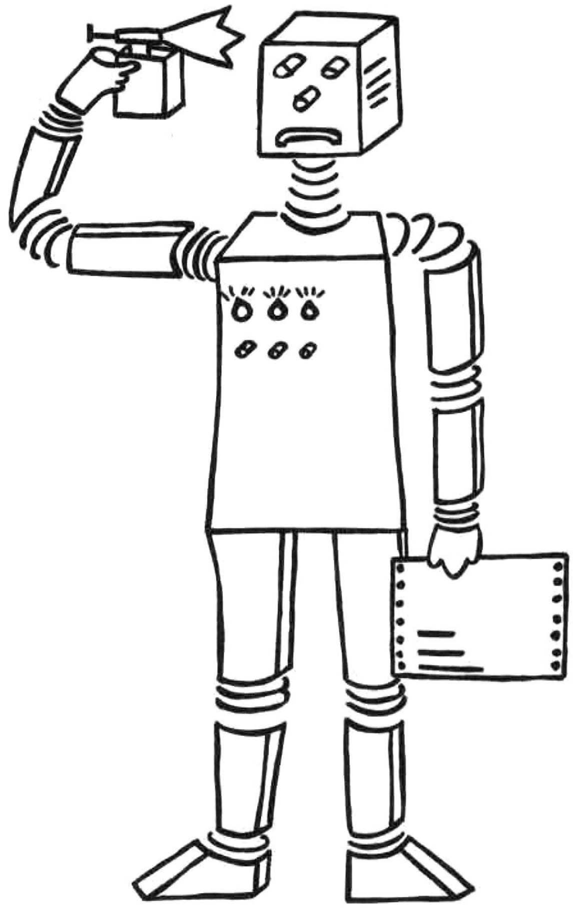
SEKI-REPORT

Artificial
Intelligence
Laboratories

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

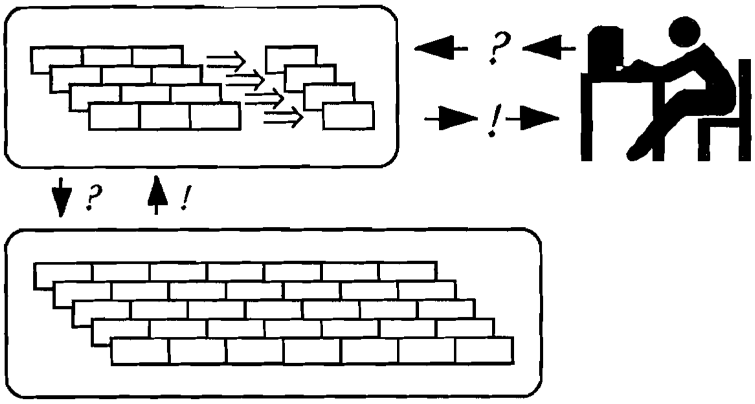


Using Automated Reasoning Techniques for Deductive Databasis

H.J. Ohlbach, J. Siekmann

SEKI Report SR-88-06

Using Automated Reasoning Techniques for Deductive Databases



H.J. Ohlbach, J. Siekmann

University of Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern
West Germany
siekmann@uklirb.uucp

Abstract

This report presents a proposal for a deduction component that supports the query mechanism of relational databases. The query-subquery (QSQ) paradigm is currently very popular in the database community since it focuses the deduction process on the relevant data. We show how to extend the QSQ paradigm from Horn clauses to arbitrary predicate logic formulae such that disjunctions in the consequent of an implication, negation in its logical meaning and arbitrary recursive predicates can be handled without restrictions. Various techniques to improve the search behaviour, such as lemma generation, query generalization etc. can be incorporated. Furthermore we show how to use clause graphs for compile time optimizations in the presence of recursive clauses and to support the run time processing.

Contents

1. Introduction

This report gives an outline for the design of a deductive component for a relational database (DB). Deductive components are currently used in two ways for DBs: automated reasoning techniques are used to check and prove integrity constraints (including access rights and the like) and secondly they are used to deduce new facts from given explicitly stored facts in a relational database. This report addresses the second problem. We do *not* address problems concerning parallel access of several users to the same database.

While there are plenty of special purpose techniques today for deductive components of DBs, including the linkage of a logic programming language with a relational DB, we view this problem as a *normal theorem proving task* - however with some special purpose deviations concerning the control regime.

A deductive DB is a special purpose theorem prover, since the queries are just existentially quantified formulae that are proved in order to obtain an answer (as in logic based question answering systems that were developed in the 1960s). However there are two essential differences between the application of a theorem prover in mathematics and in the DB world:

1. In mathematics there are usually plenty of complex clauses but only a few unit clauses (clauses with just one literal). Access to these unit clauses is not a particular problem as there are so few that the access time is essentially negligable. In contrast a deductive DB contains a very large set of unit clauses (the facts or relational entries) and access to these facts must be tightly controlled by focussing on those which are relevant for the particular query.
2. In mathematical applications one answer is usually sufficient (*completeness*), whereas in DB applications *all* answers may be required (*answer completeness*).

Various techniques have been proposed for the deduction mechanism in deductive DBs. See [BR86] or [Vi87] for an overview. As far as we know, all of them are restricted to Horn clauses and in order to gain efficiency some of them are even restricted to special Horn clauses such as function free clauses or clauses without recursive predicates. Two main groups of query evaluation methods can be distinguished:

1. The bottom up approach, a forward reasoning method using UR-resolution (Unit Resulting).
2. The top down approach, a backward reasoning method using SL-resolution (Selected Linear).

In the first approach, a definite Horn clause $A_1,\ldots,A_n \Rightarrow A$ is used to deduce from given facts $A_1',\ldots,A_n'$ an instance A' of A as a new fact. In the next steps this fact (or lemma) can be used to deduce further facts until the desired answer has been derived. The advantage of this method is that the deduced intermediate results can be reused several times whereas in many top down strategies the same fact must be derived again and again. This bottom up approach is particularily useful in mathematical applications where the formula to be proved often has the form 'hypothesis ⇒ theorem' with a conjunction of a few unit clauses as hypothesis. In database applications, however, there may be plenty of initial unit clauses which have nothing to do with the actual query. Therefore an enormous amount of useless deductions may be required until a single useful lemma is accidentally among the deduced facts. Thus, the bottom up approach (sometimes called *naive* evaluation) is obviously not an adequate strategy for database applications, at least without further refinements.

The top down approach uses Kowalski and Kuehner's SL-resolution [KK71] to work backwards from the query to the facts; it is therefore more goal directed. Since SL-resolution defines the search space without a fixed selection of the strategy to explore it, there is plenty of room to develop and optimize search strategies for different requirements and for various special cases. An advanced application of SL-resolution is its usage as a filter for UR-resolution. The idea here is that SL-resolvents are not actually created, but the backwards search imposed by SL-resolution is used to generate subqueries from queries. At the bottom level, i.e. with unit clauses, the subqueries are answered and the answers are combined using UR-resolution to get the answers for the requesting query. Thus, queries are generated backwards whereas the answers are propagated forwards. This approach combines the advantage of top down search - focusing on relevant data - with the advantage of bottom up deduction - reusage of intermediate results.

SL-resolution is not restricted to Horn clauses. Only a few modifications are necessary to use SL-resolution as a query - subquery deduction mechanism for the full predicate logic, i.e. negation in the body of a "Horn clause" is permitted and disjunctions in the consequent of an implication are also allowed as for example in "parent(x y) $\Rightarrow$ father(x y) $\vee$ mother(x y)". Mark Stickel has shown with his Prolog Technology Theorem Prover (PTTP) that this general application of SL-resolution can be implemented in a Prolog like manner yielding almost the same efficiency as Prolog itself [St86]. Therefore it should be possible to extend the techniques that have been developed for Horn clause based query evaluation systems, which usually do not apply depth first search as Prolog and the PTTP do, to arbitrary formulae.

Some problems which can be neglected in the Horn case, however, become significant in the non Horn case. For example finding an appropriate resolution partner for a given literal is no serious problem in the Horn case - only the head literals of a few clauses must be checked. In the general case each literal in the whole clause set may be a potential candidate for resolution. Thus there is a need for a sophisticated indexing mechanism. We demonstrate in the following that the clause graph idea, first proposed by R. Kowalski [Ko75], is not only an appropriate indexing mechanism, but it can be used as a "compilation" mechanism to greatly improve the run time behaviour by various optimizations: Redundant clauses and resolution possibilities (subsumption, tautology, purity) can be recognized and eliminated at "compile time", i.e. during the graph construction. Nonrecursive clauses can be compiled into the link structure of the graph, thus leaving only the really hard recursive part of the clauses for the run time mechanism.

This paper is divided into six more parts. In the second chapter we briefly sketch the basic notions of predicate logic and automated theorem proving. In the third chapter SL-resolution, both for Horn clauses and for the general case is discussed and various improvements that restrict the search space are presented. Predicate logic has no means to assign a predefined meaning to a certain symbol (other than the appropriate axiomatization). Especially in database applications however there are frequently used symbols with a fixed meaning, for example the arithmetic predicates and functions. In chapter four we therefore present theory unification and theory resolution as a general means for building in special knowledge about special symbols. Clause graphs are introduced in the fifth chapter and their application to query evaluation systems is presented in chapter six. Finally in the last chapter a prototype implementation of a deductive query evaluation system is described which solves nontrivial problems with highly recursive clauses.

2. Predicate Logic and Automated Deduction.

Before we go into technical details it may be useful to recapitulate informally the main notions of predicate logic and automated deduction. Detailed introductions can be found in [CL73], [Lo78], [Wo84], [Bu83] or [BB87].

2.1 Basic Notions

The Syntax of First Order Formulae:

Terms are built inductively from constant, variable and function symbols (the *signature*).

Examples: a, b, c We shall always use letters from the beginning of the alphabet to denote *constants.*

x, y, z We shall always use letters from the end of the alphabet to denote *variables.*

f(x g(a b)), +(3 z) (composite terms) (we use prefix notation).

(Sometimes we omit the parentheses and write for instance Pfffa instead of P(f(f(f(a)))).)

Atoms are objects of the form $P(t_1,\ldots,t_n)$ where the t_i are terms and P is an n-ary predicate symbol.

Examples: P(a b), father(mary joe),

$f(inv(x)\ x) \equiv x$, $2 < 3$ (we write equations and inequations in infix notation.)

Literals are negated or unnegated atoms.

Formulae are built from atoms, the universal quantifier $\forall$, the existential quantifier $\exists$, and the boolean connectives $\neg$ (not), $\wedge$ (and), $\vee$(or), $\Rightarrow$ (implies) and $\Leftrightarrow$ (is equivalent).

Examples: $father(mary\ joe) \wedge \neg mother(mary\ joe)$

$\forall x\ human(x) \Rightarrow mortal\ (x)$

$\forall x\ human(x) \Rightarrow \exists y\ \exists z\ father(x\ y) \wedge mother(x\ z)$

$(\forall x\ Q(x) \Rightarrow R(x)) \Leftrightarrow (\forall x\ \neg Q(x) \vee R(x))$

An object is called *ground* if it contains no variables.

The Syntax of Clauses

Any formula can be normalized into a conjunction of disjunctions of literals by the following operations:

An equivalence is replaced by two implications: $A \Leftrightarrow B \quad \rightarrow A \Rightarrow B \wedge B \Rightarrow A$

An implication is replaced by a disjunction: $A \Rightarrow B \quad \rightarrow \neg A \vee B.$

Negations are moved before atoms:

$\neg(A \vee B) \rightarrow \neg A \wedge \neg B$

$\neg(A \wedge B) \rightarrow \neg A \vee \neg B$

$\neg(\forall x\ Q(x)) \rightarrow \exists x\ \neg Q(x))$

$\neg(\exists x\ Q(x)) \rightarrow \forall x\ \neg Q(x))$

Application of the distributivity law: $A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C)$

"Skolemization" of existential quantifiers, i.e. replacing existentially quantified variables by composite terms built form a new n-ary function symbol and the n variables bound by the embracing universal quantifiers

Example: $\forall x\ human(x) \Rightarrow \exists y\ \exists z\ father(x\ y) \wedge mother(x\ z)$

$\rightarrow \forall x\ human(x) \Rightarrow father(x\ f(x)) \wedge mother(x\ m(x)).$

Due to the elimination of existentially quantified variables one may omit the universal quantifiers and take all remaining variables to be universally quantified.

Most of the existing theorem proving systems work with this normal form which is usually called clause form. A *clause* is a disjunction of *literals*. Due to the associativity and idempotence of the $\vee$ connective it is possible to regard clauses as sets of literals.

A *unit clause* is a clause with one literal only.

A *Horn clause* is a clause with at most one positive literal, a *definite* Horn clause is a clause with exactly one positive literal. Horn clauses are usually written in implication form: $A_1 \wedge \ldots \wedge A_n \Rightarrow A$ (or backwards in Prolog notation $A \Leftarrow A_1 \wedge \ldots \wedge A_n$) where A is called the *head* literal and the A_i are called the *body* literals.

Example: $\text{Parent}(x\ y) \wedge \text{Ancestor}(y\ z) \Rightarrow \text{Ancestor}(x\ z)$.

This is equivalent to $\neg\text{Parent}(x\ y) \vee \neg\text{Ancestor}(y\ z) \vee \text{Ancestor}(x\ z)$.

It is noted that every clause $B_1 \vee \ldots \vee B_m$ can be written in implication form where any B_i can serve as head literal: $\neg B_1 \wedge \ldots \wedge \neg B_{i-1} \wedge \neg B_{i+1} \wedge \ldots \wedge \neg B_m \Rightarrow B_i$. From a logical point of view this is not very exciting. From an operational point of view where such a clause is viewed as a procedure to derive the head literal from the body literals it means that a clause with m literals can be viewed as m procedures where each of its m literals is selected as a head literal. This observation provides a first means to generalize methods developed for Horn clauses to arbitrary clauses.

Substitutions

A *substitution* is a finitely representable mapping from variables to terms (actually an endomorphism on the term algebra associated with the signature). The finite set of variables on which a substitution σ is not identical, denoted $\text{DOM}(\sigma)$, is called the *domain* of σ, $\text{COD}(\sigma) := \sigma(\text{DOM}(\sigma))$ is called the *codomain* of σ. The typical representation of a substitution σ with $\text{DOM}(\sigma) = \{x_1,\ldots,x_n\}$ is a set $\{x_1 \mapsto \sigma x_1,\ldots, x_n \mapsto \sigma x_n\} = \{x_1 \mapsto t_1,\ldots, x_n \mapsto t_n\}$ of variable-image pairs. Substitutions may be applied to terms, atoms, literals, clauses and clause sets and actually to any object containing variables.

A substitution σ is *idempotent* iff $\sigma\sigma = \sigma$. The domain and codomain of idempotent substitutions have no variables in common. (for example $\{x \mapsto f(x)\}$ is not idempotent, but $\{x \mapsto f(y)\}$ is.)

A *ground substitution* consists of ground codomain terms only.

An object o_1 (a term or a substitution) is an *instance* of a second object o_2 if there is a substitution σ such that $o_1 = \sigma o_2$. In this case σ is sometimes called a *matcher.*

Examples : $g(a)$ is an instance of $g(x)$ with $\sigma = \{x \mapsto a\}$.

$g(g(x))$ is an instance of $g(x)$ with the non-idempotent substitution $\sigma = \{x \mapsto g(x))\}$.

Unification

A *unifier* for two terms s and t is a substitution σ that makes the two terms equal: $\sigma s = \sigma t$.

A unifier is *most general* (mgu) if it is no nontrivial instance of another unifier.

Examples: $\{x \mapsto a, y \mapsto a\}$ is a unifier for $g(x)$ and $g(y)$.

$\{x \mapsto y\}$ and $\{y \mapsto x\}$ are most general unifiers for these terms.

A *weak unifier* for two terms s and t is a tupel (ρ,σ) such that ρ is a variable renaming and $\sigma\rho s = \sigma t$.

Example: x and $f(x)$ are not unifiable, but they are weakly unifiable with a weak unifier $(\{x \mapsto y\}, \{y \mapsto f(x)\})$.

A *matcher* (one sided unifier) from s to t is a substitution making s equal to t: $\mu s = t$.

2.2 Inference Operations

The main task in automated theorem proving is to derive a theorem from a set of axioms. This is logically equivalent to the derivation of a contradiction from the axioms and the negated theorem, i.e. the *refutation* of a formula set. In clause based systems the empty clause represents the contradiction, therefore the theorem proving task is technically reflected as a search for the empty clause. In database applications a typical formula to be proved is an existentially quantified formula $\exists x_1 \ldots x_n \mathcal{F}$ whose negation is $\forall x_1 \ldots x_n \neg\mathcal{F}$. A byproduct of a successful proof of such a formula is the answer substitution, a binding of the variables $x_1, \ldots, x_n$ to some terms. Finding all possible answers therefore means finding all possible proofs. One inference operation which was explicitly designed for finding the empty clause and which produces the desired substitutions is Robinson's resolution rule [Ro65].

Resolution

Given two clauses $C = L \vee A$ and $D = K \vee B$ such that L and K are complementary unifiable, i.e. their signs are different and there exists a unifier σ for the corresponding atoms, one can infer a new clause, the *resolvent*, $\sigma A \vee \sigma B$, which is a logical consequence of the "parent" clauses C and D.

Example: C := ¬Parent(x y) ∨ ¬Ancestor(y z) ∨ Ancestor(x z)
D := Parent(u v) ∨ ¬Father(u, v) $\sigma = \{u \mapsto x, v \mapsto y\}$
Resolvent := ¬ Father(x y) ∨ ¬Ancestor(y z) ∨ Ancestor(x z)

Factoring

A factoring is an instantiation of a clause such that two or more literals become equal. The resulting instance is called a *factor*.

Example: Pxy ∨ Pyx ∨ Qxy → Pxx ∨ Qxx with the instantiation $\{y \mapsto x\}$.

Factoring and resolution are sometimes joined into one operation, which is also called resolution. These two inference rules are sufficient for proving any theorem which can be formulated in first order predicate logic. All other inference rules are either macrosteps, i.e. several resolution steps are comprised into one step (like hyperresolution, UR-resolution etc.) or they are special purpose inference rules taking into account the semantics of a particular symbol (like the equality symbol) if it is not expressed by the given axiom set.

UR-Resolution

Given a clause $C := L_1 \vee \ldots \vee L_{n-1} \vee L_n$, the "nucleus", and n-1 unit clauses $U_1, \ldots, U_{n-1}$, the "electrons", such that the pairs L_i, U_i are complementary unifiable with a most general simultaneous unifier σ, i.e. $\sigma L_i = \sigma U_i$ for i = 1,...,n-1, one can infer σL_n as a new unit clause.

Example: C = ¬Parent(x y) ∨ ¬Ancestor(y z) ∨ Ancestor(x z)
U_1 = Parent(tom, mary)
U_2 = Ancestor(mary, bill)
UR-resolvent = Ancestor(tom, bill)

UR-resolution is a complete inference rule for Horn clauses, i.e. every unsatisfiable Horn clause set can be refutet using UR-resolution only. For a Horn clause the nucleus consists of the clause body such that the UR-resolvent is an instance of the head literal. UR-resolution is not complete for arbitrary clause sets as the following example shows: shaves(x,x) ∨ shaves(barber,x) and ¬shaves(barber,y) ∨ ¬shaves(y,y). A refutation must apply the factoring rule.

3. SL-Resolution

SL-resolution is not actually a new deduction rule, but a restriction strategy that blocks certain resolution steps. An ordering strategy is used to select the next step from the set of admissible steps. The significance of SL-resolution is that its search strategy allows a procedural "task - subtask" view of resolution. We shall discuss the logical and the procedureal view of SL-resolution at first for Horn clauses and then for the general case.

3.1 SL-Resolution for Horn Clauses

3.1.1 The Logical View

Suppose we have a set of "input" Horn clauses and a "goal clause" G with no head literal. Starting with the goal clause as "center clause", SL-resolution selects at each step the last resolvent as the new center clause and determines the admissible resolution steps as follows: If the actual center clause is $C = C_1 \vee C_2$ where C_2 is the block of literals introduced from the latest "side clause", select among C_2 a literal L and select a corresponding resolution partner among the input clauses (the actual "side clause") and generate a new resolvent. Thus, SL-resolution resolves only between the last resolvent and an input clause. Furthermore it forces those literals to be "resolved away" which have been introduced *most recently.* While the selection of the resolution literal in the center clause is deterministic - each literal must eventually be resolved - the selection of the corresponding resolution partner among the input clauses is nondeterministic - one successful resolution chain is sufficient. This generates a tree like search space of resolvents that must be managed in some way by an actual implementation of SL-resolution.

Let us illustrate the method with an example:

Example 1: Consider the following clause set

	Prolog notation:	Clause notation:
C1:	$A(x\ y) \Leftarrow P(x\ y)$	$A(x\ y) \vee \neg P(x\ y)$
C2:	$A(x\ z) \Leftarrow P(x\ y) \wedge A(y\ z)$	$A(x\ z) \vee \neg P(x\ y) \vee \neg A(y\ z)$
C3:	$P(a\ b)$	
C4:	$P(b\ c)$	
C5:	$P(c\ d)$	

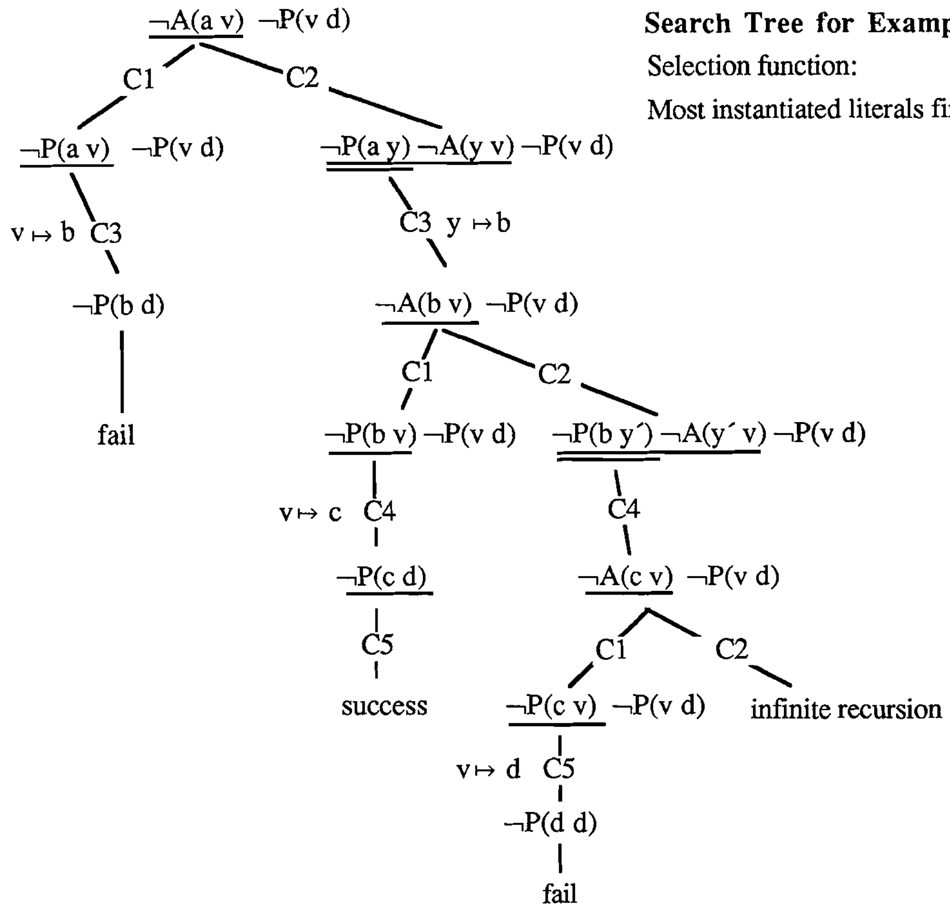
Query: $\exists\ v\ A(a\ v) \wedge P(v\ d)$? Its negation in clause notation is:

$\neg A(a\ v) \vee \neg P(v\ d)$

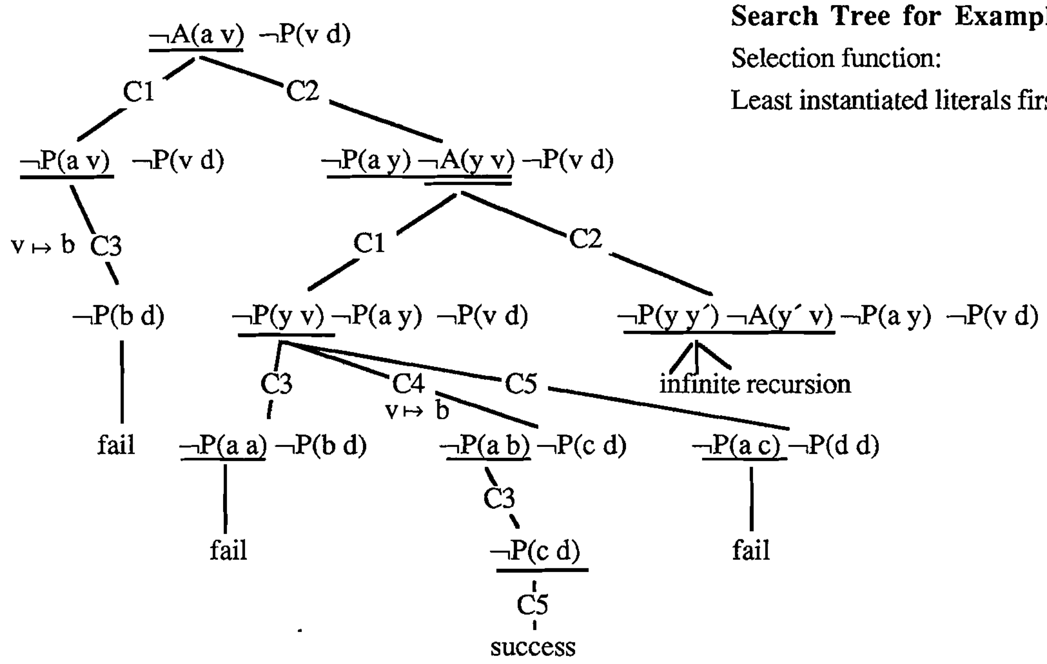
In the two figures below the search space is drawn for the negated query clause as goal clause and two different selection functions of the next literal in the center clause to be resolved upon:

1. Select most instantiated literals first
2. Select least instantiated literals first.

The most recently introduced literals which must be resolved first are underlined.



Search Tree for Example 1
Selection function:
Most instantiated literals first.



Search Tree for Example 1
Selection function:
Least instantiated literals first.

Since SL-resolution generates an ordinary search tree, the standard techniques for exploring search trees are applicable [Ni 80]. For example depth first search, as applied in Prolog, has the advantage that only one branch at a time is developed. Backtracking to previous choice points can be easily implemented using stacks. However, depth first search is not complete when the search tree contains infinite branches. Breadth first and especially heuristic search requires storing all open nodes, which may be very expensive. For highly nondeterministic problems where the standard methods to influence the depth first search by ordering of literals and clauses do not apply, however, heuristic search is much more promising.

3.1.2 The Procedural View.

While SL-resolution in its original definition was viewed as a restriction strategy of ordinary resolution, it soon turned out that a procedural interpretation is also possible and, at least in logic programming languages like Prolog, much more successful. The idea is to use the implication form of Horn clauses, where the body literals are positive, and to take the selected resolution literal L in the center clause, which is now a positive instance of a body literal, to be a ***query*** of the form "deduce instances of L". The actual resolution with a head literal of the side clause is then viewed as a procedure call that activates its body literals as a new bunch of ***related queries***, i.e. their answers must not instantiate the same variable differently.

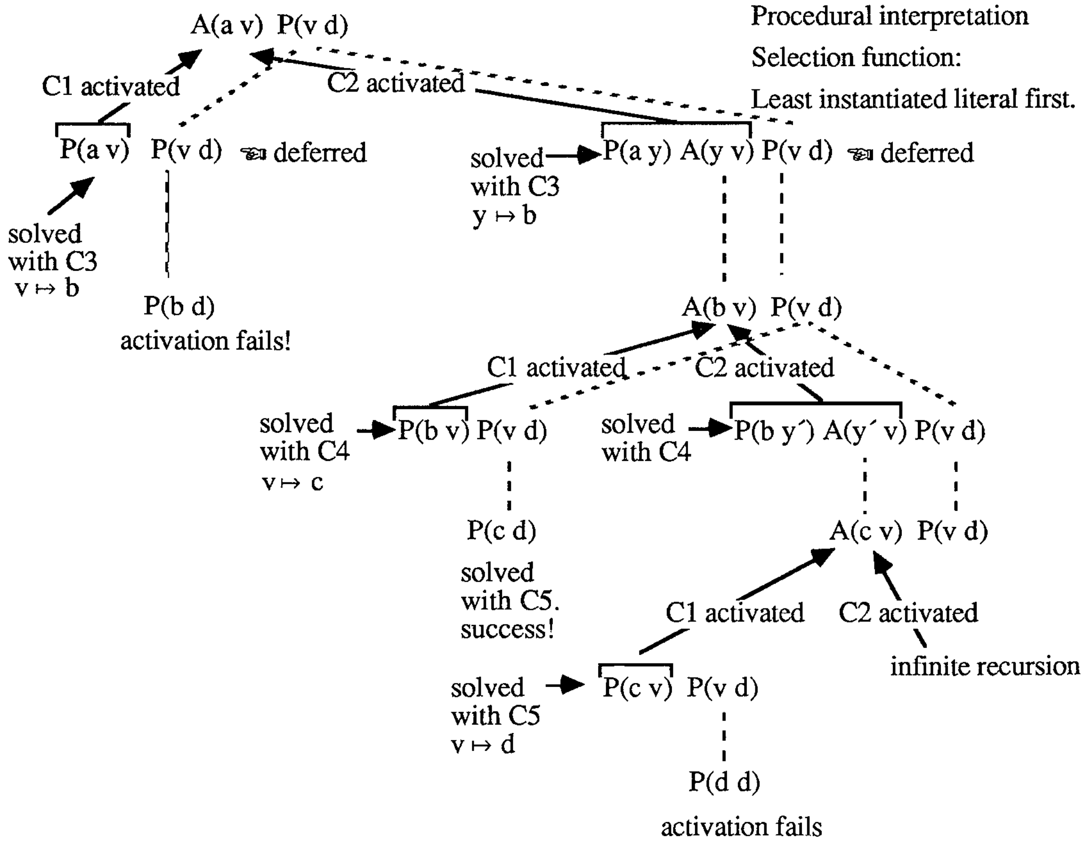
The corresponding graph below for example 1 shows that the search space is exactly the same as in SL-resolution, but there is no need to generate resolvents explicitly, just variable bindings are propagated.

Search Tree for Example 1

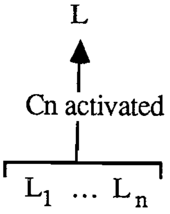
Procedural interpretation

Selection function:

Least instantiated literal first.



Read the graphical notation as follows:



means that the clause Cn allows to deduce L from $L_1 \ldots L_n$.
The task to deduce L (or instances of L respectively) can therefore be achieved by deducing $L_1 \ldots L_n$ (or compatible instances respectively).
Instances of literals are called compatible if the same variable is not instantiated differently.

Although this method has its success in logic programming where the amount of nondeterministic search can be greatly reduced by the programmer using syntactic means like the ordering of literals and clauses, there are still serious difficulties when the nondeterminism can not be controlled at the level of the initial clauses.

The main difficulties are:

1. There are infinite search trees even when the clause set belongs to a decidable class of formulae. Example 1 is of this type.
2. Multiple activation of the same query causes the same thing to be computed again and again. For example the symmetry law P(x y) ⇐ P(y x) may cause the infinite sequence of queries P(a b), P(b a), P(a b) , ... to be generated.
3. The fixed ordering of activations of body literals in a clause as subqueries causes a producer - consumer problem of the following kind:

   Consider the clauses

   C1: Pf(x) ⇐ Px

   C2: Pa

   C3: Qb

   and the query: $Px \wedge Qx$?

   The activation of Px produces the infinite sequence of answers a, f(a), f(f(a)),... for x, but none of them has a compatible solution to the second query Qx. Since the selection function is fixed in SL-resolution there is no possibility to backtrack to another sequence of query activations and to give the "consumer" Qx a chance to tell the "producer" Px to look for a solution Pb, which would immediately fail and terminate the process.

These problems are not new and some ideas have been proposed to solve them (c.f. [Vi 87]). Let us briefly discuss some of the basic intuitions behind them:

Triggered UR-Resolution (Lemma Resolution)

Each answer to a query or a subquery represents an initial or deduced unit clause. Usually the corresponding variable binding is the only thing that is exploited, but it is perfectly correct to add the derived and not subsumed unit clauses to the database in order to make it reusable for later queries. We shall illustrate the idea with an example:

Consider the clauses

C1: loves(v v)

C2: knows(x y) ⇐ loves (x y) which may be part of a larger clause set

and the subquery: knows(Tom, z)? which may be generated during the search.

Activation of clause C2 produces the binding x ↦ Tom, y ↦ z and the subquery loves(Tom, z)? which can be answered by C1 yielding the binding x ↦ Tom, y ↦ Tom, z ↦ Tom. Thus the new fact knows(Tom, Tom) can be added to the database for later use.

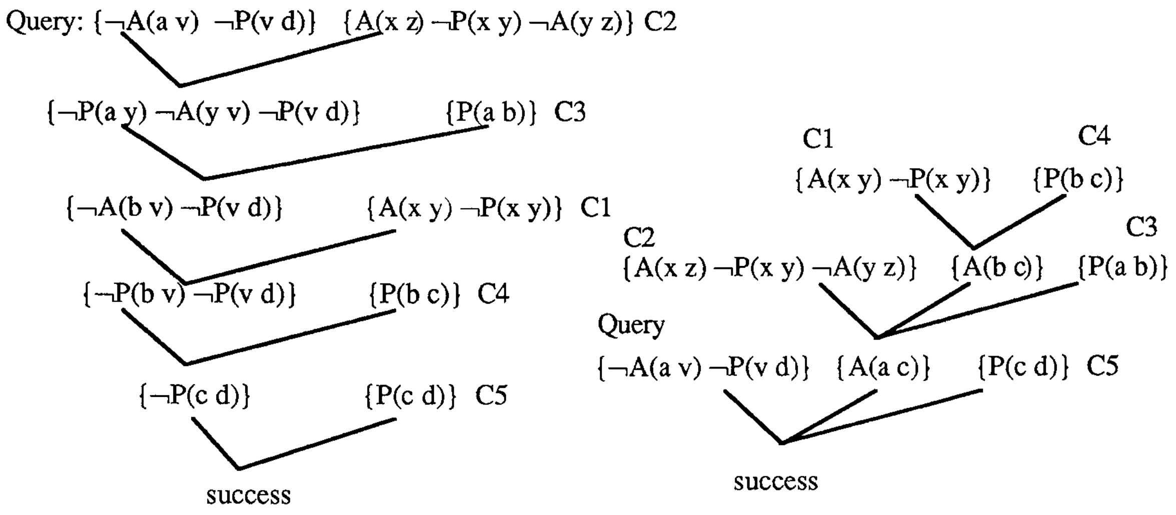
This is the usual lemma resolution procedure. However, one can do better.

When the answer to a query not only produces variable bindings, but the answer literals themselves, it is possible to compute a more general answer to the requesting query by unifying the answer literal with the original atom of the activated clause instead of the already instantiated query.

In the above example the query loves(Tom, z) would be answered by loves(v v) which can be unified with loves(x y) yielding the unifier {x ↦ y, v ↦ y}. This unifier allows to deduce knows(y y) as an intermediate result. As it is more general than the previous one it can be reused more often.

In this mode SL-resolution serves as a *top down* trigger for a *bottom up* UR-resolution: The main inference rule is UR-resolution, but only those UR-resolvents are generated which are relevant to the query. In fact it can be shown that every top down SL-resolution chain can be turned upside down into a bottom up UR-resolution chain where the query clause and the side clauses of the SL-resolution chain are the nuclei and the initial and derived unit clauses are the electrons. We demonstrate this with the successful resolution path of example 1:

SL-Resolution Chain for Example 1 | The corresponding UR-Resolution Chain



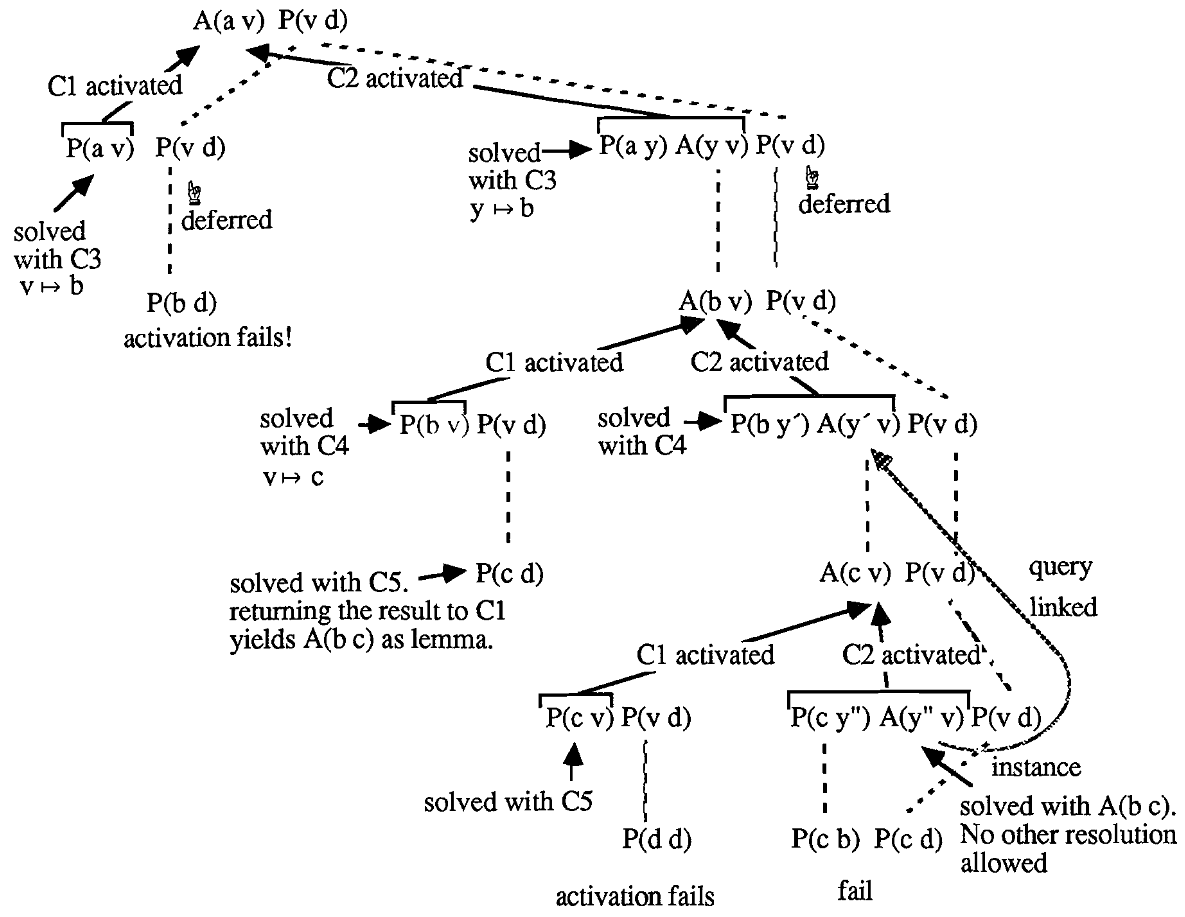
Query: {¬A(a v) ¬P(v d)} {A(x z) ¬P(x y) ¬A(y z)} C2
{¬P(a y) ¬A(y v) ¬P(v d)} {P(a b)} C3
{¬A(b v) ¬P(v d)} {A(x y) ¬P(x y)} C1
{¬P(b v) ¬P(v d)} {P(b c)} C4
{¬P(c d)} {P(c d)} C5
success
C1 C4
{A(x y) ¬P(x y)} {P(b c)}
C3
C2
{A(x z) ¬P(x y) ¬A(y z)} {A(b c)} {P(a b)}
Query
{¬A(a v) ¬P(v d)} {A(a c)} {P(c d)} C5
success

It is noted that the two versions of triggered UR-resolution coincide when there are only ground unit clauses and each variable in the head literal of a non unit clause occurs also in its body literals.

Query Linking

When a query Q_2 is an instance of a query Q_1, each answer to Q_2 is also an answer to Q_1. Conversely, it is possible to compute all answers for Q_2 just by forwarding the answers for Q_1 to Q_2. Q_2 need not activate any further clause by itself; potential infinite branches starting with Q_2 are therefore blocked. Combined with triggered UR-resolution this forwarding mechanism is easily realized by resolving Q_2 with initial or deduced unit clauses only. En exhaustive search started by Q_1 guarantees that all answers which are relevant to Q_2 will eventually be computed and stored as intermediate lemma. Vielle [Vi87] has shown for the function free case that the combination of the two ideas, triggered UR-resolution and query linking is sufficient to keep the search tree finite.

A finite search tree for example 1 using triggered UR-resolution and query linking:



Query Generalization

In the non function free case the SL-search tree is infinite in general, for otherwise either SL-resolution would not be complete or Horn logic would be decidable. But there are still possibilities to prune infinite parts in the search tree without losing answer completeness (c.f. [TS 86, KL 86])

Consider the example

C1: Px ⇐ Pf(x)

C2: Pa

and the query: Pb?

Although there is not a single fact deducible from the database, SL-resolution would generate the infinite sequence of queries: Pb?, Pf(b)?, Pf(f(b))?,.... To prevent this effect, the SL-resolution sequence can be stopped at a certain complexity limit, for example when the term depth in the query exceeds a threshold or when the depth in the search tree becomes too large. Now replace all ground terms in the query by variables, thus generalizing the query. Clearly each answer to the original query is also an answer to the generalized query, but there may also be unusable answers. The advantage, however, is that the chance to link a subquery to the generalized query is greatly increased. In the above example the generalized query Pf(f(...f(z)...)) would be generated and the next recursion with C1 produces Pf(f(...f(f(z))...)) which is an instance of Pf(f(...f(z)...)). As for this query only resolution with unit clauses is allowed and no such resolution is possible, the process terminates with failure.

Vielle [Vi 87] has proved that the combination of triggered UR-resolution, query linking and query generalization is still an answer complete proof procedure.

Query Instantiation

Query instantiation addresses the producer - consumer problem mentioned above. It is based on a combination of two ideas:

1. A subquery containing a variable is no longer seen as a request to derive all possible, but only a representative set of answers which differ in their termstructure up to a certain nesting limit. As an example consider the clauses

C1: $Pf(x) \Leftarrow Px$

C2: Pa

A query Py? would usually be answered with a, f(a), f(f(a)),.... If we set the nesting limit to 1 and filter out all equivalent answers, a and f(a) would be accepted, whereas all others would be rejected because their term skeletons f(...) are identical with the skeleton of f(a).

2. Since the rejection of "equivalent" answers destroys the answer completeness, we must extend the method with a mechanism for instantiating queries such that at least a sufficient subset of the lost answers can be regained.

Again an example:

C1: $Pf(x) \Leftarrow Px$

C2: Pa

C3: $Qf(f(a))$

query: $Py \wedge Qy$?

Suppose Py is activated first and gets the answers a and f(a). Instantiating Qy with any of the two bindings fails. However, since f(a) is only a representative for all answers f(...) we do not instantiate Qy with f(a) but with the generalized term f(z). In order to guarantee that the answers to Qf(z) are compatible with answers to P(y) we must instantiate P(y) as well, thus restarting the whole process with the instantiation $Pf(z) \wedge Qf(z)$? of the original query, but activating Qf(z) first. For the example above the whole procedure would work as follows:

1. The first answer a to Py? is not generalized and the second query Qa then fails.
2. The second answer f(a) to Py? is generalized to f(z). The whole query is then instantiated yielding $Pf(z) \wedge Qf(z)$.
3. Qf(z) is activated and returns the answer $z \mapsto f(a)$. Although in this case it is not necessary, f(a) will again be generalized to f(z′) such that the second instantiated query is $Pf(f(z')) \wedge Qf(f(z'))$?
4. Now Pf(f(z′)) is activated returning $z' \mapsto a$ which is compatible with the corresponding answer to Qf(f(a)).
5. The second answer f(a) to Pf(f(z′)) causes again an answer generalization and subsequent query instantiation $Pf(f(f(z''))) \wedge Qf(f(f(z'')))$?, but this time there is no answer to Qf(f(f(z"))) and the process stops. The process would also stop at this step if the clause C3 were Qf(f(b)), where there is no solution at all.

In general the query instantiation method activates from a bunch of related queries $Q_1\wedge\ldots\wedge Q_n$ one query Q_j according to the selection function, generalizes the answers to Q_j and rejects those answers which are instances of previously generalized answers. With the generalized variable binding σ, the instantiated queries $\sigma Q_1\wedge\ldots\wedge\sigma Q_n$ are created but the activation of σQ_j as the first subproblem is inhibited.

Since the answers to $\sigma Q_1\wedge\ldots\wedge\sigma Q_n$ are also answers to $Q_1\wedge\ldots\wedge Q_n$ the procedure is obviously sound. By induction on the termstructure it can easily be seen that any solution μ to $Q_1\wedge\ldots\wedge Q_n$ will be obtained after a finite sequence $\sigma_i Q_1\wedge\ldots\wedge\sigma_i Q_n$ of query instantiations where eventually some σ_k equals μ. Therefore query instantiation is still answer complete.

Unfortunately the query instantiation reduces the possibilities for query linking. Only those queries Q_2 which are instances of a query Q_1 can be linked to Q_1 whose terms have the same shape. For example a query Pb? can be linked to a query Px?, but a query Pf(b)? cannot because Px? might be satisfied with an answer f(a) and the equivalent answer f(b) which was the one Pf(b)? needed is discarded. It is currently not clear whether the query generalization and the query instantiation methods can be combined.

3.2 SL-Resolution for Non Horn Clauses

3.2.1. The Logical View

In case of non Horn clauses three extensions must be added to the SL-resolution system.

1. Factoring

Binary resolution without factoring is complete for Horn clauses, but it is incomplete in the general case. Therefore the factoring operation must also be included into the SL-resolution system.

2. Ancestor Resolution

It is not sufficient to resolve the center clause only with input clauses. Resolution with some of the ancestor clauses in the search tree may be necessary as the following example demonstrates:

Formulas:

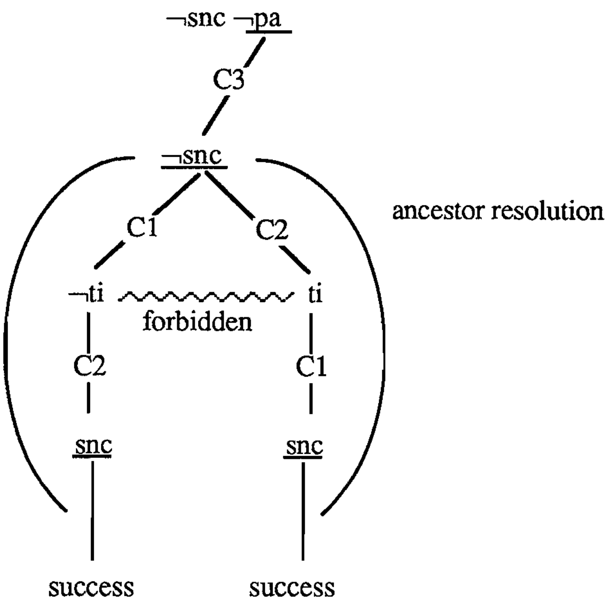
C1:	tax-increase ⇒ somebody-is-not-content
C2:	¬tax-increase ⇒ somebody-is-not-content
C3:	somebody-is-not-content ⇒ problems-arise.
query:	somebody-is-not-content ∧ problems-arise?

Abbreviated and written in conjunctive normal form the clauses are:

C1: $\neg ti \vee snc$ C2: $ti \vee snc$ C3: $\neg snc \vee pa$

negated query: $\neg snc \vee \neg pa$

The SL-resolution search tree looks as follows:



There is no proof without "ancestor resolution" because resolution between clauses in different branches of the search tree is not allowed. Although this restriction might require longer proofs, it usually decreases the branching rate of the search tree considerably.

3. Different Copies of the Query Clause.
Since the literals in the resolvents may be positive or negative it may become possible - and necessary - to reuse the initial query clause as side clause. (The reader can easily verify that this is not possible with Horn clauses.)
An example is:

C1: $Pa \vee Pb$
Negated query: $\neg Px$?

The proof tree is: $\neg Px$
C1 | $x \mapsto a$
Pb
negated query | $x \mapsto b$
success

The answer substitution is now indefinite: x may be a *or* b. A decision is not possible. Here SL-resolution offers an interesting choice to the user: Disabling the usage of the query clause as side clause guarantees that at most one instance of it is used; the answer is therefore definite. Enabling the usage of the query clause as side may produce indefinite answers. In the first case we always get a constructive proof, in the second case we allow for inconstructive proofs.

3.2.2. The Procedural View

A clause $C = L_1 \vee \ldots \vee L_n$ can be written as n implications of the form $L_i \Leftarrow \neg L_1 \wedge \ldots \wedge \neg L_{i-1} \wedge \neg L_{i+1} \wedge \ldots \wedge \neg L_n$, $i = 1,\ldots,n$. Just like the implication form of Horn clauses can be interpreted procedurally, we can give each of the implication forms of C a procedural interpretation: In order to derive instances of L_i, derive the corresponding compatible instances of $\neg L_1,\ldots,\neg L_{i-1}, \neg L_{i+1},\ldots,\neg L_n$. A negated literal $\neg L$ can be proved by refuting L. But it is not necessary then to actually multiply the clauses: We leave the clause as it is and just change the procedural interpretation. In order to derive instances of L_i, *refute* the corresponding simultaneous instances of $L_1,\ldots,L_{i-1}, L_{i+1},\ldots,L_n$. Hence it is possible to view a non Horn clause also as a procedure for deriving instances of a literal, this time not of a distinguished head literal, but just of any literal in the clause.

In the sequel we shall call a literal that is to be refuted a "task", as opposed to "query" which is a literal to be proved.

The incorporation of factoring and ancestor resolution into this procedural view, however, is not trivial. With the help of some examples we demonstrate the method.

Factoring

1. Factoring of Input Clauses

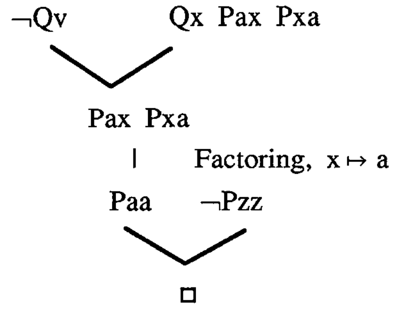
Consider the clause set

C1: Pax ∨ Pxa ∨ Qx

C2: ¬Pxx

negated query: ¬Qv

An SL-resolution proof is:



The corresponding procedural view is: The task ¬Qv activates the literal Qx of clause C1. This causes the generation of the two related subtasks Pax, Pxa. It is recognized that they are unifiable such that merging the two subtasks into Paa offers a new search path. (If instead of C2 the database contained the two facts Pab and Pba, this search path would fail, but in this case the original tasks can succeed.) Thus, factoring of the input clauses can be performed by unifying a set of related tasks.

Factoring of the Resolvents

We modify the above example slightly such that instead of factoring the input clauses, factoring the resolvent is possible.

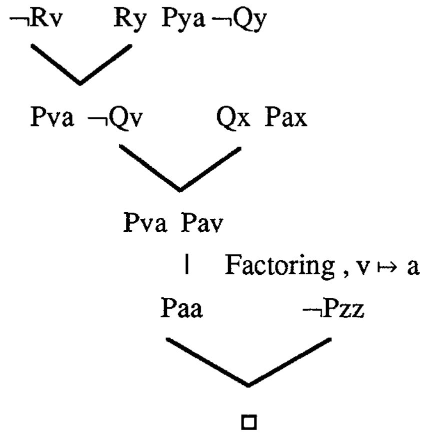
C1: Pax ∨ Qx

C2: ¬Qy ∨ Pay ∨ Ry

C3: ¬Pzz

The negated query is ¬Rv

An SL-refutation is:



The corresponding procedural interpretation is:

1. In order to satisfy the task ¬Rv, the literal Ry of C2 is activated (with substitution y ↦ v), generating the two subtasks Pva and ¬Qv.
2. ¬Qv is selected first. It activates the literal Qx of clause C1 which generates the subtask Pav.
3. This task must be checked against all "sibling tasks" of all ancestor tasks for possibilities to factorize.

In this case the temporarily deferred task Pva is found which can be unified with Pav. Thus, one solution for the current task is $v \mapsto a$.

4. Control is given back to C2 which activates the deferred task Pva with the returned binding $v \mapsto a$.
5. This task can be solved by C3.

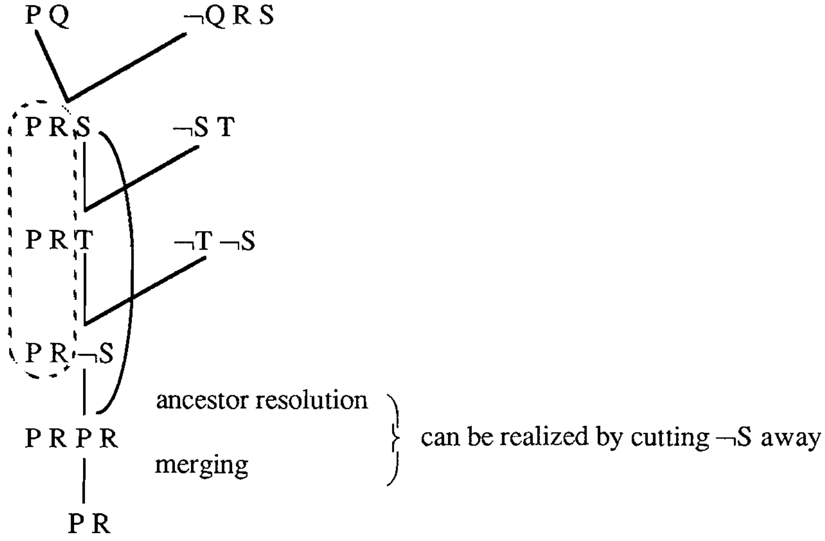
As we have seen, factoring of the resolvents requires access from the currently active bunch of related tasks to the siblings of all ancestor tasks.

Ancestor Resolution

Kowalski and Kuehner have shown that the SL-resolution strategy allows to realize ancestor resolution by just throwing away the resolution literal in the center clause, of course after instantiation with the unifier. All other literals of the ancestor clause are, from the previous resolution with this ancestor clause, still part of the current center clause and therefore automatically vanish by merging. The following example illustrates this effect.

Clauses:	C1:	$P \vee Q$
	C2:	$\neg Q \vee R \vee S$
	C3:	$\neg S \vee T$
	C4:	$\neg T \vee \neg S$

An SL-resolution path starting with C1 as initial center clause is as follows:



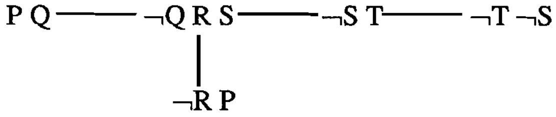
Thus, ancestor resolution can efficiently be realized by scanning the ancestor center clauses for a complementary unifiable literal that has already been used as resolution literal in the current resolution chain. Since the corresponding partner literal in the side clause is just its negation, there is no difference if instead of a complementary unifiable literal in an ancestor center clause, we look for a directly unifiable literal in the ancestor side clause. In the above example, for instance, instead of looking for a literal S in the ancestor center clauses, one can look for a literal ¬S in the ancestor side clauses. This has the advantage that factoring and ancestor resolution can be joined into *one operation* that scans the ancestor side clauses for unifiable literals.

Triggered Forward Resolution

In the case of Horn clauses we have shown how to turn a backward SL-resolution chain "upside down" into a forward UR-resolution chain. This observation has been exploited to generate unit clauses as intermediate results and to store them for further use. A very similar method can be applied in the general case as well. When a task L has been solved, i.e. instances of L have been refuted or, conversely, instances of ¬L have been proved, we want to generate them explicitly as new unit clauses Unfortunately, in the general case, a task L can be solved by subtasks which applied merging and factoring with literals of some ancestor clauses of L. From L's point of view, these literals are still part of a potential resolvent. Therefore, a solved task can only be used to generate a lemma when there is no factoring and ancestor resolution "across" L. Let us again illustrate this with an example.

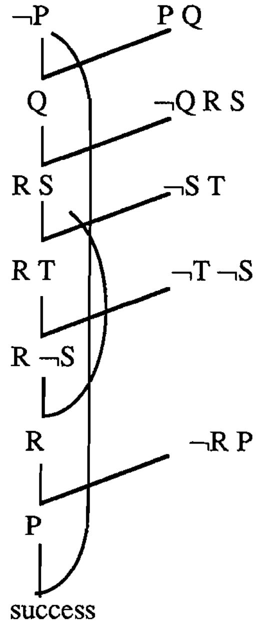
Clauses: C1: P∨Q
C2: ¬Q∨ R∨ S
C3: ¬S∨ T
C4: ¬T∨ ¬S
C5: ¬R∨ P

A graphical representation:
Links indicate potential resolution operations.

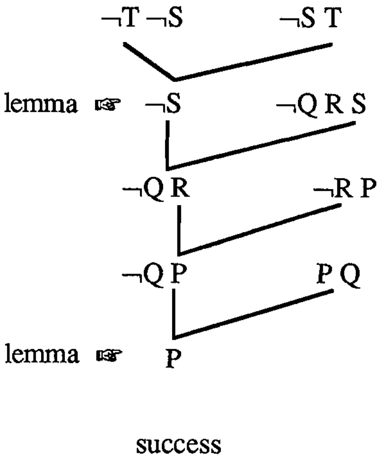


Assume again that these clauses are part of a larger clause set.
As can easily be seen, resolution with C3 and C4 yields the unit clause ¬S. Finally resolution with ¬S, C2 and C5 produces ¬Q ∨ P which can be resolved with C1 yielding a second unit clause P. No other unit clause is derivable. We shall show now how an SL-resolution chain, starting with a negated query ¬P can be turned upside down such that just these two unit clauses and no others are produced.

An SL-resolution chain:



The corresponding forward resolution chain:



The "lemma generating" procedure works as follows:

1. The initial task ¬P activates C1 with head literal P.
2. C1 generates the subtask Q.
3. Q activates C2 with head literal ¬Q.
4. C2 activates the related subtasks R and S.
5. The selection function selects S as first task to be activated.
    6. S activates C3 with head literal ¬S.

7. C3 generates the subtask T
8. T activates C4 with head literal ¬T.
9. C4 generates the subtask ¬S.
10. The merging mechanism recognizes that the current subtask ¬S is unifiable with the "sibling" literal ¬S of the ancestor task T (step 7). ¬S is taken to be solved.

Now control has to go back to step 5:

11. C4 informes the ancestor task T that the activation for ¬T succeeded, but with a merging step beyond T. Therefore no lemma can be generated.
12. C3 informes the ancestor task S that the activation for ¬S succeeded.
Since the former merging step did not go beyond S, ¬S can be generated as a new lemma.
13. The remaining task R is activated.
14. R activates C5 with head literal ¬R.
15. C5 generates the subtask P.
16. The merging mechanism recognizes that the current subtask P is unifiable with the "sibling" literal P of the ancestor task Q (step 2). P is taken to be solved.

Now control has to go back until step 1:

17. C5 informes the ancestor task R that the activation for ¬R succeeded, but with a merging step beyond R. No lemma can be generated.
18. There are no further subtasks to be solved, therefore C2 informes the ancestor task Q that the activation for ¬Q succeeded, but again with a merging step beyond Q.
19. C1 informes the initial task ¬P that the activation for P succeeded. Since the former merging step did not reach beyond ¬P, P can be generated as a new lemma.

We have demonstrated that the SL-resolution scheme can be used in the same way as for Horn clauses to design a task - subtask oriented query evaluation mechanism. Factoring and ancestor resolution can be integrated by allowing access from subtasks to related tasks of their ancestor tasks. A restricted form of lemma generation can also be integrated where the restrictions are determined by the "range" of factoring operations. The refinements like query linking, query generalization and query instantiation, that have been described for the Horn clause case, can also be incorporated into the general scheme.

This terminates our discussion of SL-resolution. We come back to this subject in chapter 6 where a method for realizing SL-resolution on the basis of a clause graph datastructure is presented.

4. Built in Operators

Predicate Logic has no means for assigning a predefined meaning to a certain symbol other than by an actual axiomatization. For example the literal < (a b) may mean "a is less than b" as well as "john loves mary". Especially in deductive database applications there are frequently used symbols with a fixed meaning, for example the arithmetic predicates and functions which we do not want to axiomatize explicitly, as this burdens the user of a deductive system with the task of writing down the same axioms for frequently occurring symbols over and over again and the system with the task of explicitly reasoning about them. Often there are more efficient specialized deduction procedures. Thus, it is usual practice to extend a logic by incorporating the special meaning of certain symbols directly into its semantics and to augment its inference mechanisms such that they respect this extended semantics.

There are different possibilities for incorporating theories into a deduction system. All of them influence the kind of deductions which are possible, either with special inference rules or with special unification algorithms.

4.1 Theory Unification

Theory unification is an extension of ordinary unification. It incorporates certain properties of functions into the unification algorithms and can be used wherever ordinary unification is used. Typically the properties are such that they can be expressed by equations like the commutativity or the associativity axiom or simple clauses like the symmetry axiom Pxy $\Leftrightarrow$ Pyx for a predicate P.

A *theory unifier* for two terms s and t and an equationally defined theory T is a substitution σ making the two terms equal under the theory T: $\sigma s =_T \sigma t$, i.e. $\sigma s = \sigma t$ holds in the theory T.

Example: Let T be the theory describing the symmetry of the predicate married: married(x y) $\Leftrightarrow$ married(y x). The substitution $\{x \mapsto \text{tom}\}$ is a T-unifier for the atoms married(x jane) and married(jane tom), which are not unifiable in the normal sense. Furthermore there are two T-unifiers for the atoms married(x y) and married(tom jane), namely $\{x \mapsto \text{tom}, y \mapsto \text{jane}\}$ and $\{y \mapsto \text{tom}, x \mapsto \text{jane}\}$. Both of them are most general.

There are theories with at most one most general unifier (called unitary), finitely many most general unifiers (called finitary theories), infinitely many most general unifiers (called infinitary theories) and non existing most general unifiers (called nullary theories). An example for a theory with infinitely many most general unifiers is the theory of associativity: g(g(x y) z) = g(x g(y z))).
Example: The terms g(a x) and g(x a) have the following sequence of unifiers: $\{x \mapsto a\}$, $\{x \mapsto g(a\ a)\}$, $\{x \mapsto g(g(a\ a)\ a)\}$, ... , $\{x \mapsto g(...g(a\ a)...a)\}$, ... None of them is an instance of another one.

4.2 Theory Resolution

Theory resolution is a general scheme for directly exploiting the information about the meaning of predicate symbols and function symbols in the calculus. In its general form it has been formulated by Mark Stickel at SRI [St85] but there were several special theory resolution rules known before. All the inference rules mentioned above and some more are instances of this general scheme. As a motivation for this scheme consider again the ordinary resolution rule for the ground case.

clause 1: $L \vee K$
clause 2: $\neg L \vee M$

---

resolvent: $K \vee M$

The main argument in the proof of the soundness of this rule is that either L or ¬L must be false and therefore either K or M must be true whenever clause 1 and clause 2 are both true, hence, the resolvent must also be true in this case. The essential fact is that literals L and ¬L are always contradictory. The syntactic criterion for a contradiction - equal atoms and different signs - can be generalized by incorporating special knowledge about the semantics of the predicate and function symbols into the notion of contradictory literals. Under the usual interpretation of the predicate <, for instance the literals a<b and b<a are not syntactically, but in a sense semantically contradictory. Therefore a resolution step

clause 1: $a<b \vee K$
clause 2: $b<a \vee M$

---

resolvent: $K \vee M$

should also be possible. As a further generalization we can give up the restriction that only two clauses are involved in a resolution step. The three literals a<b, b<c and ¬a<c are contradictory when the < relation is transitive. In this case the following deduction

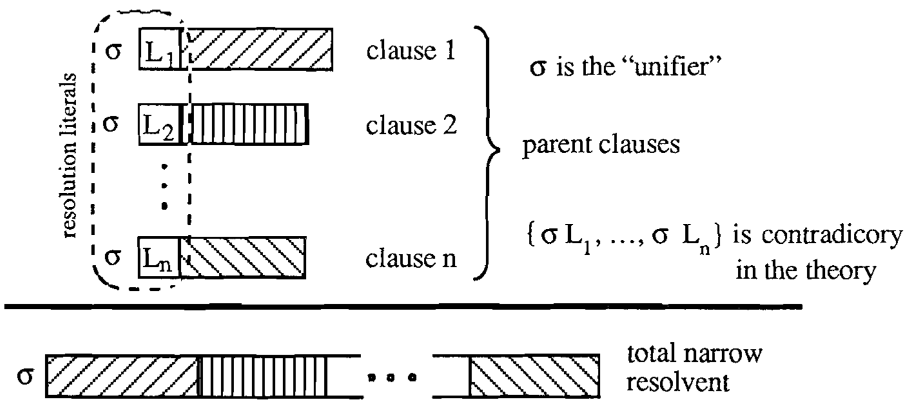
clause 1: $a<b \vee K$
clause 2: $b<c \vee M$
clause 3: $\neg a<c \vee N$

---

resolvent: $K \vee M \vee N$

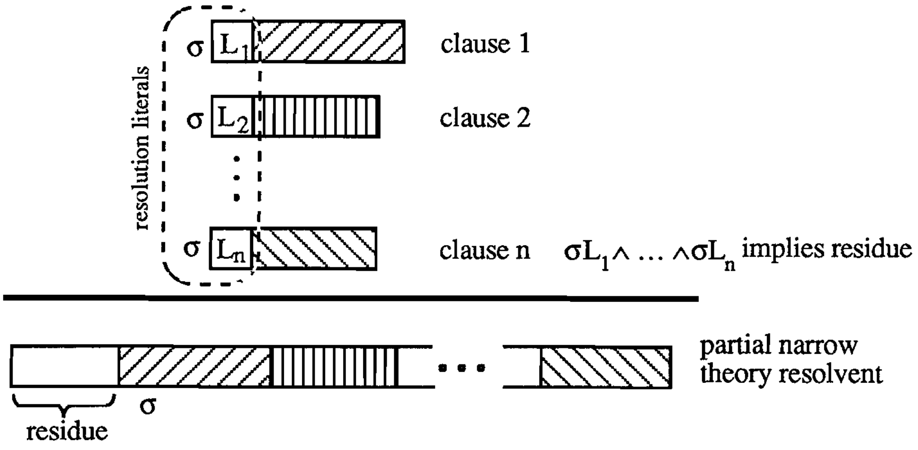
involving three clauses at once makes also sense.

In general the scheme for this type of inference, called *total narrow theory resolution* looks as follows:



Besides total narrow theory resolution there is *total wide theory resolution* which incorporates factoring and therefore uses more than one resolution literal per clause. In addition Stickel defined *partial theory resolution* which uses not a set of contradictory resolution literals but a set of literals that allow to deduce other literals. These additional literals, the "residue", are inserted into the resolvent.

The general scheme for partial narrow theory resolution is illustrated in the following figure:



A concrete example is : $a \leq b \quad P$

$a \geq b \quad Q$

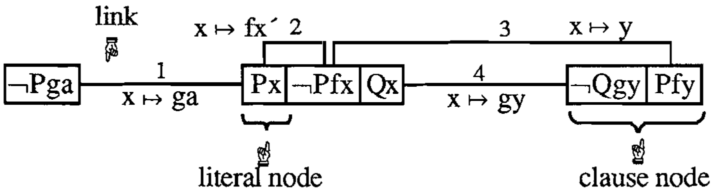
Resolvent: $a = b \quad P \quad Q$

5. Clause Graphs

In this chapter we introduce clause graphs, first as a datastructure and indexing mechanism for clauses and resolution operations, and then as a means for further reducing the search space during the query evaluation .

5.1. Clause Graphs as a Datastructure and Indexing Mechanism

Clause graphs have been used by R. Kowalski and others for representing clauses and resolution operations. Literals and clauses form the nodes of the graph and the edges (links) indicate the resolution possibilities. They are labeled with the corresponding unifiers. An example for a clause graph à la Kowalski for the three clauses ¬Pga, Px ∨ ¬Pfx ∨ Qx and ¬Qgy ∨ Pfy is:



The links 1,3 and 4 are ordinary resolution links, whereas link 2 is an "internal link" that connects complementary weak unifiable literals (unifiable after having renamed the variables of one literal). Internal links are in principle not necessary because they indicate resolutions with a renamed copy of the same clause. As we shall see below they are useful for finding links connecting a new resolvent with their own parent clauses.

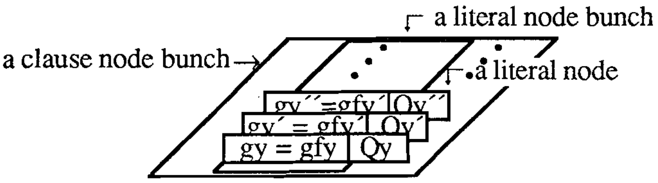
Clause Node Bunches

Considering copies of clauses with renamed variables and constructing links between these copies has only efficiency reasons, but is in principle not necessary for representing ordinary resolution operations. Clause copies, however, can no longer be neglected in the extension of the clause graph idea to theory resolution because a theory resolution operation may involve arbitrarily many copies of the same clause at once. The second clause, a conditioned equation, in the example

$$P(g(a))$$
$$g(y) = g(f(y)), Q(y)$$
$$\underline{\neg P(g(f(f(f(a)))))\qquad}$$
$$Q(a), Q(f(a)), Q(f(f(a)))$$

must be used in three different instantiations $\{y \mapsto a\}$, $\{y \mapsto f(a)\}$ and $\{y \mapsto f(f(a))\}$in order to find out that P(a), ¬P(g(f(f(f(a))))) and g(y) = g(f(y)) are contradictory. Therefore we introduced the notion of a "clause node bunch" as a conceptual entity for representing an infinite source of variable disjoint copies of a clause. A link may then connect different clause nodes in a clause node bunch.

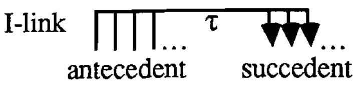
Example



(In an implementation only a finite number of clause nodes in a clause node bunch is usually generated at a time and indicated by the corresponding variable renaming substitution.)

I-links

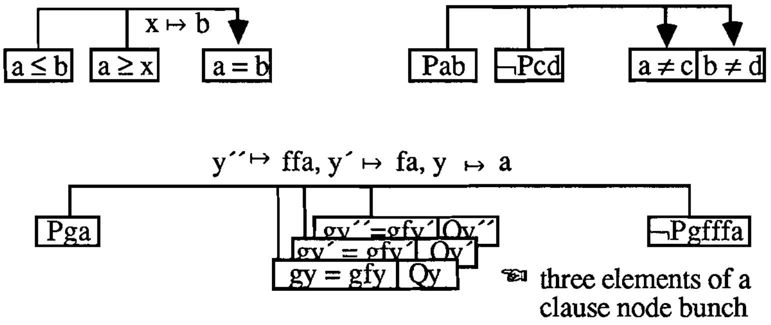
Links in the most general version of clause graphs represent partial narrow theory resolution operations. Therefore the so called "I-links" (Implication links) usually connect two groups of literal nodes, the "antecedent" and the "succedent", the first ones serving as resolution literals and the second ones serving as residue:



τ is the set of "link substitutions"
The notion "unifier" is no longer adequate because in general nothing will be "unified".

The graphical notation of an I-link is supposed to reflect the semantics of the antecedent and succedent: The conjunction of the τ-instances of the antecedent literals implies the disjunction of the corresponding instances of the succedent literals. Therefore antecedent literals must belong to different clauses (which are conjunctively connected) or at least to different copies of a clause, whereas succedent literals should belong to the same clause. Since literals of different clauses may be transferred to the same clause during a resolution operation, I-links with succedent literals in different clauses are useful as well.

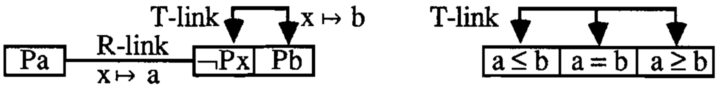
Examples
for I-links



R-links and T-links

Two special types of I-links, those with an empty succedent and with an empty antecedent respectively are of particular interest. Links of the first type (Resolution links or R-links) indicate total narrow theory resolutions and links of the second type (Tautology links or T-links) indicate actual or potential (after instantiation) tautologies.

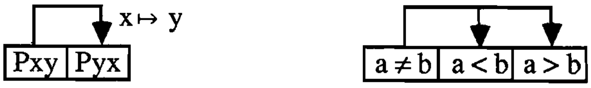
Examples
for R- and T-links



F-Links

A third special type of I-links is also of particular interest, links connected to one clause node only and with one antecedent literal node and at least one succedent literal node. Links with this structure represent a generalization of the factoring operation: Since the antecedent literal implies the succedent literal or the corresponding instances respecitvely and they are disjunctively connected, one can create an instance of the clause where the antecedent literal is removed.

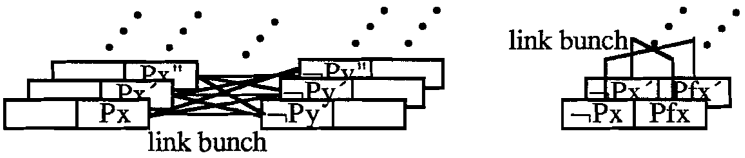
Examples
for F-links



Link Bunches

An obvious extension to the link concept that is analog to the concept of clause node bunches is the concept of "link bunches". Just as a clause node bunch is an infinite source for variable disjoint copies of a clause, a link bunch is an infinite source for copies of a link which are connected to the corresponding copies of a clause in a clause node bunch.

Examples
for link bunches:



In this chapter no further details of clause node bunches and link bunches need to be explained because the main principles of the operations on clause graphs can be described on the clause node and link level. Clause node bunches and link bunches come into play when dealing with the non-trivial cases, self resolution, recursion etc., that make life hard and first order logic undecidable.

Link Substitutions

To improve conceptual clarity we consider the link substitutions as a - possibly infinite - set of ground substitutions. For example the substitution $\{x \mapsto y\}$ is taken to be a representation of the set of ground substitutions $\{\{x \mapsto a, y \mapsto a\}, \{x \mapsto b, y \mapsto b\}, \{x \mapsto f(a), y \mapsto f(a)\}\ldots\}$. This view has the advantage that operations on substitutions which are necessary in the algorithms described below can be easily expressed with set operations. The merging of substitutions for example turns out to be the intersection of sets of ground substitutions. (As an example consider the merging of $\sigma = \{x \mapsto y\}$ and $\tau = \{x \mapsto a\}$ yielding $\{x \mapsto a, y \mapsto a\}$. In the ground substitution representation we obtain this substitution by intersecting $\{\{x \mapsto a, y \mapsto a\}, \{x \mapsto b, y \mapsto b\}, \{x \mapsto f(a), y \mapsto f(a)\}\ldots\}$ with $\{\{x \mapsto a, y \mapsto a\}, \{x \mapsto a, y \mapsto b\}, \{x \mapsto a, y \mapsto f(a)\}\ldots\}$)

As a further advantage we can use the set-difference operation on the sets of ground substitutions for representing "all σ-instances except those which are also τ-instances". The τ-instances of a link with link substitution σ might for example represent tautologous resolvents and should therefore be removed.

It is only a technical problem to find an adequate representation for these ground substitution sets in an actual implementation.

5.2 Clause Graph Inference Rules

The inference rules for clause graphs given below define a complete deduction system which can be used as an alternative to SL-resolution. In the database application however, we apply the inference rules together with the clause graph reduction rules as an optimization procedure that works on the initial graph and reduces the size of the graph such that during query evaluation the search paths between query and database become smaller. In case there are no recursive predicates the optimization with clause graph inference and reduction rules is powerful enough to generate immediate access form the query to the database.

In the sequel only simplified versions of the inference and reduction rules are presented. The exact formulations are technically more complicated, but do not show new principles.

Clause Resolution (*cres*)

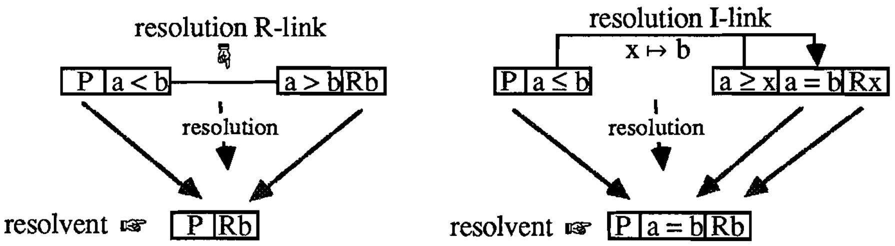
The "traditional" inference rule for clause graphs is clause resolution as proposed by R. Kowalski. It consists of two parts, the generation of the resolvent and a mechanism for "inheriting" the links connecting the new resolvent with the rest of the graph from the links connected to their parent clauses. The inheritance mechanism avoids searching the whole graph for resolvable literals. The formulation of the algorithms for partial narrow theory resolution is technically more complicated, but the ideas are the same. A partial narrow resolution is indicated by an I-link: The antecedent literals are the resolution literals and the succedent literals are the residue. The link substitution is just the resolution substitution (unifier).

Clause graph resolution works as follows:

1. Generation of a partial narrow resolvent from an I-link:

- Form a new clause node by joining the remainders of the antecedent clause nodes without the antecedent literal nodes themselves with the succedent literal nodes and applying the link substitution.
- Make the new clause node variable disjoint with all other clause nodes.

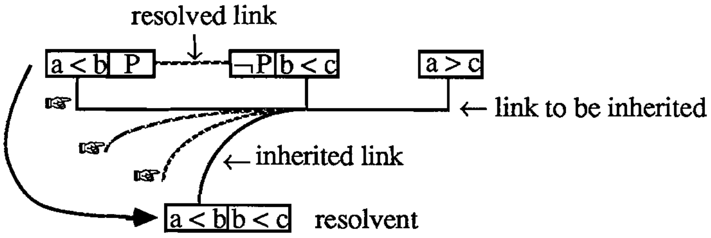
Examples for the generation of resolvents



2. Link Inheritance

The idea for the link inheritance mechanism is very simple: In order to connect a link with a literal node of the resolvent, "grasp" a link connected to its parent literal node and "pull it down" to the resolvent literal node. The link substitution for the new link can be computed from the link substitution of the old link and the link substitution of the resolved link (substitution merging).

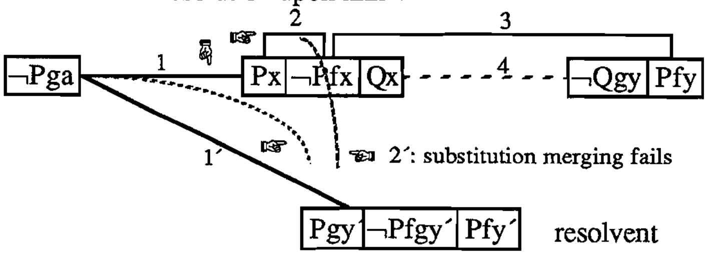
Example
for link inheritance



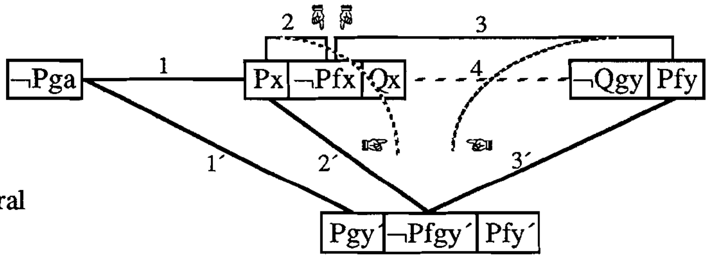
Of course only a copy of the original link is manipulated in this way. The original link itself remains untouched.

The next example demonstrates that this idea is powerful enough to get the internal links of the resolvent as well as the link connecting the resolvent with its parent clauses.

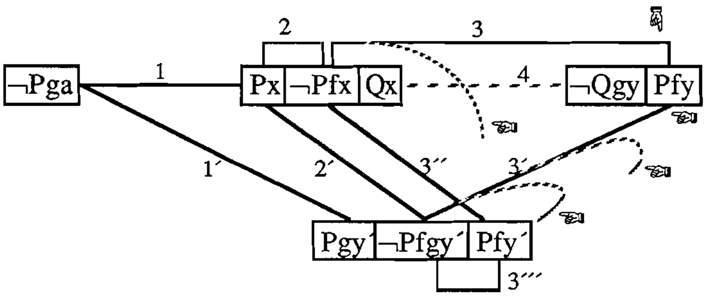
Example link inheritance after resolution upon link 4



link inheritance
to the first literal
of the resolvent



link inheritance
to the second literal
of the resolvent



link inheritance
to the third literal
of the resolvent

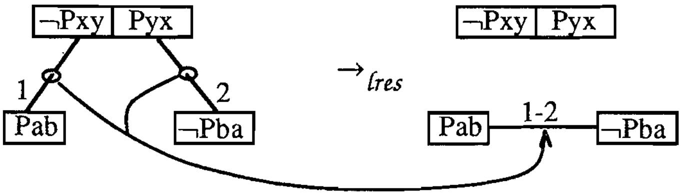
As a final operation the resolved link can be removed in order to inhibit a repetition of the resolution. (This shall not be further elaborated, c.f. [Ei86].)

Explicit generation of clause graph resolvents is very expensive because usually hundreds of new links must be generated besides the new clause. Since the literals in the resolvent are just instances of literals in the parent clauses and the links to these literals are essentially copies of the original links, a resolvent contains mostly redundant information. However, there may be resolvents which trigger some of the reduction operations explained below and cause the removal of clauses and links. Thus, clause graph resolution can be applied in certain situations in order to shrink the graph.

Link Resolution (*lres*)

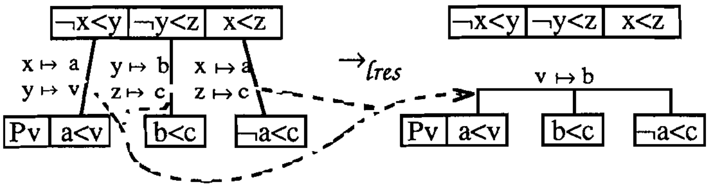
Link resolution is the main inference rule on our version of clause graphs. Just as clause resolution derives from a set of clauses a new clause, link resolution derives from a set of links and a single clause a new link. The basic idea shall be explained with a few examples.

Consider the graph for the symmetry axiom {¬Pxy, Pyx} and the two clauses {¬Pab}, {Pba}.



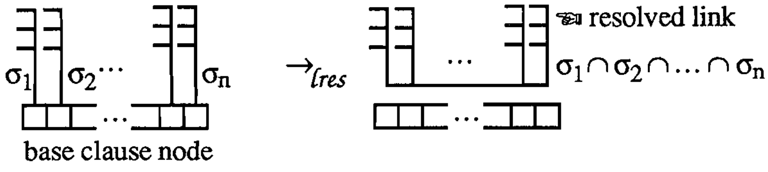
The links 1 and 2 have compatible unifiers {x ↦ a, y ↦ b}. Therefore it is possible to combine these links yielding a new link 1-2 between Pab and ¬Pba which contains the information that both literals a contradictory with respect to the symmetry of the predicate P. Thus, the symmetry clause has been "compiled" into the theory and is now contained in the semantics of the new link 1-2.

In the next example we "compile" the transitivity axiom for the relation < into the theory by combining the three binary links to one link connecting three literals. The corresponding theory resolvent is Pb. This clause could also be deduced with the three binary resolutions indicated by the three links of the transitivity axiom.



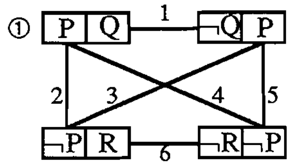
The principle for the general link resolution operation is:

Take a clause node and for each literal node in the clause take just one link which is connected with its antecedent to the literal node. Compute the new "resolved" link by intersecting all the unifiers attached to the links joining all parent literals except those in the selected clause.

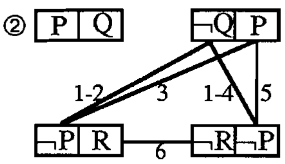


(Of course the original links remain untouched. They can only be removed when all possible resolutions with these links are executed.)

Example A sequence of link resolutions that proves the unsatisfiability of the clause set {{P,Q}, {¬Q,P}, {¬P,R}, {¬R,¬P} is shown in the next figures.

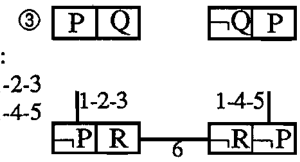


link resolution:
links 1,2 →1-2
links 1,4 →1-4

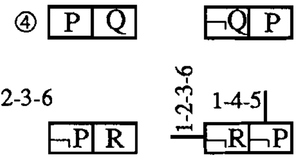


The links 1, 2 and 4 can be removed because all possible combinations are executed in graph ②.

link resolution:
links 1-2,3 →1-2-3
links 1-4,5 →1-4-5



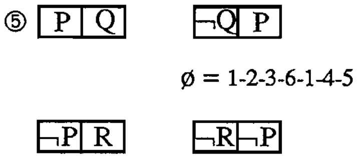
link resolution:
links 1-2-3,6 →1-2-3-6



The links 1-2-3 and 1-4-5 contain the information that ¬P is false in the theory generated by the clauses {P, Q} and {¬Q, P}. Note that a resolution of links connected with the same literal, as for example 1-2 with 3, includes an implicit factoring operation.

Finally the "empty link" is generated in graph ⑤. The empty link states that the empty set has no model in the theory generated by the axioms. This is the elementary contradiction which indicates that the axioms are unsatisfiable.

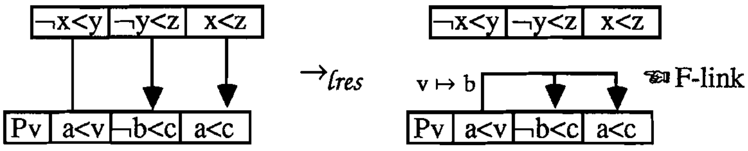
link resolution:
links 1-2-3-6, 1-4-5
→1-2-3-6-1-4-5,
the "empty" link.



■

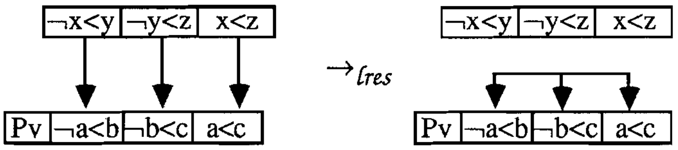
Some more examples show what happens, when different kinds of links are combined.

The resolved link in the graph



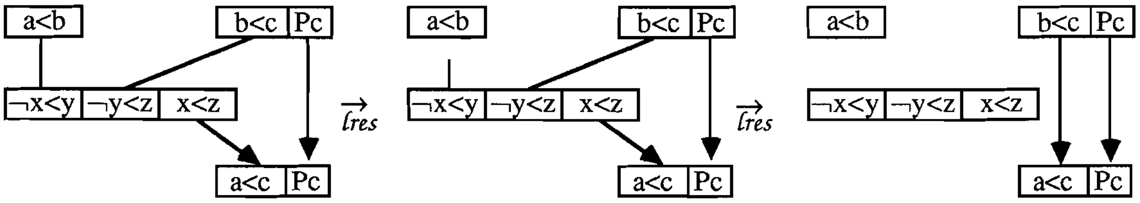
is actually an F-link stating that a < b implies ¬b < c or a < c. This information could be used to generate a shorter instance {Pb, ¬b < c, a < c} of the second clause.

The three links in the left graph



indicate that the second clause is subsumed by the transitivity clause. The corresponding combined T-link in the right graph states that the second clause is a tautology in the theory of the first clause. This example shows the very tight correlation between subsumption and tautology.

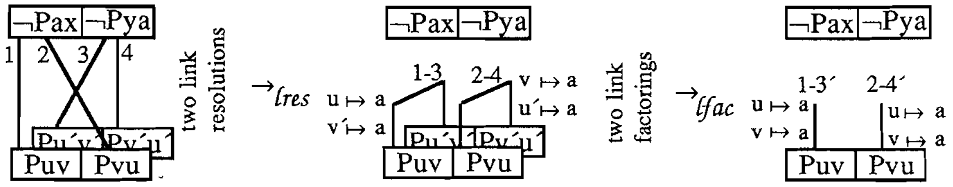
The two successive link resolutions in the example below generate a link which represents the information that b<c implies a<c under the theory a<b and the transitivity of <. Using this link and the parallel one, it can be recognized that the clause {b<c, Pc} subsumes {a<c, Pc}. A further link resolution would produce a T-link containing the same information.



(The first operation is a link resolution on the unit clause a<b which simply disconnects the link from the unit clause. The shortened link contains the information that the corresponding instance ¬a<b of ¬x<y is false in the theory of the unit clause.) ■

Link Factoring (*lfac*)

The link factoring rule is the analog to the clause factoring rule. It is applicable to a single link which is connected to two or more renamed copies of the same literal node and generates a new link with fewer adjacent literals, but a stronger instantiating substitution. In the example below the links 1-3 and 2-4, must be factorized in order to generate the empty link in a subsequent link resolution.



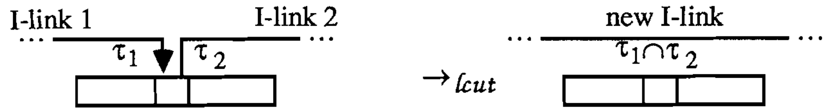
The factorized links 1-3′ and 2-4′ are not equivalent with 1-3 and 2-4 respectively because they instantiate both variables u and v with a, whereas the original links 1-3 and 2-4 leave one variable uninstantiated.

R-Link Resolution (*rlres*)

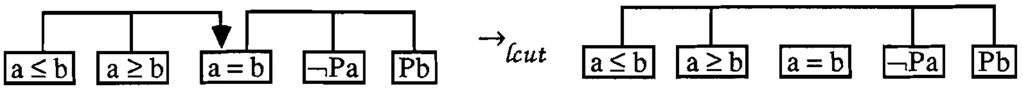
R-link resolution is just the special case of link resolution where R-links only are combined. The interesting thing about R-link resolution is, that this restricted inference rule together with the link factoring rule is refutation complete, i.e. for each unsatisfiable clause set C there exists a sequence of R-link resolutions and link factorings on some initial clause graph over C that terminates with the empty link. All other operations on clause graphs are in principle not necessary, however, they can considerably increase the efficiency of the deduction system.

The Link Cut Rule (*ℓcut*)

The link cut rule is derived from the cut rule of Gentzen's sequence calculus which states that a new sequence can be derived from two sequences by joining the antecedents and succedents and removing the common parts of the succedent of the first sequence and the antecedent of the second sequence. In the clause graph version this operation works very similarly: Two I-links with a common succedent and antecedent literal node may be joined into one new link if the two link substitutions merge. The joined link consists of the union of both antecedents and both succedents respectively, with the common literal node removed:

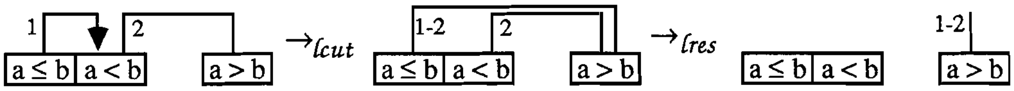


An example for an application of the cut rule:



Link Resolution with Cut Rule (*ℓrescut*)

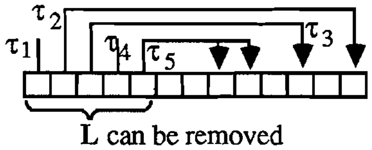
The link cut rule can be combined with the link resolution rule giving a more powerful inference rule which contains an implicit clause factoring operation and is as easy to handle as the link resolution rule itself. As the example below suggests,



F-links, i.e. I-links which are connected to one clause node only, can be treated during a link resolution like R-links, just forgetting their succedent literals. The justification for this rule is, that the succedent literal nodes can be removed applying the cut rule to the F-link and the link that is connected to the succedent literal node used in the link resolution. Another viewpoint of this rule is, that the link resolution uses instead of the original clause node the instance of the clause where the corresponding antecedent literal node of the F-link has been removed.

Subsumption Factoring (*subfac*)

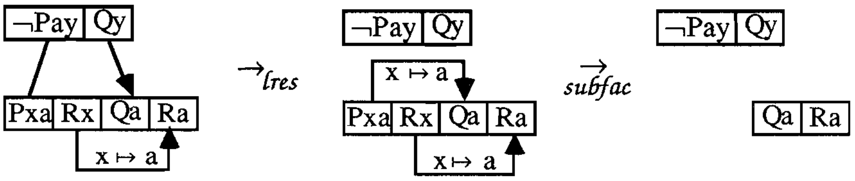
There are clauses which are subsumed by one of their own factors. Pax ∨ Paa ∨ Qx ∨ Qa for instance is subsumed by its factor {Paa, Qa} which can be generated just by removing the superfluous literals Pax and Qx from the original clause. The removal of superfluous literal nodes together with all adjacent links may considerably decrease the search space and should be performed as early as possible. Therefore this operation, called subsumption factoring, has the status of a separate inference rule, although it could be achieved with a (more expensive) resolution followed by a clause subsumption operation. Subsumption factoring is applicable if the clause can be partitioned into two parts **L** and **K** such that all literals in **L** may be removed by resolution and factoring operations without adding new literals and instantiating the literals in **K**. A subsumption factoring possibility is indicated by a group of F-links and unary R-links with a non empty common link substitution that instantiates only the antecedent literals. A typical situation of this kind looks as follows:



$\tau = \tau_1 \cap \tau_2 \cap \tau_3 \cap \tau_4 \cap \tau_5 \neq \emptyset$

τ instantiates L only

An example where link resolution makes a subsumption factoring step applicable is:



5.3. Clause Graph Reduction Rules

The clause graph inference rules defined above introduce new objects into the graph that represent information explicitly which was contained implicitly in the previous state of the graph. Older clauses and links may therefore become worthless because their information is fully contained in derived links and clauses. They should be removed as soon as possible. In this section we therefore present a number of rules for removing redundant objects from the graph. An object - a clause, a link or a link substitution - is redundant if its removal turns a graph which is refutable by r-link resolution and link factoring into a graph which is still refutable with these two rules.

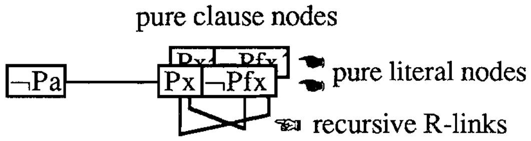
In the sequel we mean by a "refutable clause graph" a clause graph which is refutable by the r-link resolution rule and the link factoring rule.

Clause Deletion Rules

Clause Purity (*cpur*)

In the original definition of the clause purity rule, a clause can be removed, if one of its literals is not connected to any R-link. The reason is that this literal would remain part of any further resolvent with this clause, and could therefore never be used to generate the empty clause. The clause purity condition can be slightly weakened: A clause can be removed if it contains a literal that is connected at most with some recursive R-links. (Recursive R-links connect different literal nodes of different copies of a clause node.)

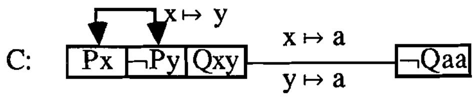
Example



Pure clause nodes can never be used in a refutation because it can be shown that the link resolution rule transforms recursive links always into recursive links, but the last usage of a clause node in a refutation must involve nonrecursive links only.

Clause Tautology (*ctau*)

A tautologous clause is true in every interpretation and can therefore be removed from a set of unsatisfiable clauses without losing its unsatisfiability. A general clause may be regarded as a representation for a certain set of its ground instances, some of those may be tautologies, some others may not. For example the ground instances of {Px,¬Py} which instantiate the variables x and y with equal terms are tautologies and can be removed, all others are not. The whole clause can be removed if no non tautologous ground instances remain. The general case, where only some ground instances can be removed makes not much sense in the basic resolution calculus because the clause still remains part of the clause set. In the clause graph environment, however, this rule can be used to enable further link deletions. Consider the following example:



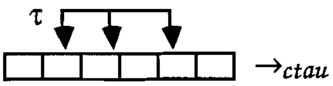
All substitutions which are instances of x ↦ y generate tautologous instances of the clause C. If these substitutions are marked to be illegal all the link substitutions of the attached links which are themselves instances of the removed substitutions become illegal too and must be removed as well. In the example above, the whole R-link must be removed because {x ↦ a, y ↦ a} is an instance of the "tautologous substitution" {x ↦ y}.

The *ctau*-rule consists of three parts: recognition - completion - deletion.

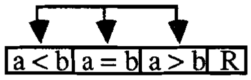
➤ Recognition

Tautologous instances of a clause are indicated by internal T-links, i.e. T-links which are connected only with the corresponding clause node. All link substitutions of such a T-link denote tautologous instances of the clause which therefore can be removed. The whole clause can be removed if the link substitution is empty.

Tautology Recognition: A typical situation, an example:



The τ instances are redudant

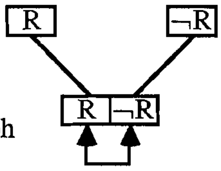


tautology

(In an actual implementation the redundant instances of a clause must be stored as a set of substitutions which is a special component of the clause.)

➤ Completion

A tautologous clause can be removed from an unsatisfiable clause set without losing the unsatisfiability. As the following example demonstrates, this does not imply, that a tautologous clause node can be removed without further provisions from a refutable clause graph without losing its refutability:



a refutable clause graph

the clause graph is no longer refutable after deletion of the tautology

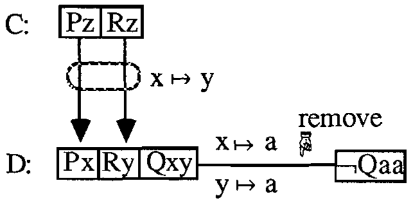
The problem is the missing "bridge link" between R and ¬R that would allow for another refutation. (This link might have been removed in previous steps by some link reduction rules.) One could either renounce on removing tautologies if such links are missing, or insert them before the clause node is removed. Since removing a clause node usually shrinks the search space much more than the insertion of some links increases it, we decided to formulate the clause tautology rule in the second way: tautology removal with link insertion. Which of the links must actually be inserted before a tautology can be removed, depends on the structure of the clause node´s T-links.

➤ Deletion:

All "tautologous" instances of the clause are taken to be illegal. If there are no remaining clause instances, the whole clause node is deleted. Furthermore the link substitutions of all adjacent links which are instances of the illegal clause instances are also removed. Links with an empty link substitution are deleted altogether.

Clause Subsumption (*csub*)

A clause of an unsatisfiable clause set C which is implied by some other clauses in C can be removed without losing unsatisfiability. A full application of this deletion rule is not only impossible because of the undecidability of this implication problem, but it is also undesirable because derived clauses may be very useful in finding the refutation. Therefore one is interested only in a restricted version where the implication can be easily detected (subsumption) and where the removal of the implied clause does not lengthen the refutation. Similar to the clause tautology rule, we do not only consider the case where a subsumed clause can be completely removed, but it is also possible to declare some instances of a clause as illegal which are subsumed by another clause. This rule can trigger further link deletions, as is shown in the next example:



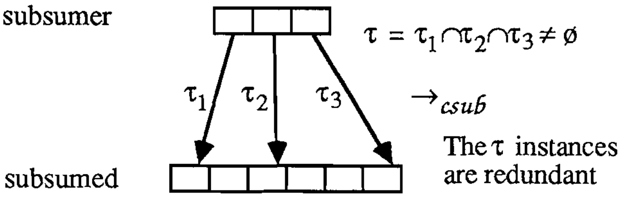
All instances of D where x and y become equal are subsumed by C.
The R-link can therefore be removed.

The clause subsumption rule consists also of the three parts: recognition - completion - deletion:

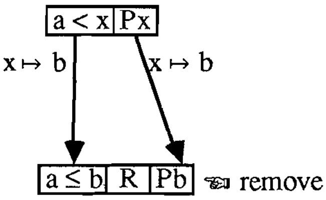
➤ Recognition:

In the environment of clause graphs, a clause subsumption situation can be easily detected using binary I-links: There must be a set of binary I-links with compatible link substitutions which map the literal nodes of the subsumer injectively to the literal nodes of the subsumed clause:

A typical situation:

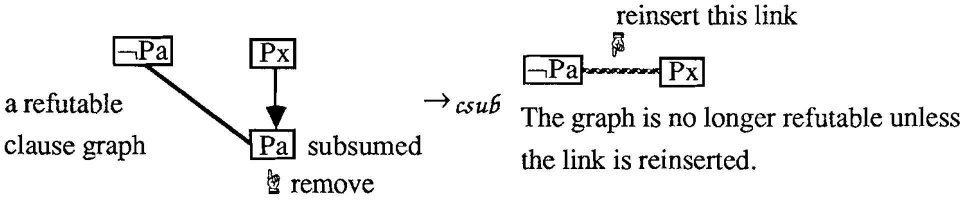

subsumer
$\tau = \tau_1 \cap \tau_2 \cap \tau_3 \neq \emptyset$
τ_1
τ_2
τ_3
$\rightarrow_{csub}$
The τ instances are redundant
subsumed

an example:



➤ Completion:

Again clause nodes which are logically superfluous cannot be removed from a clause graph without further ceremony. Due to certain link deletion rules, the subsumer may have lost some links which are necessary for a refutation without the subsumed clause node or the removed instances, respectively. These links must be reinserted before the subsumed clause node can be removed. The example below shows such a situation. Before the subsumed clause node Pa can be removed, a link between ¬Pa and Px must be inserted into the graph.



The graph is no longer refutable unless the link is reinserted.

➤ Deletion:

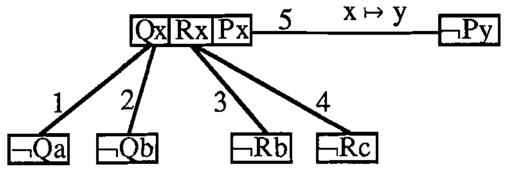
All "subsumed" clause instances are taken to be illegal. If there are no remaining clause instances, the whole clause node is deleted. Furthermore the link substitutions of all adjacent links which are instances of the illegal clause instances are also removed. Links with an empty link substitution are completely deleted.

Link Deletion Rules

A link or at least some of its link substitutions may be removed either if it can be shown that the link cannot contribute to a refutation or if there is some other link in the graph which can take over its role in a refutation.

Link Incompatibility (*ℓinc*)

The general idea of this deletion rule is to instantiate the link substitutions such that they represent only those ground substitutions which can actually be used in a link resolution. The example below illustrates the idea.

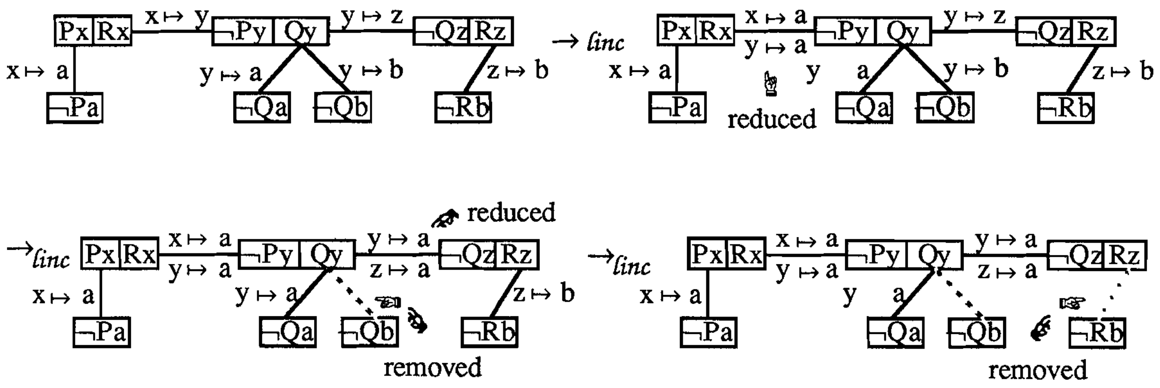


The only possible link resolution is 2,3,5 yielding a merge substitution $x \mapsto b, y \mapsto b$. The link substitution of link 5 may therefore be reduced to $x \mapsto b, y \mapsto b$.

The rule works as follows: Given a link, select one of its antecedent clause nodes as base clause node. Compute the merge substitutions of all possible link resolutions with this link and the selected base clause node. Reduce the link substitutions to the computed set of merge substitutions.

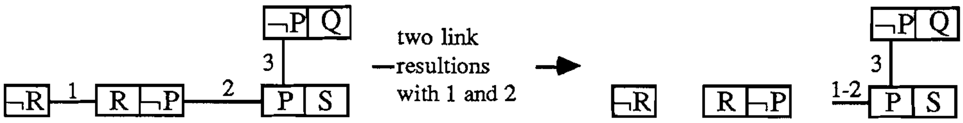
Of course this might be very expensive, but it may be used to start a constraint propagation sequence which can cause a very helpful snowball effect of further deletions.

Example Constraint propagation with the link incompatibility rule.



Parallel Link - Link - Subsumption (*pℓsub*)

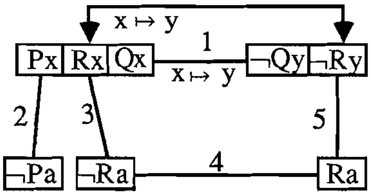
Consider the following situation:



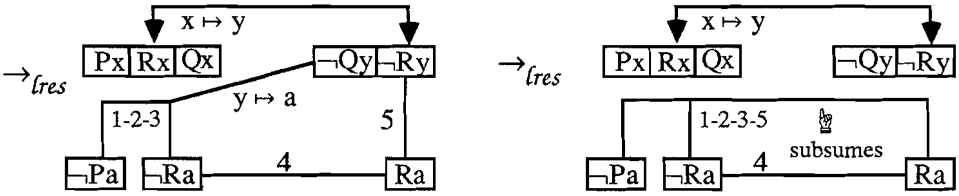
The combined link 1-2 contains the information that the literal P must be false, whereas the link 3 contains only the information that P and ¬P are contradictory. Link 3 is subsumed by link 1-2 and can therefore be removed without losing information. The *pℓsub* rule allows the deletion of any link which is "larger" than another one, i.e. its antecedent and succedent literal nodes are supersets of the antecedent and succedent literal nodes of a "parallel" link and its link substitution is an instance of the link substitution of the parallel link.

Link Tautology (*ltau*)

The link tautology rule is a consequent extension of the clause tautology rule. Each link represents a resolvent and if this resolvent has tautologous instances which can be removed, these instances should already be removed from the link substitution. Consider for instance the following graph:

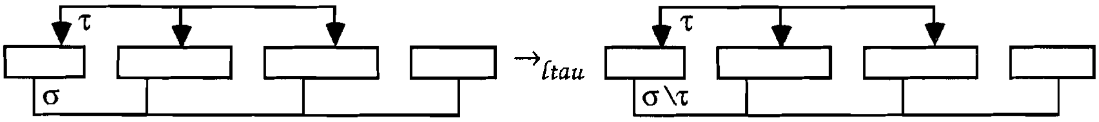


The link 1 represents a tautologous resolvent and is therefore useless. Since resolution is not the main operation of our procedure, we need another justification for such a removal. And in fact, link resolutions involving the corresponding instance of the link substitution of link 1 and its successors generate a combined link which is subsumed by the "bridge link" 4:



This effect is no coincidence and "tautologous" instances can be removed from link substitutions - provided the graph contains enough "bridge links".

Link tautologies can easily be recognized using T-links which are parallel to the considered R-link. A typical situation is shown below.



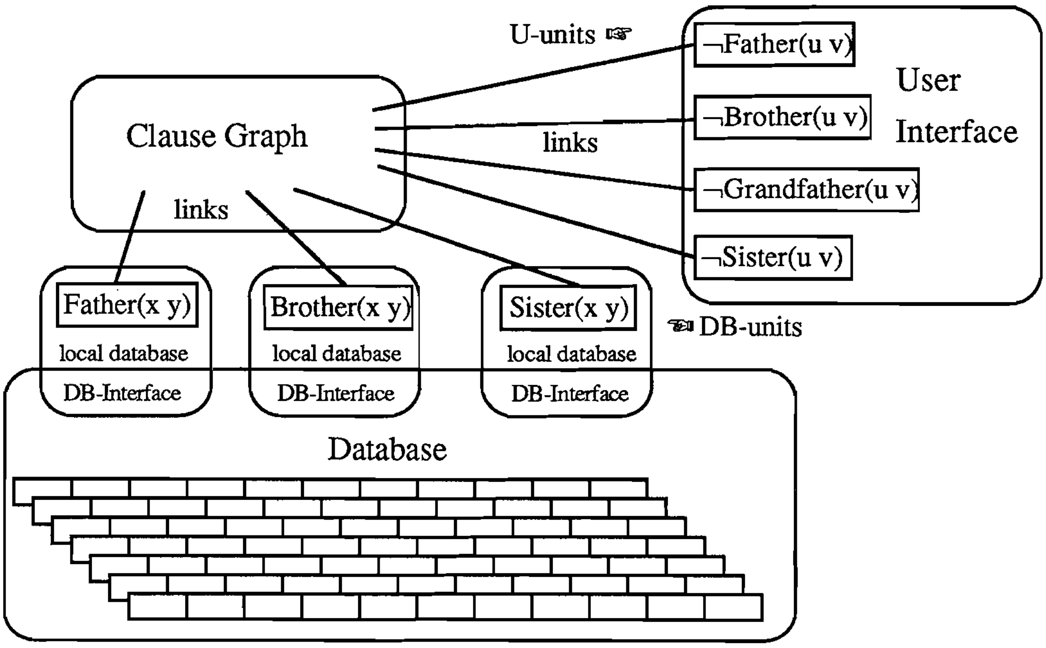
This terminates our list of operations on clause graphs. Inventing further useful operations is subject to ongoing research.

6. Using Clause Graphs for Supporting Database Query Evaluation.

In order to apply clause graphs in a deductive database environment we need three additional components:

1. There must be an interface between the clause graph and the database such that unit clauses need not be part of the graph, but can be accessed via this interface. For every relation in the database we assume such an interface that can be viewed from the graph as one additional unit clause (DB-unit) which is unifiable with every literal in the database with the corresponding predicate. Such a DB-unit is a representative for all database entries with the corresponding relation and must therefore actually be a clause node bunch, i.e. a source for arbitrary many copies of the clause.
2. There must be a "local database" which temporarily stores some deduced unit clauses.
3. There must be a user interface for submitting the queries to the system. The user interface must also be viewed from the graph as some additional negated clauses (U-units). The U-units are representatives for all potential queries to the system.

An example: Assume there are three relations "brother", "sister" and "father" in the database and an additional predicate "grandfather" in the clauses. The initial state of the system looks as follows:



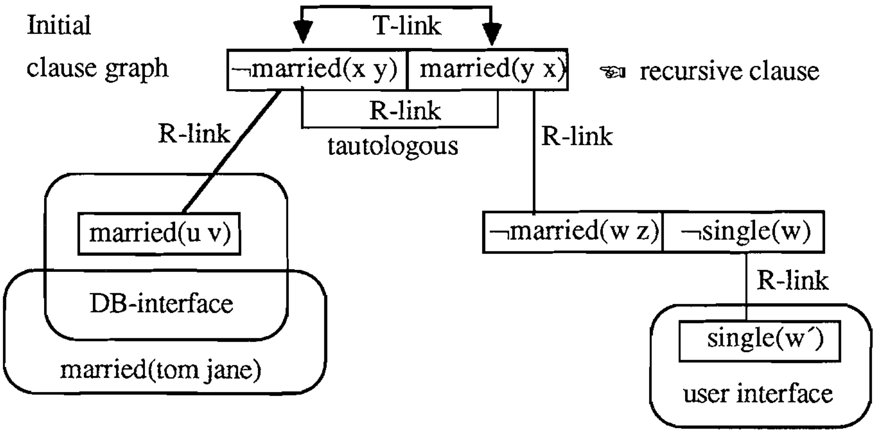
The U-units ¬Father(u v) etc. represent all negated queries that can be submitted to the system, i.e. the user can actually ask "who is the father of whom?". If also negative queries are to be allowed, i.e. queries "who is *not* the father of whom" then also positive U-units "Father(x y)" must be added to the unser interface.

6.1 Initial Optimizations

An initial clause graph may be optimized using the clause graph inference rules and reduction rules in the following way:

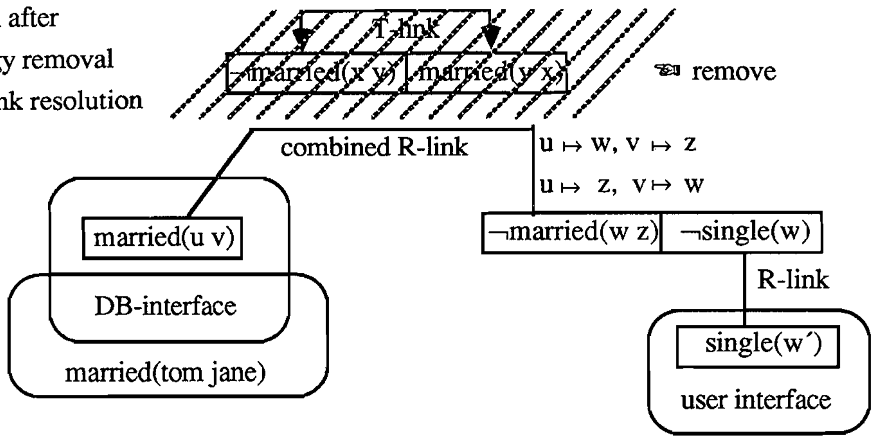
1. Apply the reduction rules for clauses and links:
   - clause purity
   - clause tautology
   - clause subsumption
   - link incompatibility
   - link tautology
   - parallel link - link subsumption
2. For all clauses without internal links (links to copies of the clause):
   - Apply the link resolution rule to all combinations of links connected to this clause. Remove the clause afterwards.
   - Apply again the reduction rules.
   This may cause further deletions of internal links and therefore make step 2 applicable to new clauses.
3. Create new resolvents only when enough reduction operations are applicable to the new graph such that the final graph is smaller in size than the original one.

As an example consider a graph containing the symmetry axiom for the predicate "married".

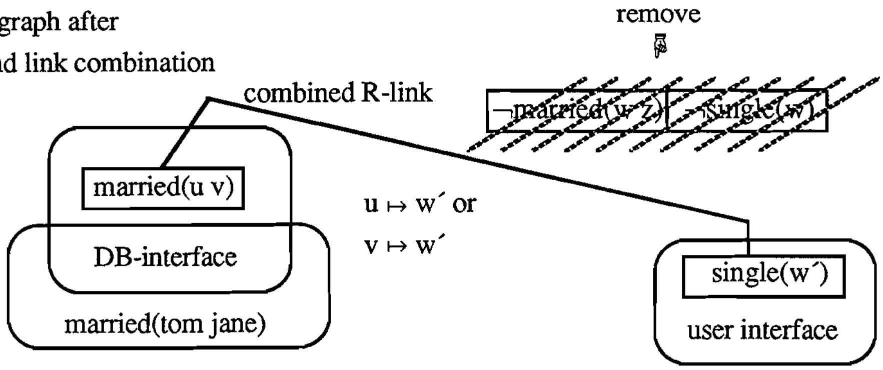


The link tautology reduction rule recognizes and removes the tautologous R-link. That means the system recognizes that the recursive symmetry axiom need not be applied recursively because recursive application reproduces the original result. After removing this tautologous internal link, the link resolution rule can be applied to both clauses.

clause graph after
link tautology removal
and a first link resolution



clause graph after
a second link combination

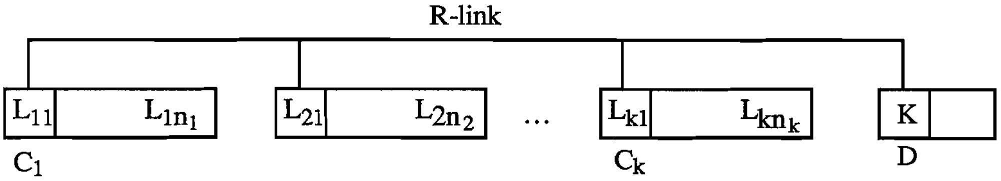


A negated user query "single(jane)?" can now be directly translated into an access "married(u,jane)?" to the database. The two clauses are no longer necessary and may be removed.

It is noted that in case the clause set contains no recursive predicates, it is always possible to transform the initial clause graph into a graph containing only R-links between the U-units and the DB-units. That means an actual query can be immediately translated into a database access. The answers of the database can immediately be transformed into answers of the query. No further inferences are necessary.

6.2 The Run Time System

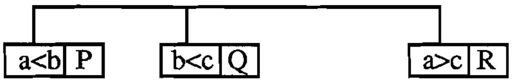
A query can be evaluated using the procedural view of the optimized SL-resolution as proposed in chapter 3 and using the links in the optimized clause graph as datapaths that transfer tasks into subtask and collect the answers. The basic idea for using an R-link as datapath for tasks is as follows:
Assume an R-link connects a clause D with k clauses $C_1,\ldots,C_k$ as indicated below



(Let us assume for the moment that all clauses are ground.)
Assume further the current task is K, i.e. K is to be refuted. Since the literals connected by the R-link are contradictory, this task can be transformed into the k queries $L_{11},\ldots,L_{k1}$, i.e. $L_{11},\ldots,L_{k1}$ are to be *proved*. Since clauses are disjunctions of literals, each of these queries L_{i1} can then be transferred into the tasks to *refute* $L_{i2},\ldots,L_{in_i}$ and the process continues with the links adjacent to the literal nodes L_{ij}

until the DB-interface has been reached and a query can be answered directly. When for a given query L_{i1} the corresponding subtasks succeeded, L_{i1} has been proved and as a side effect L_{i1} can be added as a lemma to the local database. Furthermore when all queries $L_{11},\ldots,L_{k1}$ have been answered in this way, the literal K has been refuted and therefore $\neg$K must hold and can also be added as a lemma to the loca database.

As a concrete example assume the current task is to refute a>c and the situation in the graph is as follows:

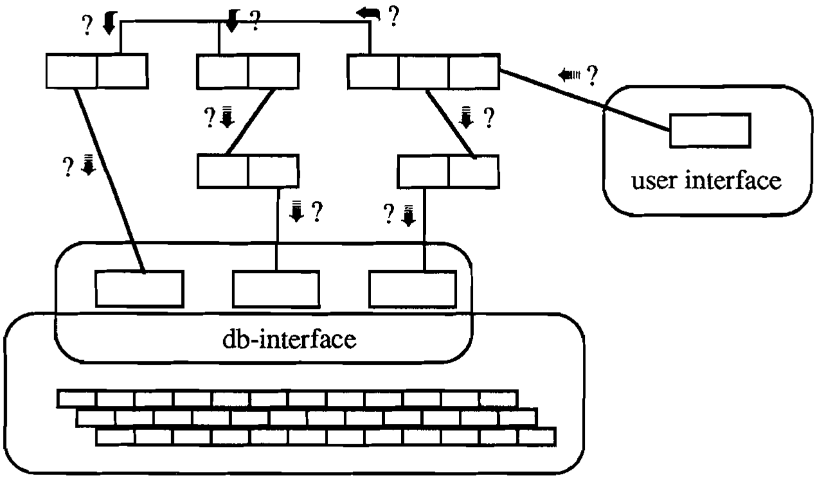


The R-link transfers this task a>c into the two queries a<b and b<c. These two queries are then transferred into the corresponding tasks to refute P and Q. As soon as P and Q have been refuted, a<b, b<c and $\neg$a>c have been proved and can be added as lemmata.

In the non-ground case when a link substitution τ is attached to the R-link, a task σK is transferred into the k queries $(\sigma\cap\tau)L_{11},\ldots, (\sigma\cap\tau)L_{k1}$. Each of these queries $(\sigma\cap\tau)L_{ki}$ is then transferred into the tasks to refute $(\sigma\cap\tau)L_{i2},\ldots, (\sigma\cap\tau)L_{ini}$.

The factoring and ancestor resolution operations which are necessary in the non Horn case are indicated by I-links with nonempty consequence literals. When there is such an I-link pointing to a literal node which is already processed in the current search path, these operations are realized by simply stopping the splitting of the current task into subtasks and considering the current task (or corresponding instances respectively) to be solved already. The justification is that a variant of the current task is already processed in the search path that lead to the current task and each answer to this task is also an answer to the current task.

A typical dataflow during query evaluation is visualized in the figure below. The arrows denote the direction of the splitting of tasks into subtasks. In the opposite direction the answers of the subtasks are collected and combined to answers of the supertasks.



7. A Prototype Implementation

In this section an implementation of a deductive query evaluation mechanism is described which, however, is currently still *restricted to Horn clauses* and does *not yet use clause graphs* as the basic datastructure. (To read this chapter, it is therefore best to forget all previous paragraphs after 3.1.2, SL-resolution for Horn clauses.) The main purpose of this implementation is to test the optimizations of SL-resolution such as lemma generation, query linking etc. and the selection strategies on nontrivial examples. Furthermore the usefulness of advanced implementation techniques like object oriented programming, demons etc. was to be evaluated.

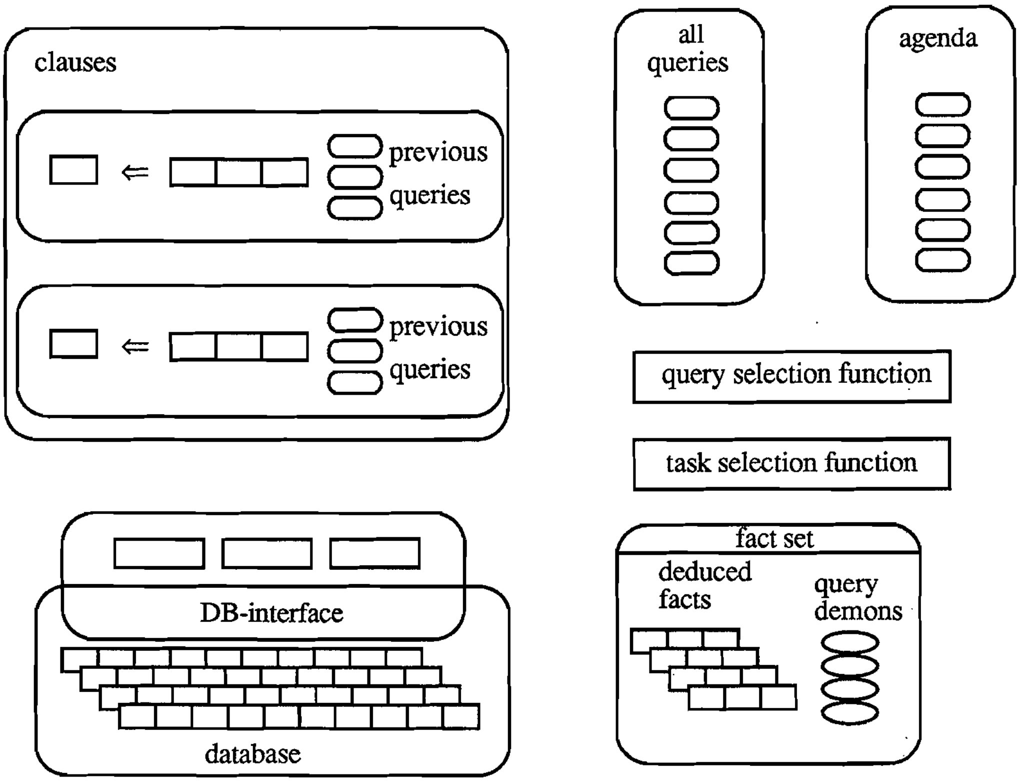
7.1 Object Oriented Programming

The basic programming paradigm we used to implement the deductive database retrieval mechanism is object oriented programming. Since the control flow in the system is extremely data driven, the object oriented view is in this case a natural and elegant means to describe its working.

In most object oriented programming languages, there are object classes, objects, methods and messages. An object class is a datastructure definition in the usual sense, i.e. a description of the objects to be manipulated and the algorithms, usually called *methods*, working on these objects. What is specific to the object oriented approach is the fact that an algorithm realized as a method is not a function that gets an object belonging to the given class (together with other parameters) as an argument and reacts in some way, but the piece of code representing the method is attached to the object itself and activated by sending a "message" to this object. Therefore an object, as an instance of an object class, is an n-tuple $(a_1,\ldots,a_n)$ of internal values together with a set of methods which are identified with method names. Method names must be unique only within one object class. That offers the chance to define object classes sharing the names of methods which do different, but in some more abstract sense analogous things, with other object classes. Typical examples are streams, i.e. data channels. For instance one stream class may be linked to terminals, another one to files on a disk. Both have to react on a request to print something with completely different actions, but the program that wants to print need not know anything about the structure of the stream. It only sends a "print" message with the string to be printed as an argument, and the overall control mechanism guarantees that the particular print method of the stream the message was addressed to is applied to the argument.
(We did not mention the inheritance mechanisms which are part of most object oriented programming languages because they are not necessary for the rest of the paper.)

7.2 The System Components

The main components of the system are:



Datastructures

The Database Interface

We assume the existence of an interface to a relational database that answers to a query in form of a positive literal by returning all instances of this literal in the database. A query might be for example Loves(John x)? and a possible answer might be {Loves(John Mary), Loves(John Sue)}. More complicated queries, like for instance Loves(John x) ∧ Loves(x Tom)? do not occur. (It is only a technical problem to extend the inference mechanism such that also queries of this kind may occur which can exploit the query optimization facilities of the database)

Term and Termlist Stores

A term or termlist store is very similar to a relational database. The - non-ground - terms in a term or termlist store, however, are indexed such that for a given term (or termlist) s there is fast access to all instances of s, all terms which are unifiable with s, all terms which are renamed variants of s and all terms which are more general than s. Operations like "insert s if it is not an instance (renamed variant) of an already existing term in the store" or "remove all terms which are instances of s from the store" etc. are available. It is noted that atoms are also represented as terms and therefore they can be inserted into a term store as well. Term and termlist stores are used at various places in the deductive database retrieval mechanism to store temporary information.

It is beyond the scope of this paper to describe the internal structure of these stores and the algorithms working on them.

Clauses

The object class "clause" represents Horn clauses with some additional information attached to them. A clause basically consists of four components:

head-atom
body-atoms
variables
query codomain termlist store.

The component "variables" is a list of variables occurring in the clause. The ordering of the variables in this list is arbitrary but fixed.

The component "query codomain termlist store" is a termlist store and stores information about previous activations of the clause in order to prevent superfluous activations of the clause as follows: When the clause is requested to deduce σ-instances of the head-atom for some substitution σ, then the σ-instance of the component "variables", which is just the codomain of σ restricted to the clause's variables is inserted into the query codomain termlist store.

An example:

Consider the clause $C := P(f(x)\ z) \Leftarrow P(x\ y) \wedge P(y\ z)$

The representation of C as a "clause" object is

head-atom	=	$P(f(x)\ z)$
body-atoms	=	$\{P(x\ y),\ P(y\ z)\}$
variables	=	$(x\ y\ z)$

The initial query codomain termlist store is empty.

When for example a request to deduce σ-instances with $\sigma := \{x \mapsto a, z \mapsto g(v)\}$, i.e. instances of $P(f(a)\ g(v))\}$, is sent to the clause, then the termlist $\sigma(x\ y\ z) = (a\ y\ g(v))$ is inserted into its query codomain termlist store. Another request with $\sigma' = \{x \mapsto a, z \mapsto g(b)\}$ which is an instance of the previous one can then be rejected because $\sigma'(x\ y\ z) = (a\ y\ g(b))$ can easily be recognized as an instance of $(a\ y\ g(v))$. (The overall control mechanism ensures that the answers to the more general request are automatically forwarded to the rejected request.)

Fact Set

The object class "fact set" is used to store deduced unit clauses. It consists of the two components:

fact term store
query demons

The first component is a term store which actually contains the atoms. The second component is the list of literal queries (see object class "literal query" below) waiting for new facts to be inserted into the fact set.

Queries

We distinguish two types of queries *literal queries* and *clause queries.*

Literal Queries

An object of this class represents a request to deduce instances of a given atom wherever it is possible. Since a query has various relations to other objects, it consists of a whole bunch of components:

clause
atom
general substitution
instantiated substitution
sibling queries
more special queries
more general query
previous answers

For illustrating the components of a literal query with examples, we refer again to the clause

$C := P(f(x)\ z) \Leftarrow P(x\ y) \wedge P(y\ z)$

Assume C has been requested to deduce σ-instances of the head literal $P(f(x)\ z)$ with $\sigma := \{x \mapsto a, z \mapsto g(v)\}$. This causes the generation of two literal queries Q_1 and Q_2, derived from the σ-instances $P(a\ y)$ and $P(y\ g(v))$ of the two body atoms of C. As soon as for example Q_1 is told that the new fact $P(a\,b)$ has been derived, Q_2 will be instantiated to the query $Q_{2'}$ for the atom $P(b\ g(v))$ such that the answers to $Q_{2'}$ are compatible with the answer $P(a\,b)$ to Q_1.

"Clause" is the clause that caused the activation of the query.
"Clause" for Q_1, Q_2 and $Q_{2'}$ is the object representing the clause C.

"Atom" is the original atom in the requesting clause the query is an instance of.
Atom of Q_1 is $P(x\ y)$ and
atom of Q_2 and $Q_{2'}$ is $P(y\ z)$.

"General substitution" gathers the combined substitution for the UR-resolution step. It is represented as a codomain termlist with the variables of the clause being the domain termlist.
When the variables of the clause object representing C are (x, y, z) then
general substitution of Q_1 and Q_2 is (a, y, z) and
general substitution of $Q_{2'}$ is (a, b, z).

"Instantiated substitution" gathers the combined substitution resulting from the *initial substitution* given to the requesting clause and the answers to parts of the sibling queries. It is also represented as a codomain termlist with the variables of the clause being the domain termlist.
When the variables of the clause object representing C are (x, y, z) then
instantiated substitution of Q_1 and Q_2 is (a, y, g(v)) and
instantiated substitution of $Q_{2'}$ is (a, b, g(v)).
It is noted that the literal which represents the actual query is the "Instantiated substitution"-instance of atom.

("Instantiated substitution" is used to ensure that only those subqueries are generated which answer the original query whereas "general substitution" is used to deduce the full UR-resolvent as explained in the "triggered UR-resolution" section of chapter 3.)

"Sibling queries" is the list of queries that are still to be served in order to enable a full UR-resolution step.

Sibling queries of Q_1 is $\{Q_2\}$ and
sibling queries of Q_2 is $\{Q_1\}$ and
sibling queries of $Q_{2'}$ is $\{\}$.
(The information about the answer P(a b) to Q_1 that caused the instantiation of Q_2 to $Q_{2'}$ has been gathered in the general and instantiated substitutions of $Q_{2'}$.)

"More special queries" is a list of other queries whose instantiated atom is an instance of the current query's instantiated atom. Answers to the current query are automatically forwarded to the queries in this list. In our example, $Q_{2'}$ is <u>not</u> in the "more special queries" list of Q_2 although $Q_{2'}$'s instantiated atom is an instance of Q_2's. This has to do with the fact that Q_2 will never be activated itself, but only used to generate instances fitting the answers of Q_1. See below for more details.

"More general query" contains the query from which the current query gets its answers forwarded, if there are some. If a query Q_i contains a query Q_j in its "waiting more special queries" then Q_i is the "more general query" of Q_j. This link between queries is necessary when a query is killed.

"Answers" is a list which gathers all answer atoms to the query. This list must be available when a query Q_i has gathered some answers and at a later time another query Q_j is generated and linked to Q_i telling Q_i to forward all its answers to Q_j. Since Q_j never receives information from the database or the fact set directly, this is the only channel to get previously retrieved facts.

The *instantiated atom* of a query, i.e. the actual atom that represents the query can be computed by applying the instantiated substitution to the query's atom. Therefore it need not be explicitly stored.

Clause Queries

A clause query represents a particular request to a particular clause.
The components of a clause query are:

clause
substitution
requesting query

"Clause" is a clause object.

"Substitution" is the restricted codomain of the unifier for the head atom of the clause and the instantiated atom of the requesting query. The variables of the clause represent the corresponding domain of the substitution.

"Requesting query" is the query that caused the activation of this clause.

Query Store

In order to find queries which can be linked to a given query, i.e. which ask for a more general atom we must store all relevant queries such that for a given query there is fast access to all more general queries. We use a term store that stores for each query its instantiated atom and attaches the query itself as an associated property. (Term stores may attach properties to each term.)

Initial Query

An object of this type represents the initial query to the system. Its components are:

body atoms
answer substitutions

"Body atoms" is just the list of atoms of the query clause.
"Answer substitutions" gathers the set of final answer substitutions.
An initial query is in principle nothing else than a literal query. It is however responsible for counting the number of generated answers and aborting the process when - according to a user specification - enough answers have been generated.

The Agenda

The agenda is a set of queries.

7.3 Algorithms

Our approach is data driven, that means, once activated the system acts by creating objects and exchanging messages between objects. Control is therefore local to each object which decides for itself what to do next. There is only a rudimentary global control structure consisting of the select - remove - activate loop of the agenda. The behaviour of the whole system can therefore only be described by presenting the set of methods associated with each object class.

The Database Interface

We need only the method "retrieve(atom)" returning the set of atoms in the database which are instances of its atom.

Term and Termlist Stores

We need methods

insert (s)	inserts s into the store
insert if no variant (s)	inserts s if no renamed variant of s is in the store
insert if not subsumed (s)	inserts s if there is no more general term in the store
insert with backward subsumption(s)	inserts s if it is not subsumed and removes all t´s which are subsumed by s
remove s from the store	

return s if it is in the store (together with its associated properties)
return the first or all variants of s
return the first or all instances of s
return the first or all more general terms than s.

Clauses

Method :activate (query)
This method checks if the clause may satisfy the query, i.e. if the head atom is unifiable with the instantiated atom of the query and if there is no previous more general activation of the clause. In the positive case a clause query is generated and inserted into the agenda. The query´s instantiated atom is inserted into the clause´s query codomain termlist store.

Method :new-substitution (substitution-codomain)
A :new-substitution message tells the clause that a UR-resolution step can be performed because a simultaneous unifier for the body atoms and appropriate unit clauses has been computed. Substitution-codomain is the part of the unifier, together with the clause´s variables as the corresponding domain which is relevant to the clause. (Substitution components of variables in the used unit clauses are irrelevant.) The clause generates the new unit clause by instantiating its head atom and sends it to the fact set.

Fact Set

Method :insert (atom)
An :insert message tells the fact set that a new unit clause has been deduced and can be stored for later use. The atom is inserted into the fact term store with forward and backward subsumption test. That means it is just ignored if a more general atom is already in the fact term store. On the other hand, if it is not ignored all its instances in the fact term store are eliminated.
Finally if the atom is successfully inserted into the fact term store, a message :new-fact (atom) is sent to all waiting queries in the fact-set´s query demons list. All demon queries responding that they are fully satisfied now are removed from the query demons list.

Method :retrieve (query)
A :retrieve message is a request to send all atoms in the fact set which are unifiable with the query´s instantiated atom to the query. If the query does not respond at least once that it is fully satisfied now, it is inserted into the fact set´s query demons list. The :retrieve message responds with a flag indicating whether the query has been fully satisfied or not.

Literal Queries

Method :activate

The :activate message is sent by the agenda to the selected query. It is processed as follows:

1. A :retrieve message is sent to the fact set
   (which sends all suitable unit clauses to the query and installs the query as a query demon.)
2. If the query has not been fully satisfied
   a :retrieve message is sent to the database (which also sends all suitable unit clauses to the query.)
3. If the query still has not been fully satisfied
   :activate messages are sent to all clauses with unifiable head atoms.
   If there is no such clause the query is killed (thus removed as query demon from the fact set.)

Method :new-fact (atom)

The atom is renamed, i.e. each variable is replaced by a new one, and unified with the query´s instantiated atom. If it is not unifiable nothing else happens.

If the new atom can be accepted there are two main cases to be considered:

1. If the query has no further sibling queries this means that the atom is the last one to complete an UR-resolution step. The atom is unified with the query´s original atom, i.e. the corresponding body atom of the original clause that generated the query. The unifier is applied to the query´s general substitution yielding the final substitution for the UR-resolution step. This step is executed by sending a :new-substitution message with the final substitution as argument to the query´s requesting clause.

2. If there are still sibling queries they are now instantiated by copying the old queries, replacing the general substitution and the instantiated substitution with their corresponding instances, and removing the current query from the list of sibling queries of the instantiated queries. The new bunch of sibling queries must be installed now by the function install-sibling-queries (see below).

Finally the atom is forwarded to all more special queries and then inserted into the query´s answers list. If the query´s instantiated atom is an instance of atom then the query is fully satisfied. It kills itself and the answer to the :new-fact message is "fully-satisfied" in this case.

We illustrate the method with an example.

Consider again the clause $C := P(f(x)\ z) \Leftarrow P(x\ y) \wedge P(y\ z)$ with a "variables" list (x y z).

Assume the clause has been requested to deduce instances of P(f(a) g(v)). Initially it generates the two related queries Q_1 and Q_2. Their components are:

Q_1:	atom:	P(x y)	Q_2:	atom:	P(y z)
	sibling queries	Q_2		sibling queries	Q_1
	general substitution:	(x y z)		general substitution:	(x y z)
	instantiated substitution:	(a y g(v))		instantiated substitution:	(a y g(v))
	(instantiated atom:	P(a y)		instantiated atom:	P(y g(v)))

A message :new-fact (P(a b)) sent to Q_1 is processed as follows:

P(a b) is unified with P(a y) yielding the unifier$\{y \mapsto b\}$, i.e. the atom can be accepted. Since Q_1 has still sibling queries, Q_2 must be instantiated as follows:

$Q_{2'}$:	atom:	P(y z)		
	sibling queries	{ }		
	general substitution:	(a b z)	(The unifier of P(a b) and P(x y) applied to the general substitution of $Q_{2'}$.)	
	instantiated substitution:	(a b g(v))	(instantiated atom:	P(b g(v)))

A message :new-fact (P(b w)) sent to $Q_{2'}$ is processed as follows: P(b w) is renamed to P(b w´). (This is in general necessary to avoid variable conflicts when the same literal can be used to answer several related queries.)
Since P(b w´) is unifiable with P(b g(v)) and there are no sibling queries, the unifier of P(b w´) with the original atom P(y z) is applied to the general substitution (a b z) yielding (a b w´). This substitution codomain is sent to the requesting clause which can now generate the lemma P(f(a) w´).

Method :link (more-special-query)
The more-special-query is inserted into the query´s "more special queries" list.
All atoms in the answers list are sent to the more-special-query with the :new-fact message.

Method :kill
A query is killed either if it is fully satisfied or if there is no unifiable clause head.
Killing a query means deleting it as well as all its sibling queries and all its more special queries. Furthermore the killed queries must be removed from all the relevant lists.

Function install-sibling-queries (queries)
This function selects one of the queries and usually it inserts it into the agenda.
First of all it is tested if one of the queries can be linked to a more general query by scanning the query store for a more general query and sending a :link message. If no linking is possible, the **query selection function** is applied to select the query to be activated first. Usually the "most instantiated" query is selected. In most examples this is the query with fewest number of variables in its instantiated atom. For deeply nested terms, however, the nesting depth must also influence the selection. In case **query generalization** is activated and the selected query Q exceeds the corresponding limit, a generalized query QG is created by replacing the constants in the selected query´s instantiated atom with variables. Q is linked to QG and QG is inserted into the agenda and the query store. When no query generalization is possible, Q itself is inserted into the agenda and the query store.

Clause Queries

Method :activate
A bunch of sibling queries is generated from the body atoms in the clause query´s clause. The clause query´s substitution codomain becomes their combined substitution codomain. They are installed with install-sibling-queries. Activating a clause query therefore means to link a new query to an old one or to insert a new query into the agenda. Nothing more happens.

Query Store

Since the query store is just a term store, all operations on term stores are available.

Initial Query

Method :activate
This method is identical with the :activate method for clause queries except that the combined substitution codomain for the new queries is the identity.

Method :new-substitution (substitution-codomain)
This method is similar to the :new-substitution method for clauses. It tells the query that all queries generated form the query clause´s atoms have succeeded in deducing compatible unit clauses. Therefore substitution-codomain with the query clauses´s variables as the corresponding domain represents an answer substitution. The method gathers the answer substitutions and terminates the process depending on the user´s requirements.

Agenda

Besides the functions for inserting and removing tasks there is one function that performs the overall select-remove-activate loop. A query is selected according to the **task selection function**. The task selection function computes for both literal queries and clause queries a numerical value by combining some characteristic values of the query with heuristic parameters as follows:
heuristic-value :=

	c_{siblings}	* number of related queries (or number of body atoms respectively)
+	$c_{\text{variables}}$	* number of variables in the instantiated atom
+	c_{depth}	* depth in the search tree
+	c_{linking}	* number special queries linked to the current one (0 for clause queries.)

The query with smallest heuristic value is selected.

Function Query (clause maximum-number-of-answers)

This is the toplevel function which submits a user query to the system. The system is started as follows:

1. From the clause a query clause object is generated.
2. An :activate message is sent to this query clause (which puts one query into the agenda.)
3. Control is passed to the agenda loop function.

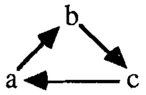
(The system is stopped inside the :new-substitution method of the query clause.)

Examples for a Query Evaluation

Example 1

A simple example will illustrate the control flow during the query evaluation.
Assume the database contains the following cyclic relations between the three objects a,b,c
P(a b), P(b c), P(c a)

Graphically



Assume further we have only one clause expressing the transitivity law for P:

$$D := P(x\ z) \Leftarrow P(x\ y) \wedge P(y\ z)$$

Query : We ask P(a x)?

1. An initial query IQ = P(a x)? is generated.

2. From the single body atom a literal query LQ_1 = P(a x)? is generated and inserted into the agenda and the query store.

3. The agenda has no other choice but selecting it and sending it an :activate message.

4. The :activate message is processed by LQ_1 as follows:

4.1 LQ_1 sends a :retrieve message to the fact set which is empty.
The :retrieve method inserts Q_1 as a query demon.

4.2 LQ_1 sends a :retrieve message to the database which reacts with a :new-fact P(a b) message sent to LQ_1.

4.2.1 LQ_1 forwards this solution to IQ which records x ↦ b as a first answer.

4.3 LQ_1 sends an :activate message to the clause D.
D generates a clause query CQ_1 = P(a x)? and inserts it into the agenda.

This is the last action of the :activate message sent to LQ_1.

5. The agenda selects the single clause query CQ_1 sends an :activate message to it.

6. The clause query generates the two sibling queries LQ_2 = P(a y)? and LQ_3 = P(y x)?.
LQ_2 is an instance of LQ_1 and therefore linked to LQ_1.

6.1 The first answer P(a b) to LQ_1 is sent to LQ_2 which instantiates its sibling LQ_3 to $LQ_{3,1}$ = P(b x). $LQ_{3,1}$ is inserted into the agenda and the query store.

This is the last action of the :activate message sent to CQ_1.

7. The agenda selects $LQ_{3,1}$ and sends an :activate message to it.
(This is again the only task in the agenda.)

7.1 $LQ_{3,1}$ sends a :retrieve message to the fact set which is still empty.
The :retrieve method inserts $LQ_{3,1}$ as a query demon.

7.2 $LQ_{3,1}$ sends a :retrieve message to the database which reacts with a :new-fact P(b c) message sent to $LQ_{3,1}$.

7.2.1 $LQ_{3,1}$ has no more siblings, therefore it sends a :new-substitution message to the clause D.

7.2.2 D generates the new lemma P(a c) and inserts it into the fact set.
This new fact is sent to all query demons.

7.2.3 The query LQ_1 receives the message :new-fact P(a c) and forwards it to IQ which records x ↦ c as a second answer.

7.3 $LQ_{3,1}$ sends an :activate message to the clause D.
D generates a clause query CQ_2 = P(b x)? and inserts it into the agenda.

The system closes the cycle by generating the two further queries P(c x)? and P(a x)? The first one gives the last answer x ↦ a. The last one is again an instance of LQ_1 and therefore linked to LQ_1. All answers are subsumed by previous ones and therefore the process stops.

Example 2

The last example is taken from lattice theory and is known as SAM´s lemma in the theorem proving literature [GOBS 69]. Although lattice theory is surely not a typical application for deductive database, it is an interesting test example because the structure of the clauses is not atypical and the difficulty is their terrible recursivity and its nondeterminism, i.e. there is no way to arrange the clauses without forcing a PROLOG strategy into infinite loops. Since it is a rather hard problem, not because of the structure of the relations in the database but because of the clause structure, it might be a good supplement to the frequently stressed "same generation" problem. (Currently only a few theorem provers can solve SAM´s lemma.)

Facts:

min(a b c)	max(c d 1)	min(b d e)	min(a e 0)
max(b a2 b2)	max(a b2 1)	max(a b c2)	min(a2 c2 0)
min(d a d2)	max(a2 d2 e2)	min(d b a3)	max(a2 a3 b3)
$\forall x\,\text{max(1 x 1)}$	$\forall x\ \text{max(x x x)}$	$\forall x\ \text{max(0 x x)}$	$\forall x\ \text{min(0 x 0)}$
$\forall x\,\text{min(x x x)}$	$\forall x\ \text{min(1 x x)}$		

Clauses

$\forall x,y,z$	$\text{min(x y z)} \Leftarrow \text{min(y x z)}$
$\forall x,y,z$	$\text{max(x y z)} \Leftarrow \text{max(y x z)}$
$\forall x,y,z$	$\text{max(x z x)} \Leftarrow \text{min(x y z)}$
$\forall x,y,z$	$\text{min(x z x)} \Leftarrow \text{max(x y z)}$
$\forall x,y,z,u,v,w$	$\text{min(w x z)} \Leftarrow \text{min(u v w)} \wedge \text{min(v x y)} \wedge \text{min(u y z)}$
$\forall x,y,z,u,v,w$	$\text{max(w x z)} \Leftarrow \text{max(u v w)} \wedge \text{max(v x y)} \wedge \text{max(u y z)}$
$\forall x,y,z,u,v,w$	$\text{max(v y z)} \Leftarrow \text{min(u v v)} \wedge \text{max(w v x)} \wedge \text{min(u w y)} \wedge \text{min(u x z)}$
$\forall x,y,z,u,v,w$	$\text{min(u x z)} \Leftarrow \text{min(u v v)} \wedge \text{max(w v x)} \wedge \text{min(u w y)} \wedge \text{max(v y z)}$

Query:

min(b3 e2 a2)?

Our prototype implementation answers it in about 30 seconds CPU time on a Symbolics 3640 machine. The statistical values are:

866 literal queries generated, 581 of them linked.

322 tasks selected by the agenda.

1021 facts deduced, 891 of them subsumed, 130 facts remaining.

130 accesses to the database.

8. Summary

We have shown how to combine the advantages of several approaches to the query evaluation problem in deductive databases: The SL-resolution strategy has the advantage to work in a query - subquery oriented way backwards from the query to the facts and can therefore be strongly focused on the relevant data. Using this strategy for triggering the "right" forward deductions instead of generating the resolvents directly has the advantage that intermediate results can be reused several times. In some cases this can be sufficient to shrink an infinite search tree to a finite subtree, i.e. to enforce the termination of the query evaluation process. For the non Horn case we have shown how to use clause graphs as an efficient indexing mechanism which guides the dataflow during query evaluation. Inference operations and reduction operations on clause graphs can be used to optimize the data access paths at "compile" time.

A first prototype implementation, although still restricted to Horn clauses, shows a very good goal directed behaviour and solves already nontrivial problems.

Acknowledgements
We wish to thank Norbert Eisinger for many fruitful discussions about clause graphs. He and Axel Präcklein read the manuscript and gave useful hints.

References

BR86 Bancilhin, F., Ramakrishnan R., *An Amateur's Introduction to Recursive Query Processing Strategies.* MCC, University, Austin, Texas (1986).

BB87 Bläsius, K.H., Bürckert, H.J., *Deduktionssysteme.* Oldenbourg Verlag, München 1987.

Bu83 Bundy, A., *The Computer Modelling of Mathematical Reasoning.* Academic Press, London, New York (1983).

CL73 Chang, C.-L., Lee, R.C.-T., *Symbolic Logic and Mechanical Theorem Proving,* Computer Science and Applied Mathematics Series (Editor Werner Rheinboldt), Academic Press, New York (1973).

Ei86 Eisinger, N., *What you always wanted to know about clause graph resolution.* Proc. of CADE 1986 Oxford, England (1986).

GOBS 69 Guard, Oglesby, Bennet, Settle, *Semi Automated Mathematics.* J.ACM 16,1, (1969).

KK71 Kowalski, R., Kuehner, D., Linear Resolution with Selection Function. Artificial Intelligence 2 (1971) pp. 227-260.

KL86 Kifer, M., Lozinskii, E.L., Can we implement Logic as a Database System. Techn. Report 86/16 SUNY, Stony Brook, (1986).

Ko75 Kowalski, R., *A Proof Procedure using Connection Graphs,* JACM , Vol .22 No.4 (1975), 424-436.

Lo78 Loveland, D., *Automated Theorem Proving.* North Holland (1978).

Ni80 Nillsson N.J., *Principles of Artificial Intelligence.* tioga publishing company, Palo Alto, California (1980).

Ro65 Robinson, J.A., *A Machine Oriented Logic Based on the Resolution Principle,* J.ACM, Vol. 12, No. 1 (1965), 23-41.

St85 Stickle, M.E., *Automated Deduction by Theory Resolution.,* JAR, Vol.1, No.4 (1985), 333-356.

St86 Stickle, M.E., *A PROLOG technology theorem prover: implementation by an extended PROLOG compiler.* Proc. of 8th Int. Conf. on Automated Deduction, Oxford (1986) pp. 573-587.

TS86 Tamaki H., Sato T., *OLD resolution with tabulation.* Proc. of 3rd Int. Conf. on Logic Programming, London (1986) pp. 84-98.

Vi 87 Vielle, L., *Recursive Query Processing: The Power of Logic.* ECRC Munich, Technical Report TR-KB-17 (1987).

Wo84 Wos, L., Overbeek, R., Lusk, E., Boye, J., *Automated Reasoning - Introduction and Applications.* Prentice-Hall, Englewood Cliffs, NJ (1984).