution inference systems by literal demodulation. The following example shows the striking effect that this new reduction rule can have.

6.1.16 Example:
Let n be any even natural number and let S be the clause set $\{C_1, C_2\}$ with

$C_1 = Px\,Pfx,$

$C_2 = \neg Px\,\neg Pf^n x.$

Except for the rather trivial cases n=2 and n=4, this clause set is not easy to refute. For instance, the Markgraph Karl theorem prover (Markgraph 1984) failed for numbers n>10. However, the problem has a straightforward

solution based on literal demodulation: First, the following four resolvents of C_1 and C_2 are deduced:

$C_3 = \quad Pfx \; \neg Pf^nx$

$C_4 = \neg Px \quad Pf^{n-1}x$

$C_5 = \neg Px \quad Pf^{n+1}x$

$C_6 = \quad Px \; \neg Pf^{n+1}x$

Next, the E-clause E_1: $Px \equiv Pf^{n+1}x$ can be derived from C_5 and C_6. No matter, which strong reduction ordering is employed, this clause is always directed to the rule $R_1 = Pf^{n+1}x \rightarrow Px$. The rule R_1 rewrites its parent clauses C_5 and C_6 to tautologies, or, in other words, it subsumes them.

In the same way, the E-clause E_2: $Pf^nx \equiv Pfx$ with the corresponding rule R_2: $Pf^nx \rightarrow Pfx$ can be derived from C_3 and C_4. E_2 subsumes C_3, and R_2 rewrites C_2 to $C_2' = \neg Px \neg Pfx$. At this stage, we have the following set of clauses and rules

$C_1 \quad = \quad Px \quad Pfx,$

$C_2' \quad = \neg Px \quad \neg Pfx.$

$C_4 \quad = \neg Px \quad Pf^{n-1}x$

$R_1 \quad = \quad Pf^{n+1}x \rightarrow Px$

$R_2 \quad = \quad Pf^nx \rightarrow Pfx$

The clauses C_1, C_2', and C_4 are irreducible. The rule R_1, however, reduces with R_2 to $Pf^2x \equiv Px$, which can be directed to $R_1' = Pf^2x \rightarrow Px$. Then the repeated application of R_1' to R_2 yields finally - to be precise, after $n/2$ steps - the rule $Pfx \rightarrow Px$ (note that n is even!), which in turn reduces C_1 to Px and C_2 to $\neg Px$, and a last resolution step concludes the refutation.

The original clause set S with an odd number n is consistent - for instance, the natural numbers with P being the even-predicate, and f being the successor function is a model for this set. It is interesting to see that the same process as above results in the final set {R}, where R is the irreducible rule $Px \rightarrow \neg Pfx$. This proves the original set's consistency. Resolution theorem proving, however, fails already for n=1: The clause set $\{PxPfx, \neg Px\neg Pfx\}$ allows for an infinite derivation of clauses $\neg PxPf^{2k}x$ and $Px\neg Pf^{2k}x$ for all natural numbers k, and there is no way to detect that S is satisfiable. Ordered resolution (see Chang & Lee 1973), however, would perform analogously to ER-resolution in this case.

The following example illustrates the translation from a predicative into an equivalence notation that comes very close to the equality notation.

6.1.17 Example:

Consider the axiomatization for a group (G,*), avoiding where possible the use of equality (see Wos 1988). The equation $x*y = z$ is represented by the literal Pxyz, the term x*y by the term pxy.

The associativity axiom is represented by the two clauses

¬Pxyu ¬Pyzv ¬Puzw Pxvw

¬Pxyu ¬Pyzv ¬Pxvw Puzw

and the closure axiom by

P(x,y,pxy)

Resolving the closure axiom against the first two literals of each of the associativity clauses, one obtains

¬P(pxy,z,w) P(x,pyz,w) and

P(pxy,z,w) ¬P(x,pyz,w)

From these two clauses the E-clause $P(pxy,z,w) \equiv P(x,pyz,w)$ can be derived, which is a far more useful representation for the associativity axiom.

The role of reduction for resolution based systems is traditionally underestimated compared with that of deduction rules and particular strategies. Completion theorem provers like Hsiang (1982) and (1985), Kapur & Narendran (1985), Müller (1987), however, emphasize the role of reduction, which surely accounts for their increasing success. The transformation of clauses into rewrite rules, which is the basic feature of our approach, is also a common principle to completion theorem proving. Thus the resolution calculus extended by equivalence may be regarded as a partial incorporation of principles of completion theorem proving into resolution theorem proving.

7 Conclusion

The basic questions like completeness largely being solved, Automated Theorem Proving still poses intriguing problems, most of which come under the need for efficiency. One of those issues, the derivation of unneeded information[1], was the starting point for the second part of this thesis. Besides the development of an efficient subsumption test in chapter 4, our main interest was aimed at avoiding, or at least reducing, the derivation of those useless clauses, instead of removing them after their generation. As it is frequently the case, it turned out that the answer to this question was relevant also to some other basic problems of automated reasoning, showing yet again that most of these problems are deeply connected.

Among those basic research problems, which Wos (1988) ranks among the most important challenges for reasearch in automated theorem proving, the following questions turned out to be connected with the reduction of redundancy.

The first question[2] deals with issues of choosing the appropriate representation and inference rule. As already remarked (see chapter 6), for many problems one has the choice between an equality representation, permitting the use of the inference rule of paramodulation and other equality reasoning methods, and an equality free notation, which restricts the possible inferences to resolution and derived rules like hyperresolution. Usually, these two approaches exhibit a disparate performance. While some very small problems seem to favour the hyperresolution approach[3], most other problems are almost intractable when the inference rule hyperresolution is employed. It is natural to ask for the reason of this different behavior. It seems that the results of chapter 5 shed some light on the poor performance of hyperresolution in those cases. At a first glance, it seems that

---

1 Compare also problem 6 in Wos (1988).

2 Compare also problems 4 and 11 in Wos (1988).

3 For instance, the following problem admits a short and fast solution using hyperresolution: Given a group G, if $x^2=1$ holds for each $x \in G$, then G is commutative. Paramodulation, on the other hand, performs very poorly on this problem (see Bläsius (1987)).

equations are eliminated in the equality free formulation of those problems. On closer inspection, however, it turns out that equalities take the form of equivalences in all these examples. It seems that the absence of demodulators accounts largely for the poor performance of hyperresolution on these examples, and the results of chapter 5 and 6 also provide a means to employ hyperresolution with almost the same efficiency as paramodulation on these class of problems.

The second question reads as follows: "What is the appropriate theory for demodulating across argument and across literal boundaries - a theory similar to the current use of demodulation or similar to that for complete sets of reducions - to replace certain predicates by other predicates and certain collections of literals by other collections?" (Wos 1988). This question seems to be answered completely by the extension of resolution with equivalence as proposed in chapter 6.

A last problem concerns the question of how to choose the clauses a particular inference rule like hyperresolution, or linked inference rules, is applied to. Most of these rules rely exclusively on syntactic criteria, like literal polarity, or clause length. As far as linked inference is concerned, the results of chapter 5 suggest a particular choice of clauses as "nuclei" of linked inference rules. Those clauses that represent "hidden" redundancies like subsumed links, or complete cycles, should preferably be chosen to act as such nuclei, in order to avoid their being involved in resolution steps.

A first step towards a satisfying solution to these problems has been made in this thesis, however, many more questions remain unanswered. For instance, many unneeded clauses will still be derived, even if cyclic clause sets or subsumed links are removed. Which structures account for these redundancies, and how could they be used to avoid redundancies? Is there any reasonable means to *completely* avoid the derivation of redundant information? It might be suspected that subsumption is irrelevant for those approaches that entirely avoid the retention of new information like Stickel's (1988) Prolog technology theorem prover or the Matrix methods developed by Andrews (1981) and Bibel (1982). These methods thus seem to represent a means to sidestep the problem with the derivation of unneeded information. However, as Overbeek & Wos (1989) remark, such a position is flawed. In these approaches, subsumption comes under the form of identical

or subsumed subgoals to solve. For instance, a logic program might first make an attempt to solve the goal ¬Pxyz, and later as a subgoal, the clause ¬Pabc, and succesively numerous instances of the first subgoal. Or, in the presence of the *recursive* program clause ¬PxyPyx (in Prolog notation Pxy :- Pyx), the successive subgoals might be of the form Pab, Pba, Pab, and so on.[1]

The problem with the derivation of redundant information is among the most important problems for automated reasoning. It would thus be of great value to transfer some of the results of chapter 5 to other inference rules like paramodulation. For instance, an analogon to the inheritance of subsumed links (compare theorem 5.3.5 of this thesis) for paramodulation is given by the following conjecture: Suppose the paramodulant K of some literal L at subterm u with the equation $t_1=t_2$ is subsumed. Then for each descendant L' of L, which admits a paramodulant K' at the same subterm u with the equation $t_1=t_2$, K' is also subsumed. Of course, the notions of links and link inheritance for paramodulation have to be made precise in order to investigate this question.

[1] Compare the discussion in section 5.2.

References

Andrews, P.B. (1981). Theorem Proving via General Matings. *J. of the ACM*, **28**, 193 - 214.

Bachmair, L. & Dershowitz, N. (1987). Inference Rules for Rewrite-Based First Order Theorem Proving. In: *Proc. of 2nd Conference on Logic in Computer Science.*

Berztiss, A.T. (1973). A Backtrack Procedure for Isomorphism of Directed Graphs. *Journal of the ACM* , **20/3**, 365-377.

Bibel, W. (1981). On Matrices with connections. *Journal of the ACM* **28/4**, 633 - 645.

Bibel, W. (1982). Automated Theorem Proving. Braunschweig. Vieweg.

Bibel, W. (1987). Advanced Topics in Automated Deduction. Technical Report 87-39. University of Vancouver.

Bläsius, K. H. (1987). Equality Reasoning Based on Graphs. SEKI-Report SR-87-01, University of Kaiserslautern.

Bürckert, H.-J., Herold, A. & Schmidt-Schauß, M. (1987). On Equational Theories, Unification and Decidability. In: Lescanne, P. (ed): *Proc. of 2nd Conference on Rewriting Techniques and Applications*, Bordeaux, France. Springer LNCS 256, 204 - 215.

Chang, C.L. & Lee, R.C. (1973). Symbolic Logic and Mechanical Theorem Proving. Academic Press. New York.

Eisinger, N. & Weigele, M. (1983). A Technical Note on Splitting and Clausal Normal Form Algorithms. In: *Proc. of 7th German Workshop on Artificial Intelligence, Dassel/Solling*. Springer IFB 76, 225 - 231.

Eisinger, N. (1981). Subsumption and Connection Graphs. In: A. Drinan (Ed.): *Proc. of the 7th Int. Joint Conf. on AI*, Vancouver, 480-486.

Eisinger, N. (1988). Completeness, Confluence, and Related Properties of Clause Graph Resolution. PhD thesis and SEKI-Report SR-88-07, Universität Kaiserslautern.

Gallier, J.H. (1986). Logic for Computer Science. New York. Harper & Row.

Garey, M.R. & Johnson, D.S. (1979). Computers and Intractability. Freeman, San Francisco.

Gottlob, G. & Leitsch, A. (1985). On the Efficiency of Subsumption Algorithms. *Journal of the ACM*, **32/2**, 280-295.

Guard, J. et al (1969). Semi-Automated Mathematics. *Journal of the ACM*. **16**, 49 - 62.

Henschen, L. et al. (1980). *Challenge Problem 1*. SIGART Newsletter 72, 30 - 31.

Herold, A. & Siekmann, J. (1985). Unification in Abelian Semigroups. Memo-SEKI-85-III. Universität Kaiserslautern.

Herold, A. (1983). Some Basic Notions of First Order Unification Theory. Internal Report, Universität Kaiserslautern.

Hsiang, J. (1982). Topics in Automated Theorem Proving and Program Generation. Ph.D. Thesis, Dep. of Comp. Sc. , University of Illinois at Urbana-Champaign.

Hsiang, J. (1985). Refutational Theorem Proving using Term-rewriting Systems. *Artificial Intelligence* **25,** 255 - 300.

Huet, G. (1976). Résolution d'équations dans des langages d'ordre 1,2,...,ω. Thèse d'Etat, Université de Paris VII.

Hullot, J.M. (1980). A Catalogue of Canonical Term Rewriting Systems. Report CSL-113, SRI International, Menlo Park.

Joiner, W. (1973). Automatic Theorem-Proving and the Decision Problem. Report 7/73Center Research Comp. Tech., Harvard University.

Kaplan, S. (1984). Fair Conditional Term Rewriting Systems: Unification, Termination, and Confluence. LRI, Orsay.

Kapur, D. & Narendran, P. (1985). An Equational Approach to Theorem Proving in First-Order Predicate Calculus. 84CRD322, General Electric Corp. Research and Development Report, Schenectady, N.Y.

Kapur, D. & Narendran, P. (1986). NP-Completeness of the Set Unification and Matching Problems. In: J. H. Siekmann (Ed.): *Proc. 8th International Conference on Automated Deduction*, Oxford. Springer LNCS 230, 489 - 495.

Kowalski, R. (1970). The Case for Using Equality Axioms in Automated Demonstration. Symp. on Automated Demonstration. Springer LNM, 125, 181 - 201.

Kowalski, R. (1975). A Proof Procedure Using Connection Graphs. *Journal of the ACM*, **22/4**, 572 - 595.

Lewis, H.R. & Papadimitriou, C.H. (1981). Elements of the Theory of Computation. Englewood Cliffs, Prentice Hall.

Loveland, D.W. & Shostak, R.E. (1980). Simplifying Interpreted Formulas. In: W. Bibel & R. Kowalski (Eds.): *Proc. of 5th Conference on Automated Deduction, Les Arcs, France.* Springer LNCS 87, 97 - 109.

Loveland, D.W. (1970). A Linear Format for Resolution. In: *Proc. IRIA Symp. on Automatic Demonstration.* Versailles, France. Springer Lecture Notes in Math. 125, 147 - 162.

Loveland, D.W. (1978). Automated Theorem Proving: A Logical Basis. North-Holland.

Markgraph, K. (1984). The Markgraph Karl Refutation Procedure. SEKI Memo MK-84-01, Universität Kaiserslautern.

Martelli, A. & Montanari, U. (1982). An Efficient Unification Algorithm. ACM TOPLAS, 4/2, 258 - 282.

McCharen, J. Overbeek, R. & Wos, L. (1976). Complexity and related enhancements for automated theorem-proving programs. *Computers and Mathematics with Applications* **2**, 1 - 16.

Müller, J. & Socher, R. (1988). Topics in Completion Theorem Proving. SEKI-Report SR-88-13, Universität Kaiserslautern.

Müller, J. (1987). Theopogles - A Theorem Prover Based on First-Order Polynomials and a Special Knuth-Bendix Procedure. In: K. Morik (Ed.): Proc. of the 11the German Workshop on Artificial Intelligence, Geseke. Springer IFB 152, 241 - 250.

Müller, J. (1988). Theorembeweisen mit Rewritetechniken. Dissertation. Universität Kaiserslautern.

Murray, N.V. & Rosenthal, E. (1987). Inference with Path Resolution and Semantic Graphs. *Journal of the ACM*, **34/2**, 225 - 254.

Nicolaita, D. (1989). Structure Sharing with Literal Indexing Mechanisms for Resolution Systems. Unpublished manuscript. University of Bucharest.

Noll, H. (1980). A Note on Resolution: How to Get Rid of Factoring Without Loosing Completeness. In: W. Bibel & R. Kowalski (Eds): Proc. of 5th Conf. on Automated Deduction. Springer LNCS 87, 50 - 63.

Ohlbach, H.J. & Siekmann, J. (1988). Using Automated Reasoning Techniques for Deductive Databases. SEKI-Report SR-88-06, Universität Kaiserslautern.

Overbeek, R. & Wos, L. (1989). Subsumption, a Sometimes Undervalued Procedure. Technical report MCS-P93-0789, Argonne National Laboratory, Argonne, IL.

Overbeek, R. (1975). An Implementation of Hyper-Resolution. *Computational Mathematics with Applications* **1**, 201 - 214.

Pelletier, F.J. (1986). Seventy-five Problems for Testing Automatic Theorem Provers. Journal of Automated Reasoning. **2/2**, 191 - 216.

Quine, W.V. (1952). The Problem of Simplifying Truth Functions. *American Math. Monthly*, **59**, 521 - 531.

Quine, W.V. (1959). On Cores and Prime Implicants of Truth Functions. *Am. Math. Monthly*, **66**, 755 - 760.

Rabinov, A. (1988). A Restriction of Factoring in Binary Resolution. In: E. Lusk & R. Overbeek (Eds.): Proceedings of the 9th International Conference on Automated Deduction. Argonne, Illinois, U.S.A. Springer LNCS 310, 582 - 591.

Robinson, G. & Wos, L. (1969). Paramodulation and theorem-proving in first-order theories with equality. In: B. Meltzer, & D. Michie, (Eds.): Machine Intelligence 4. Edinburgh, 135 - 150.

Robinson, J.A. (1965a). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, **12/1**, 23 - 41.

Robinson, J.A. (1965b). Automated Deduction with Hyper-Resolution. *Intern. Journal of Comp. Mathematics*, **1**, 227 - 234.

Rusinowitch, M. (1987). Démonstration automatique par des techniques de réécriture. Thèse de doctorat d'etat. Nancy.

Schmidt-Schauß, M. (1986). Some Undecidable Classes of Clause Sets. Interner Bericht, SEKI-Report 86-08, Universität Kaiserslautern.

Shostak, R.E. (1976). Refutation Graphs. *Artificial Intelligence*. **7/1**, 51 - 64.

Shostak, R.E. (1979). A Graph-Theoretic View of Resolution Theorem-Proving. Report SRI International, Menlo Park, CA.

Slagle, J.R.; Chang, C.L. & Lee, R.C.T. (1970). A New Algorithm for Generating Prime Implicants. *IEEE Trans. on Comp.* **19/4**, 304 - 310.

Socher, R. (1987a). Boolean Simplification of First Order Formulae. SEKI-Report SR-87-4, Universität Kaiserslautern.

Socher, R. (1987b). Graph Isomorphism: Some Special Cases. SEKI-Report SR-87-10, Universität Kaiserslautern.

Socher, R. (1987c). Optimizing the Clausal Normal Form Transformation. To appear in: *Journal of Automated Reasoning.*

Socher, R. (1988a). A Subsumption Algorithm Based on Characteristic Matrices. In: E. Lusk; R. Overbeek (Eds.): *Proc. 9th International Conference on Automated Deduction,* Argonne. Springer LNCS 310, 573 - 581.

Socher, R. (1990). On the Relation Between Resolution Based and Completion Based Theorem Proving. To Appear in: *Journal of Symbolic Computation,* special issue on Theorem Proving using rewrite techniques.

Socher-Ambrosius, R. & Müller, J. (1989). A Resolution Calculus Extended by Equivalence. In: D. Metzing (Ed.): *Proc. of 13th German Workshop on Artificial Intelligence,* Eringerfeld. Springer IFB 216, 102 - 106.

Socher-Ambrosius, R. (1989a). Detecting Redundancy caused by Congruent Links in Clause Graphs. In: D. Metzing (Ed.): *Proc. of 13th German Workshop on Artificial Intelligence,* Eringerfeld. Springer IFB 216, 74 - 82.

Socher-Ambrosius, R. (1989b). Reducing the Derivation of Redundant Clauses in Reasoning Systems. In: N. Sridharan (Ed.): *Proc. of the 11th Int. Joint Conf. on AI,* Detroit. Morgan Kaufman Publ., 401 - 406.

Socher-Ambrosius, R. (1989c). Another Technique for Proving Completeness of Resolution. SEKI-Report SR-89-05, Universität Kaiserslautern.

Socher-Ambrosius, R. (1990). Boolean Algebra Admits no Canonical Term Rewriting System. SEKI-Report, Universität Kaiserslautern, to appear.

Stickel, M. E. (1985). Automated Deduction by Theory Resolution. *Journal of Automated Reasoning.* **1/4**, 333 - 356.

Stickel, M. E. (1986). Schubert's steamroller problem: Formulations and Solutions. *Journal of Automated Reasoning.* **2/1**, 89 - 102.

Stillman, R.B. (1973). The Concept of Weak Substitution in Theorem Proving. *Journal of the ACM,* **20/4**, 648-667.

Tison, P. (1967). Generalized Consensus Theory and Application to the Minimization of Boolean Functions. *IEEE Trans. on Comp.* **16/4**, 446 - 456.

Unger, S.H. (1964). GIT - A Heuristic Program for Testing Pairs of Directed Line Graphs for Isomorphism. *Comm. of the ACM* **7**, 26 - 34.

Vieille, L. (1987). Recursive Query Processing: The Power of Logic. ECRC Munich, Technical Report TR-KB-17.

Walther, C. (1981). Elimination of Redundant Links in Extended Connection Graphs. In: J.H. Siekmann (Ed.): *Proc. of German Workshop on Artificial Intelligence.* Bad Honnef. Springer IFB 47, 201 - 213.

Wos, L. (1988). Automated Reasoning: 33 Basic Research Problems. Prentice Hall, Englewood Cliffs.

Wos, L.; Robinson, G.; Carson,D. & Shalla, L. (1967). The Concept of Demodulation in Theorem Proving. *Journal of the ACM,* **14**, 698 - 709.

Wos, L.; Veroff, R.; Smith, B. & McCune, W. (1984a). The Linked Inference Principle, II: The User's Viewpoint. In: R.E. Shostak (Ed.): *Proceedings of the 7th Int. Conference on Automated Deduction*. Springer LNCS 170, 316 - 332.

Wos, L.; Winker, S.; Veroff, R.; Smith, B. & Henschen, L. (1984b). A New Use of an Automated Reasoning Assistant: Open Questions in Equivalential Calculus and the Study of Infinite Domains. *Artificial Intelligence*, **22**, 303 - 356.

Zhang, H. & Kapur, D. (1988). First-Order Theorem Proving Using Conditional Rewrite Rules. In: E. Lusk & R. Overbeek (Eds.): Proceedings of the 9th International Conference on Automated Deduction. Argonne, Illinois, U.S.A. Springer LNCS 310, 1 - 20.

Zhang, H. (1984). Reveur 4: Etude et Mise en Œuvre de la Réécriture Conditionelle. Thèse de Doctorat de 3ème cycle. Université de Nancy I.

Appendix: Boolean Algebra Admits no Canonical Term Rewriting System

In the following a formal proof will be given that a canonical term rewiting system for Boolean algebra cannot exist. First we introduce the basic notions of equational term rewriting systems.

A.1 Definition (Equational System):

An **equational** system E is a set of termpairs s=t. This system generates an equality relation $=_E$ in the following way: We define a relation $=_E^1$ by $s =_E^1 t$, iff there exists an occurrence u in s, an equation s'=t' or t'=s' in E, and a substitution σ, such that $s/u = s'\sigma$ and $t = s[u \rightarrow t'\sigma]$. The relation $=_E$ is defined as the transitive, reflexive closure of $=_E^1$. It is clear that $=_E$ is an equivalence relation. The equivalence class of t modulo E will be denoted as $[t]_E$.

A.2 Definition (Equational Term Rewriting System):

A **term rewriting system** R (over $\mathbb{T}$) is a set of termpairs $l \rightarrow r$ (the so called **rules**), such that $\mathbb{V}(r) \subseteq \mathbb{V}(l)$ (and $l,r \in \mathbb{T}$). A term t_1 **R-reduces** to a term t_2, written $t_1 \Rightarrow_R t_2$, iff there exists an occurrence u in t_1, a rule $l \rightarrow r$ in R, and a substitution σ, such that $t_1/u = l\sigma$ and $t_2 = t_1[u \rightarrow r\sigma]$.

A term t_1 **E,R-reduces** to t_2, written $t_1 \Rightarrow_{E,R} t_2$, iff there exist $t'_1 \in [t_1]$, $t'_2 \in [t_2]$ with $t'_1 \Rightarrow_R t'_2$.

$\overset{+}{\Rightarrow}_{E,R}$ denotes the transitive, $\overset{*}{\Rightarrow}_{E,R}$ denotes the reflexive transitive closure of $\Rightarrow_{E,R}$ and $=_{E,R}$ denotes the reflexive, symmetric, and transitive closure of $\Rightarrow_{E,R}$.

The pair (E,R) is called an equational term rewriting system (ETRS). It can be understood also as a rewriting system for $\mathbb{T}/=_E = \{[t] \mid t \in \mathbb{T}\}$.

(E,R) is **noetherian**, iff there is no infinite sequence of E,R-reductions from any term.

(E,R) is **confluent**, iff $t \overset{*}{\Rightarrow}_{E,R} t_1$ and $t \overset{*}{\Rightarrow}_{E,R} t_2$ implies the existence of a term t_3 with $t_1 \overset{*}{\Rightarrow}_{E,R} t_3$ and $t_2 \overset{*}{\Rightarrow}_{E,R} t_3$.

A noetherian and confluent system is called **convergent**.

A term t_1 is called **(E,R-)irreducible**, iff there is no term t_2 with $t_1 \Rightarrow_{E,R} t_2$, and (E,R-)reducible otherwise.

An irreducible term t is called a **normal form** for t_1, iff $t_1 \Rightarrow^*_{E,R} t$.
If (E,R) is convergent, then each term t has a normal form $t\downarrow$, and $s\downarrow =_E t\downarrow$ holds for each term s with $s =_{E,R} t$.

A.3 Definition:

Let $\mathfrak{R}$ be a convergent ETRS over $\mathbb{T}$. Then the noetherian partial ordering $>_{\mathfrak{R}}$ on $\mathbb{T}$ generated by $\mathfrak{R}$ is defined by $s>t$ iff $s\Rightarrow^+_{\mathfrak{R}} t$. In the following we shall usually drop the index $\mathfrak{R}$.

A.4 Lemma:

Let $\mathfrak{R}$ be a convergent system on $\mathbb{T}$ with $=_{\mathfrak{R}} = =_E$.

a) The ordering > generated by $\mathfrak{R}$ is compatible with substitutions, that is, $s>t$ implies $s\sigma>t\sigma$ for any $s,t\in\mathbb{T}$ and any substitution σ.

b) Let $s,t\in\mathbb{T}$. If $s =_E t$ and t is $\mathfrak{R}$-irreducible, then $s>t$ holds.

Proof: Obvious. ■

In the following let AC be the equational system
$AC = \{x\vee y = y\vee x, x\wedge y = y\wedge x, x\vee(y\vee z) = (x\vee y)\vee z, x\wedge(y\wedge z) = (x\wedge y)\wedge z\}$.
and ACD the system $AC \cup \{x\vee(y\wedge z) = (x\vee y)\wedge(x\vee z), x\wedge(y\vee z) = (x\wedge y)\vee(x\wedge z)\}$.

In the following we shall consider exclusively the term set $\mathbb{T}=\mathbb{T}(F_B,\mathbb{V})$, where F_B is the signature $(\wedge,\vee,\neg)$ of boolean algebra.

For ease of notation, we shall use the following convention: For any $t\in\mathbb{T}$, we define the dual term $\bar{t}$, which is obtained from t by simultaneously replacing each occurrence of $\vee$ by $\wedge$ and vice versa, and each occurrence of 0 by 1, and vice versa.

In the following equality will tacitly be understood to be equality modulo AC. As it is done in chapter 3, equality modulo BA will be denoted by $\cong$, and terms which are equal under BA, will also be called *equivalent*. We will use the customary notion of *literals*, *clauses* and a *conjunctive normal form* (CNF). Recalling the notions of chapter 3, a term t is called a literal, iff it is either of the form a, or of the form $\neg a$, with a being a constant or a variable. The term t is a *clause*, if $t = s_1\vee\ldots\vee s_n$, with pairwise distinct literals s_i. A term t is called a *CNF-term*, if $t = s_1\wedge\ldots\wedge s_n$, where the s_i are pairwise distinct clauses. A term with topsymbol $\vee$ is also called a *disjunction*, a term with topsymbol $\wedge$ a *conjunction*, and a term with topsymbol $\neg$ a *negation*.

A.5 Theorem:

There exists no convergent ETRS (AC,R) such that $=_{AC,R}$ coincides with $\cong$.

Note that we deal exclusively with term rewriting systems over the fixed signature F_B. There exists, for instance, a convergent system over the extended signature $(\wedge,\vee,\neg,+,*,0,1)$, see Hsiang (1985).

In order to prove the theorem above, we first provide some lemmata. For the remainder of this section, we shall assume that there exists a convergent system $\Re = (AC,R)$ for BA. Let $>$ be the noetherian ordering associated with $\Re$.

A.6 Lemma:

The following relations hold:

$(x\vee y)\wedge y > y$

$\neg x\vee x > 1$

$x\vee x > x$

$x\vee 0 > x$

$x\vee 1 > 1$

$\neg\neg x > x$

$(x\vee y)\wedge(\neg x\vee y) > y$

Proof: For each line, the two terms are equivalent according to definition 3.1.1 and lemma 3.1.2. Furthermore, each right hand side is obviously irreducible, hence the assertion follows from lemma A.4.b. ■

The proof of our main theorem proceeds essentially by considering a particular term t, and proving that all terms $t'\cong t$ are reducible. The following lemmata will provide two important techniques to prove a term t reducible, which are used heavily in the sequel. The first states that the normal form of a symmetric term must be symmetric.

If t is a term containing the (distinct) symbols p,q, and $t(p,q) = t(q,p)$, then the term t is called *symmetric* in (p,q). t is called *semi-symmetric* in (p,q), iff $t(p,q) \cong t(q,p)$.

A.7 Lemma (Symmetry Lemma):

Let $x,y \in \mathbb{V}$ with $x \neq y$, and let $t=t(x,y)$ be irreducible. If t is semi-symmetric in (x,y), then t is even symmetric in (x,y).

Proof: Assume $t(x,y) \neq t(y,x)$. Then we have $t(x,y) > t(y,x)$, since the latter is irreducible. But then, according to A.4.a also $t(x,y)\sigma > t(y,x)\sigma$ for $\sigma=\{x \to y;\ y \to x\}$, which implies $t(y,x) > t(x,y)$, a contradiction. ■

The symmetry lemma can also be stated as follows: If the term t is symmetric in (x,y), then $t\downarrow$ is also symmetric in (x,y).

The next "subterm lemma" shows that a term t is reducible, if a subterm of t can be replaced by a shorter term, without changing the original term's value.

A.8 Lemma (Subterm Lemma):

Let $t = s_1 \wedge \ldots \wedge s_n$, with $n \geq 1$, and let $\sigma=\{x \to t_0\}$ be a substitution with $x \in \mathbb{V}(t)$ and $x \notin \mathbb{V}(t_0)$. If $s_1\sigma \not\equiv s_1$, and $s_1\sigma \wedge s_2 \wedge \ldots \wedge s_n \cong t$, then t is reducible.

Proof: Assume that t is irreducible. Let $s_1' = (s_1\sigma)\downarrow$, and let $t' = s_1' \wedge s_2 \wedge \ldots \wedge s_n$. Then, since $s_1\sigma \not\equiv s_1$, and $t' \cong t$, we have $t' > t$. In particular, we have

$$t'\sigma > t\sigma,$$

which implies

$$s_1'\sigma \wedge s_2\sigma \wedge \ldots \wedge s_n\sigma > s_1\sigma \wedge s_2\sigma \wedge \ldots \wedge s_n\sigma,$$

and, since $s_1\sigma > s_1' = s_1'\sigma$, we have

$$s_1'\sigma \wedge s_2\sigma \wedge \ldots \wedge s_n\sigma > s_1'\sigma \wedge s_2\sigma \wedge \ldots \wedge s_n\sigma,$$

which is a contradiction. ■

It should be noted that the assertion of the subterm lemma also holds for a disjunction $t = s_1 \vee \ldots \vee s_n$.

A.9 Example:

Let $t = (x \vee y) \wedge \neg x$. We show that t is reducible. Let $\sigma = \{x \to 0\}$. First it is easy to see that $t \cong y \wedge \neg x$, and $y = y\sigma \not\equiv (x \vee y)$. If t were irreducible, then we had

$$y \wedge \neg x > (x \vee y) \wedge \neg x$$

hence

$$y \wedge \neg 0 = (y \wedge \neg x)\sigma > ((x \vee y) \wedge \neg x)\sigma = (0 \vee y) \wedge \neg 0 > y \wedge \neg 0$$

which is a contradiction.

A.10 Lemma:

Let t be a term with $\mathbb{V}(t) = \{x_1,\ldots,x_n\}$. Then there is a unique CNF-term $\tilde{t} = \tilde{c}_1 \wedge \ldots \wedge \tilde{c}_m$, where each $\tilde{c}_i$ is a clause containing all x_j's, and $\tilde{t} \cong t$. The term $\tilde{t}$ is called the *standardized CNF* of t. Each $\tilde{c}_i$ is called a *standard clause* of t. The notion of a *standardized DNF* is defined analogously.

Proof: See, for instance, Rudeanu (1974). ■

A.11 Example:

Let $t = (\neg x \vee y) \wedge (\neg x \vee \neg z)$. Then $\tilde{t} = (\neg x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z)$ is the standardized CNF of t.

A.12 Lemma:

If $t = t_1 \wedge \ldots \wedge t_n$, then for each $i \in \{1,\ldots,n\}$, there are standard clauses $\tilde{c}_{i1},\ldots,\tilde{c}_{ik_i}$, with

$$t_i \cong \tilde{c}_{i1} \wedge \ldots \wedge \tilde{c}_{ik_i}.$$

Moreover,

$$\bigcup_{i=1}^{n} \bigcup_{j=1}^{n_i} \tilde{c}_{ij} = \{\tilde{c}_1,\ldots,\tilde{c}_n\}.$$ ■

A.13 Lemma:

Let $t = x \vee y$. Then either $t\downarrow = t$, or $t\downarrow = \neg(\neg x \wedge \neg y)$.

Proof: Obvious. ■

A.14 Lemma:

Let $t = (x \vee y) \wedge (y \vee z) \wedge (z \vee x)$. Then $t\downarrow \in \{t_1,\ldots,t_8\}$, where

$t_1 = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$,

$t_2 = \neg(\neg y \vee \neg z) \vee \neg(\neg x \vee \neg z) \vee \neg(\neg y \vee \neg x)$,

$t_3 = (x \vee y) \wedge (y \vee z) \wedge (z \vee x)$,

$t_4 = \neg(\neg y \wedge \neg z) \wedge \neg(\neg x \wedge \neg z) \wedge \neg(\neg y \wedge \neg x)$

$t_5 = \neg[\neg(y \vee z) \vee \neg(x \vee z) \vee \neg(y \vee x)]$,

$t_6 = \neg[(\neg y \wedge \neg z) \vee (\neg x \wedge \neg z) \vee (\neg y \wedge \neg x)]$,

$t_7 = \neg[(\neg y \vee \neg z) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg x)]$,

$t_8 = \neg[\neg(y \wedge z) \wedge \neg(x \wedge z) \wedge \neg(y \wedge x)]$.

Proof:

a) Let $t{\downarrow} = s_1 \vee \ldots \vee s_n$, and let $\tilde{t}$ be the standardized DNF of t. Then $\tilde{t} = d_1 \vee d_2 \vee d_3 \vee d_4$, with

$d_1 = x \wedge y \wedge z$, $d_2 = \neg x \wedge y \wedge z$, $d_3 = x \wedge \neg y \wedge z$, $d_4 = x \wedge y \wedge \neg z$.

According to A.12, each s_i is equivalent to a disjunction of d_j's. Moreover, $t{\downarrow}$ must be symmetric in (x,y), in (y,z), and in (x,z), and thus there are only the following cases: Either $t{\downarrow} = s_1 \vee s_2$, with $s_1 \cong d_1$, and $s_2 \cong d_2 \vee d_3 \vee d_4$, or $t{\downarrow} = s_1 \vee s_2 \vee s_3$, with the following possibilities:

$s_1 \cong d_1 \vee d_2$, $s_2 \cong d_1 \vee d_3$, $s_3 \cong d_1 \vee d_4$,

$s_1 \cong d_1 \vee d_2 \vee d_3$, $s_2 \cong d_1 \vee d_3 \vee d_4$, $s_3 \cong d_1 \vee d_2 \vee d_4$.

Let $t{\downarrow} = s_1 \vee s_2$ with $s_1 \cong d_1$, and $s_2 \cong d_2 \vee d_3 \vee d_4$, and let $\sigma = \{z \to 1\}$. Then $s_1\sigma \not\cong s_1$. We show that $s_1\sigma \vee s_2 \cong s_1 \vee s_2$: We have

$s_1\sigma \vee s_2 \cong (x \wedge y \wedge z) \vee (x \wedge y \wedge \neg z) \vee d_2 \vee d_3 \cong$

$(x \wedge y) \vee d_2 \vee d_3 \cong (x \wedge y) \vee (x \wedge y \wedge \neg z) \vee d_2 \vee d_3 \cong s_1 \vee s_2$

Hence the subterm lemma implies that $s_1 \vee s_2$ is reducible.

Let $t{\downarrow} = s_1 \vee s_2 \vee s_3$. If $s_1 \cong d_1 \vee d_2 \cong y \wedge z$, $s_2 \cong d_1 \vee d_3 \cong x \wedge z$, $s_3 \cong d_1 \vee d_4 \cong y \wedge x$, then we have either $s_1 = y \wedge z$, $s_2 = x \wedge z$, $s_3 = x \wedge y$, and $t{\downarrow} = t_1$, or $s_1 = \neg(\neg y \vee \neg z)$, $s_2 = \neg(\neg x \vee \neg z)$, $s_3 = \neg(\neg y \vee \neg x)$, and $t{\downarrow} = t_2$.

If $s_1 \cong d_1 \vee d_2 \vee d_3 \cong (x \vee y) \wedge z$, $s_2 \cong d_1 \vee d_3 \vee d_4 \cong x \wedge (y \vee z)$, $s_3 \cong d_1 \vee d_2 \vee d_4 \cong y \wedge (x \vee z)$, then let $\tau = \{x \to 0\}$. It is easy to see that

$s_1\tau \vee s_2 \vee s_3 \cong s_1 \vee s_2 \vee s_3$,

and $s_1\tau \not\cong s_1$. Hence the subterm lemma implies that $s_1 \vee s_2 \vee s_3$ is reducible.

b) Let $t{\downarrow} = s_1 \wedge \ldots \wedge s_n$. Analogously to a) it can be shown that $t{\downarrow} \in \{t_3, t_4\}$ in this case.

c) Let $t{\downarrow} = \neg t'$, with $t' = s_1 \vee \ldots \vee s_n$. Then $t{\downarrow} \cong \neg s_1 \wedge \ldots \wedge \neg s_n$. Let $\tilde{t}$ be the standardized CNF of t. Then $\tilde{t} = c_1 \wedge c_2 \wedge c_3 \wedge c_4$, with

$c_1 = x \vee y \vee z$, $z_2 = \neg x \vee y \vee z$, $z_3 = x \vee \neg y \vee z$, $z_4 = x \vee y \vee \neg z$.

Then each $\neg s_i$ is equivalent to a conjunction of c_j's, and analogously to part a) it can be shown that either $t{\downarrow}$ is reducible according to the subterm lemma, or $t{\downarrow} \in \{t_5, t_6\}$. The case where $t' = s_1 \wedge \ldots \wedge s_n$ is treated analogously. ■

A.15 Lemma:

If the terms $x \vee (y \wedge z)$ and $x \wedge (y \vee z)$ are both irreducible, then $\mathfrak{R}$ is not convergent.

Proof: The assumption of the lemma implies $(x \vee y) \wedge (x \vee z) > x \vee (y \wedge z)$, $(x \wedge y) \vee (x \wedge y) > x \wedge (y \vee z)$, and, in particular, since both $y \wedge z$ and $y \vee z$ are

irreducible, $\neg(\neg y \wedge \neg z) > y \vee z$, and $\neg(\neg y \vee \neg z) > y \wedge z$. This proves all terms $t_1,\ldots,t_8$ of the previous lemma to be reducible, hence $\mathfrak{R}$ cannot be confluent. ■

Hence it will be assumed in the following that one of the terms $x \vee (y \wedge z)$ and $x \wedge (y \vee z)$ is reducible. It is sufficient to assume the term $x \vee (y \wedge z)$ to be reducible, the alternative case admitting an analogical proof. In particular, this assumption implies that each disjunct s_i of an irreducible term $t = s_1 \vee \ldots \vee s_n$ is either a negation or an atom.

A.16 Lemma:
Either the term $x \vee y$ or the term $x \wedge y$ is reducible.

Proof: We consider the term $t = (\neg x \vee y) \wedge (\neg y \vee x) \wedge (x \vee z)$. Since t is semi-symmetric in (x,y), but not symmetric, t must be reducible.

a) Let $t\downarrow = s_1 \wedge \ldots \wedge s_n$, where the s_i are not conjunctions.
If $n \geq 3$, let a be an arbitrary constant and let $\sigma = \{x \rightarrow a, y \rightarrow a, z \rightarrow \neg a\}$. We have $t > t\downarrow$, and in particular $t\sigma > t\downarrow\sigma$, where $t\sigma = (\neg a \vee a) \wedge (\neg a \vee a) \wedge (a \vee \neg a)$, and $t\downarrow\sigma = s_1\sigma \wedge \ldots \wedge s_n\sigma$. From $t\sigma \cong 1$ follows $t\downarrow\sigma \cong 1$, and hence $s_i\sigma \cong 1$, for each $i \in \{1,\ldots,n\}$. Hence $s_i\sigma > 1$, and, since $s_i\sigma$ is composed solely of the literals a and $\neg a$, the last step of this derivation must be of the form $a \vee \neg a \Rightarrow 1$. Thus we have the reduction $(\neg a \vee a) \wedge (\neg a \vee a) \wedge (a \vee \neg a) \Rightarrow^{+}_{\mathfrak{R}} (\neg a \vee a) \wedge \ldots \wedge (a \vee \neg a)$, where the second term has $n \geq 3$ conjuncts, which obviously contradicts the finite termination property of $\mathfrak{R}$.
Now let n=2, that is $t\downarrow = s_1 \wedge s_2$. Let $\tilde{t}$ be the standardized CNF of t. Then $\tilde{t} = c_1 \wedge \ldots \wedge c_5$, with

$c_1 = \neg x \vee y \vee z$, $c_2 = x \vee \neg y \vee z$, $c_3 = \neg x \vee y \vee \neg z$, $c_4 = x \vee \neg y \vee \neg z$, $c_5 = x \vee y \vee z$.

We distinguish two cases:

Case 1: s_1 is symmetric in (x,y). Then s_2 is also symmetric in (x,y), since $t\downarrow$ is. From lemma A.12 follows that s_1 and s_2 are equivalent to conjunctions of the c_i. Taking into account the symmetry property, there remain the following possibilities:

$s_1 \cong c_1 \wedge c_2$, $s_2 \cong c_3 \wedge c_4 \wedge c_5$,

$s_1 \cong c_3 \wedge c_4$, or $s_1 \cong c_3 \wedge c_4 \wedge c_5$, and $s_2 \cong c_1 \wedge c_2 \wedge c_5$,

$s_1 \cong c_1 \wedge c_2 \wedge c_3 \wedge c_4$, $s_2 \cong c_5$, $s_2 \cong c_1 \wedge c_2 \wedge c_5$, or $s_2 \cong c_3 \wedge c_4 \wedge c_5$.

In the first line, let $\sigma = \{z \rightarrow 0\}$. We have $s_1\sigma \wedge s_2 \cong t$, and $s_1 \not\cong s_1\sigma$. From the subterm lemma follows that $s_1 \wedge s_2$ is reducible.

In the second line, let $\tau=\{z\rightarrow 1\}$. We have $s_1\tau\wedge s_2 \cong t$, and $s_1 \not\equiv s_1\sigma$. From the subterm lemma follows that $s_1\wedge s_2$ is reducible.
In the third line, let $\varphi=\{x\rightarrow y\}$. We obtain in all three cases $s_1\wedge s_2\varphi \cong t$, and $s_2 \not\equiv s_2\varphi$, and from the subterm lemma follows that $s_1\wedge s_2$ is reducible.

Case 2: s_1 is not symmetric in (x,y). Then $s_1 = s_2\{x\rightarrow y;\ y\rightarrow x\}$, and for each c_i occurring in s_1, $c_i\{x\rightarrow y;\ y\rightarrow x\}$ must occur in s_2. Hence both s_1 and s_2 must consist of at least 3 c_i's, and both contain c_5. We have the following possibilities:

$s_1 \cong c_1\wedge c_3\wedge c_5$, $s_2 \cong c_2\wedge c_4\wedge c_5$,

$s_1 \cong c_1\wedge c_4\wedge c_5$, $s_2 \cong c_2\wedge c_3\wedge c_5$,

$s_1 \cong c_1\wedge c_2\wedge c_3\wedge c_5$, $s_2 \cong c_1\wedge c_2\wedge c_4\wedge c_5$,

$s_1 \cong c_2\wedge c_3\wedge c_4\wedge c_5$, $s_2 \cong c_1\wedge c_3\wedge c_4\wedge c_5$.

In the first, third, and fourth line, let $\sigma=\{z\rightarrow 1\}$. In either case, we have $s_1\sigma\wedge s_2 \cong t$, and $s_1 \not\equiv s_1\sigma$, hence $s_1\wedge s_2$ must be reducible according to the subterm lemma.
In the second line, we have $s_1 \cong (y\vee z)\wedge(x\vee\neg y\vee\neg z)$, and $s_2 \cong (x\vee z)\wedge(\neg x\vee y\vee\neg z)$. Let $\tau=\{z\rightarrow\neg x\}$. Since $s_1\tau\wedge s_2 \cong t$, and $s_1 \not\equiv s_1\tau$, $s_1\wedge s_2$ must be reducible according to the subterm lemma.

b) Let $t\downarrow = s_1\vee\ldots\vee s_n$. Let $\tilde{t}$ be the standardized DNF of t. Then $\tilde{t} = c_1\vee c_2\vee c_3$, with

$d_1 = \neg x\wedge\neg y\wedge z$, $d_2 = x\wedge y\wedge z$, $d_3 = x\wedge y\wedge\neg z$.

Obviously, $n\leq 3$, since otherwise one s_i, say s_n, would be redundant, that is $t\downarrow \cong s_1\vee\ldots\vee s_{n-1}$, which obviously contradicts the irreducibility of $t\downarrow$. If n=3, then $t\downarrow = s_1\vee s_2\vee s_3$, with $s_i \cong d_i$. But then $s_2\vee s_3 \cong x\wedge y \cong \neg(\neg x\vee\neg y)$, hence $s_2\vee s_3$ is reducible.
Thus we have $t\downarrow = s_1\vee s_2$, where both s_1 and s_2 are negations, with the following possibilities:

$s_1 \cong d_1$, $s_1 \cong d_1\vee d_3$, or $s_1 \cong d_1\vee d_2$, and $s_2 \cong d_2\vee d_3$,

$s_1 \cong d_3$, or $s_1 \cong d_1\vee d_3$, and $s_2 \cong d_1\vee d_2$,

$s_1 \cong d_2$, $s_2 \cong d_1\vee d_3$,

In the first line, $s_2 \cong d_2\vee d_3 \cong x\wedge y \cong \neg(\neg x\vee\neg y)$ holds. One of the last two terms is irreducible, hence $s_2 = x\wedge y$, or $s_2 = \neg(\neg x\vee\neg y)$. But s_2 is a negation, hence $t\downarrow = s_1\vee\neg(\neg x\vee\neg y)$, from which follows that $\neg(\neg x\vee\neg y)$ is irreducible and thus $x\wedge y$ is reducible.

In both the second and the third line, let $\sigma=\{z\rightarrow 1\}$. Then $s_1\sigma \vee s_2 \equiv t$, and from the subterm lemma follows that $s_1 \wedge s_2$ is reducible.

c) Let $t\downarrow = \neg s$. Then either $t\downarrow = \neg(s_1 \vee \ldots \vee s_n)$, which can be treated analogously to a), or $t\downarrow = \neg(s_1 \wedge \ldots \wedge s_n)$. In this case we obtain, similarly to b), $t\downarrow = \neg(s_1' \wedge s_2')$, with $s_1' \equiv d_1'$, or $s_1' \equiv d_1' \wedge d_2'$, or $s_1' \equiv d_1' \wedge d_3'$ and $s_2' \equiv d_2' \wedge d_3'$, where

$d_1' = x \vee y \vee \neg z$, $d_2' = \neg x \vee \neg y \vee \neg z$, $d_3' = \neg x \vee \neg y \vee z$.

First of all, $t\downarrow = \neg(s_1' \wedge s_2')$ implies that $\neg(x \wedge y)$ is irreducible, hence $\neg x \vee \neg y$ is reducible. We have $s_2' \equiv d_2' \wedge d_3' \equiv \neg x \vee \neg y$, and since s_2' is irreducible, $s_2' = \neg(x \wedge y)$. Now $t\downarrow = \neg(s_1' \wedge \neg(x \wedge y))$ implies that $\neg(x \wedge \neg y)$ is irreducible, hence $\neg x \vee y$ is reducible. Assume that s_1' is a disjunction, say $s_1' = u_1 \vee \ldots \vee u_m$. Then each u_j must be an atom, since both $x \vee (y \wedge z)$ and $x \vee \neg y$ are reducible. But it is easy to see that there is no disjunction of the atoms x, y, and z can be equivalent to one of the terms d_1', $d_1' \wedge d_2'$, or $d_1' \wedge d_3'$. Hence s_1' must be of the form $s_1' = \neg u$, which implies that $t\downarrow = \neg(\neg u \wedge \neg(x \wedge y))$ is irreducible. Hence also $\neg(\neg x \wedge \neg y)$ is irreducible, which implies that $x \vee y$ is reducible. ■

A.17 Lemma:

Either the terms $x \vee y$ and $\neg(x \wedge y) \wedge \neg(x \wedge z)$ are both reducible, or the terms $x \wedge y$ and $\neg(x \vee y) \vee \neg(x \vee z)$ are both reducible.

Proof:According to the previous lemma, either $x \vee y$ or $x \wedge y$ is reducible.
Case 1: $x \vee y$ is reducible. Consider the term $t = (\neg x \vee y) \wedge (\neg y \vee x) \wedge (\neg x \vee \neg z)$. Since t is semi-symmetric in (x,y), but not symmetric, t must be reducible. Since $x \vee y$ is reducible, $t\downarrow$ cannot be a disjunction. Hence we have either $t\downarrow = s_1 \wedge \ldots \wedge s_n$ or $t\downarrow = \neg s$. The first case is treated analogously to case a) of the previous lemma. In the case, where $t\downarrow = \neg s$, we have $t\downarrow = \neg(s_1' \wedge s_2')$, with $s_1' \equiv d_1'$, or $s_1' \equiv d_1' \wedge d_2'$, or $s_1' \equiv d_1' \wedge d_3'$ and $s_2' \equiv d_2' \wedge d_3'$, where

$d_1' = \neg x \vee \neg y \vee z$, $d_2' = x \vee y \vee z$, $d_3' = x \vee y \vee \neg z$.

Analogouly to case c) of the previous lemma, we obtain $s_2' = \neg(\neg x \wedge \neg y)$, hence from $t\downarrow = \neg(s_1' \wedge s_2')$ follows that the term $t_0 := \neg(x \wedge \neg(\neg x \wedge \neg y))$ is irreducible, which in turn implies that $t_1 := \neg(x \wedge y) \wedge \neg(x \wedge z)$, which is equivalent to t_0, is reducible.
Case 2: $x \wedge y$ is reducible. Consider the term $t = (x \vee y \vee z) \wedge (\neg x \vee \neg y)$. Since $x \wedge y$ is reducible, t is also reducible, and, moreover, $t\downarrow$ cannot be a conjunction.

Hence we have either $t\downarrow = s_1 \vee \ldots \vee s_n$ or $t\downarrow = \neg s$. The first case is treated analogously to case a) of the previous lemma. In the case, where $t\downarrow = \neg s$, we have $t\downarrow = \neg(s_1' \vee s_2')$, with $s_1' \cong d_1'$, or $s_1' \cong d_1' \vee d_2'$, or $s_1' \cong d_1' \vee d_3'$ and $s_2' \cong d_2' \vee d_3'$, where

$d_1' = \neg x \wedge \neg y \wedge \neg z$, $d_2' = x \wedge y \wedge z$, $d_3' = x \wedge y \wedge \neg z$.

Analogouly to case c) of the previous lemma, we obtain $s_2' = \neg(\neg x \vee \neg y)$, hence from $t\downarrow = \neg(s_1' \wedge s_2')$ follows that the term $t_0 := \neg(x \vee \neg(\neg x \vee \neg y))$ is irreducible, which in turn implies that $t_1 := \neg(x \vee y) \vee \neg(x \vee z)$, which is equivalent to t_0, is reducible. ■

A.18 Corollary:

$\mathfrak{R}$ is not confluent.

Proof: We consider again the term $t = (x \vee y) \wedge (y \vee z) \wedge (z \vee x) \cong (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$ of lemma A.14.

Case 1: The terms $x \vee y$ and $\neg(x \wedge y) \wedge \neg(x \wedge z)$ are both reducible. The reducibility of $x \vee y$ excludes t_1, t_2, t_3, t_5, t_6, and t_7 of lemma A.14 from being irreducible, and the reducibility of $\neg(x \wedge y) \wedge \neg(x \wedge z)$ excludes both t_4 and t_8 from being irreducible.

Case 2: The terms $x \wedge y$ and $\neg(x \vee y) \vee \neg(x \vee z)$ are both reducible. The reducibility of $x \wedge y$ excludes t_1, t_3, t_4, t_6, t_7, and t_8 of lemma A.14 from being irreducible, and the reducibility of $\neg(x \vee y) \vee \neg(x \vee z)$ excludes both t_2 and t_5 from being irreducible. ■

This corollary provides the proof of our main theorem A.5.

A.19 Lemma:

Let R be the following set of rules:

r_1: $\neg x \vee x \rightarrow 1$

r_2: $0 \vee x \rightarrow x$

r_3: $1 \vee x \rightarrow 1$

r_4: $\neg 1 \rightarrow 0$ $\qquad\qquad$ r^4: $\neg 0 \rightarrow 1$

r_5: $x \vee x \rightarrow x$

and let $\mathfrak{R}$=(AC,R). Then the system $\mathfrak{R}$ is confluent on clause terms.

Proof: It is easy to verify that there are no divergent critical pairs (t_1, t_2) with clause terms t_1, t_2. ■

Special Symbols

Index

Lebenslauf

Rolf Socher-Ambrosius, geboren am 11.5.1956 in Ludwigshafen/Rh. Verheiratet, zwei Kinder.

1962 - 1966	Volksschule in Maikammer
1966 - 1974	Kurfürst-Ruprecht-Gymnasium in Neustadt/Wstr. Abitur im Mai 1974
1974 - 1975	Grundwehrdienst
1975 - 1977	Studium der Mathematik mit Nebenfächern Informatik und Physik an der Universität Kaiserslautern
1977 - 1982	Studium der Mathematik mit Nebenfach Physik an der Johannes Gutenberg-Universität in Mainz
1982	Diplom in Mathematik
1982 - 1984	Tätigkeit als Softwareingenieur bei der Firma STS Systemtechnik und Software in Frankfurt
1984 - 1985	wis. Hilfskraft am Fachbereich Informatik der Universität Kaiserslautern
seit 1986	wiss. Mitarbeiter am Fachbereich Informatik der Universität Kaiserslautern