Laurent VIEILLE


Communicated by G. Ausiello
Received October 1987
Revised October 1988

Abstract. The purpose of this paper is to show that a computational model developed in the framework of resolution provides a very adequate tool to study and develop query answering procedures for deductive databases, as well as for logic programs. As a result, we introduce an effective query answering procedure for deductive databases. To achieve our goal, we first develop techniques (applicable to general logic programs) for the construction of abstract search spaces associated with a query, and we discuss their properties. We then show how these techniques can be practically applied to the problem of answering recursive queries in a deductive database (consisting of function-free clauses). This approach has given rise to a new general-purpose procedure, termed QoSaQ, which improves on earlier general-purpose methods, in particular by its ability of incorporating a so-called global optimization technique. We show that the framework provided by QoSaQ is powerful enough to account for the best-known recursive query evaluation methods. We also produce an upper-bound to the number of tuples manipulated by QoSaQ, which improves on known upper-bounds.

Contents

1. Introduction ................................................................. 2
   1.1. Further relevant work .................................................. 6
2. SLD resolution ............................................................. 7
   2.1. Introduction: analogy with graph-searching ...................... 7
   2.2. Notions of first-order language and of resolution ............. 8
   2.3. Goals, resolution and SLD trees .................................... 9
   2.4. The descendence relationship and local selection functions ... 11
   2.5. Side-way information passing and focus on relevant data .... 13
3. SLD-AL resolution ....................................................... 14
   3.1. Introduction ........................................................... 14
   3.2. SLD-AL trees ......................................................... 15
   3.3. Search strategy and completeness ................................. 16
   3.4. Variations on the admissibility test ............................... 17
   3.5. Further examples .................................................... 17
   3.6. Subgoal generalization and SLD-ALG resolution ............... 19
4. Redundancy elimination on SLD and SLD-AL trees ................. 20
   4.1. Presentation ......................................................... 20
   4.2. (Db-)subsumption ................................................... 23
   4.3. Optimized SLD trees ............................................... 24
   4.4. Local and global optimization of SLD-AL resolution .......... 24
The question of efficiently answering a query over a logic database (also called logic program) has attracted much attention over the past few years, both in the logic programming field [36] and in the deductive database field [17]. A logic database essentially consists of a set of definite clauses (also called rules) and of a set of facts. A (classic) example is the ancestor example: the two rules stated below define the ancestor relationship, given a parenthood relationship parent between individuals:

\[
\text{anc}(X, Y) :- \text{parent}(X, Y).
\]

\[
\text{anc}(X, Y) :- \text{parent}(X, Z), \text{anc}(Z, Y).
\]

Various issues of query processing have to be considered: completeness and termination of the answering process, its (logical) efficiency (in particular, its ability to focus on the data relevant to the query, and its ability to avoid redundant computation), its worst-case complexity, and finally, engineering issues (these issues may differ between a programming and a database environment). These issues have attracted a considerable amount of work using, in particular in the database community, a number of different formalisms. The need for a unifying computational model has often been expressed.

The purpose of this paper is to propose such a unifying computational model, based on the resolution principle [47, 10], and making use of particular techniques that we develop here. This computational model gives new insights into known evaluation methods, yields extensions to several of them [61] and suggests further
developments. In our mind, resolution provides a basis as adequate for query answering procedures in (deductive) databases, as for the Prolog programming language [36].

To achieve our goal, we first consider formal aspects of query answering and we study (abstract) techniques that can be used to construct various types of search spaces to be associated with a query. Search spaces are abstract notions which help discussing general key properties such as completeness, termination, logical efficiency. Then, we consider more practical aspects of query answering and we study the search techniques that can be used to represent and to manipulate the nodes of these search spaces. These search techniques tend to solve the engineering issues that are specific to a given type of system. Hence, they may differ between, say, a database system and Prolog. In the database case, a query answering procedure can be seen as “constructing” a particular type of search space, while using particular search techniques to represent and manipulate the nodes of this search space, in a way hopefully best adapted to its task. To summarize, we can write

\[ \text{Query Answering Procedure} = \text{Search Space} + \text{Search Techniques}. \]

**SLD-AL resolution**

We obtain our search spaces by enriching SLD resolution [3, 36] through two types of techniques, applicable in the case of general logic databases (i.e. countable sets of clauses with or without function symbols). The rationale behind our use of SLD resolution lies in its properties: its top-down features, together with the freedom it leaves for the selection of the next subgoal to answer, provides a nice environment to develop procedures able to focus on relevant data. We now outline the two types of techniques with which we enrich SLD resolution.

The AL-technique (Section 3), standing for Admissibility test and Lemma resolution, amounts to checking whether the current subgoal is new (admissibility test), and, in the case it is not, to re-using the answers obtained for its previous occurrence (lemma resolution). This technique permits each subgoal to be answered only once, while guaranteeing answer completeness. As a first property, the resulting SLD-AL trees are finite in a number of cases where SLD trees would be infinite, in particular for finite and function-free databases. A second gain is on complexity: for instance, SLD-AL resolution can be used to describe the computation of \( \text{Fibonacci}(n) \) in time and space linear in \( n \).

Our second technique (Section 4) aims at pruning redundant parts of SLD and SLD-AL trees, i.e. subtrees which contain answers that are, in any case, contained by another part of the tree. This redundancy elimination is based on a so-called \( db\)-subsumption test between nodes, which improves on the classical subsumption test in that it preserves answer-completeness of the search space, instead of simply refutation-completeness. We distinguish between local and global redundancy elimination. Essentially, when performing local optimization, we will prune a subtree only if we can guarantee the completeness of the answer to any local subgoal on the tree. On the other hand, when performing global optimization, we will guarantee the
completeness of the answer to the top query, without necessarily requiring this completeness for any local subquery.

**Deductive databases**

As for the *search techniques* side of our initial distinction, we are especially interested here (Section 5) in deductive database systems. A *deductive database* is essentially a finite database whose clauses are often supposed to be function-free, but which may contain an enormous amount of facts. The (practical) problem of query answering in deductive databases has recently attracted much attention [1, 4, 5, 7, 8, 11, 15, 19, 20, 25, 29, 38, 40, 48, 50, 51, 53, 55, 56]. It has also been the initial motivation for the present work [58, 59, 60, 61], which must be seen as a development of the Query/SubQuery approach that we first proposed in [58] and that was first implemented in the DedGin system [59].

In the database context, engineering issues are made hard by the storage of data in secondary memory, and by the expensive page-based access to it. The implementation of relational systems has shown that it was advantageous, from the overall performance point of view, to generate operations over (sets of) tuples. We therefore develop a representation of SLD/SLD-AL nodes by means of tuples and the associated manipulation operators, derived from the basic operations of SLD-AL resolution. This leads to a general purpose procedure, termed QoSaQ (QoSaQ = Q.S.Q + g!ObAl optimization), for (recursive) query processing. A detailed presentation of QoSaQ can be found in [61].

The unique features of QoSaQ are obtained thanks to its roots in SLD-AL resolution: QoSaQ provides, as a toolkit, a small set of basic operations, derived from those of SLD-AL resolution. This toolkit turns out to be a powerful framework for the description of numerous methods proposed in the literature. As an example, general methods, such as QSQR [58], Magic Set [5, 8], Alexandre [48], which fulfill the main requirements (termination, completeness, set-oriented processing, focus on relevant data), can be seen as implementations of SLD-AL resolution with local optimization. Also, specialized methods, like Counting [5] and the Henschen/Naqvi algorithm [19, 20], which do not apply in all cases but have been shown [4] to be potentially more efficient on those cases where they apply, can be seen as implementations of SLD resolution with global optimization, thus explaining both their non-termination and their (potentially) increased performance. To summarize, QoSaQ provides a unique framework where a few basic techniques (and operations) can be combined in various ways, hopefully to yield better performance.

We provide (Section 6) a (worst-case) upper-bound to the number of tuples manipulated by QoSaQ, which improves on known upper-bounds [9, 24, 57]. First, this upper-bound is always less than or equal to the ones previously known. Second, it is expressed in terms of the size $p$ of the relevant data, whereas previous upper-bounds were expressed in the size $n$ of the whole database. In most current database cases, $p$ is going to be much smaller than $n$; hence, in these cases, the upper-bound for QoSaQ is going to be much smaller than these earlier upper-bounds. This
validates, from the complexity point of view, the interest of a procedure which focuses on relevant data.

On the power of logic

The use of resolution as a basic mechanism for query evaluation in database systems is not customary, and is often questioned. The main criticism made to this approach is its "tuple-at-a-time" nature, which may seem, a priori, contradictory to the "set-oriented" processing favored in relational systems. Indeed, the inference mechanism is, by nature, described on a one-operation-at-a-time basis: (one) resolution against (one) fact or against (one) clause. A first answer to this point consists in noting that a so-called set-oriented operator is, after all, nothing but a set of elementary operations on tuples. A set of "possible inferences" from a "set of facts" can, as well, be collected in a "set-oriented" operator.

We would like, however, to go even further and to claim that the practical interest of our approach lies in its one-inference-at-a-time basis, as opposed to having a set-theoretic basis.

First, this tuple-based computational model permits a fine analysis of the duplicate elimination issue. As a first example, QoSaQ provides an exact specification of when duplicate intermediate tuples must be eliminated to guarantee termination and of when this expensive operation can be avoided. As a second example, it is the distinction between local and global duplicate elimination that permits such a precise description of various methods for recursive query processing. Finally, let us note that this duplicate elimination issue is far from settled in current relational systems.

Second, this computational model provides, in our mind, an adequate framework for studying existentially quantified queries, ground queries, and, in general, yes/no queries. To answer these queries, it is sufficient for the evaluation procedure to find one way to answer it. For instance, to answer "\( \exists X p(X) \)", it suffices to find one value for \( X \); there is no need to enumerate all of them. This issue finds a natural translation in the computational model advocated here ("find one success node in the search space"), whereas it may be more involved in a set-theoretic computational model [13].

Finally, let us note some further developments of this work. An implementation of QoSaQ is being done by Lefebvre in the DedGin* project at ECRC [35]. The completeness of the extension of SLD-AL resolution and of QoSaQ to the case of stratified databases (where rules can contain negative literals) has recently been proved by Kemp and Topor [26] (see also [52]). Hulin [23] has investigated different solutions to implement global optimization as in QoSaQ.

1.1. Example. To complete this introduction, we give an example which is easy to understand and complex enough to illustrate the need for the query answering techniques we develop here. Let parent, hu (human-being) and married be base relations. The same-generation relation can be defined as: any human-being \( X \) is of the same generation as himself; \( X \) and \( Y \) are of the same generation if their respective
parents $U$ and $V$ are either of the same-generation, or are respectively of the same generation as $Z$ and $W$, and $Z$ and $W$ are married. This yields the following set of rules:

$$sg(X, X):-hu(X).$$

$$sg(X, Y):-\text{parent}(X, U), sg(U, V), \text{parent}(Y, V).$$

$$sg(X, Y):-\text{parent}(X, U), sg(U, Z), \text{married}(Z, W), sg(W, V), \text{parent}(Y, V).$$

1.1. Further relevant work

The originality of our work may not reside in one or several of the single ideas it contains, but more in their formal treatment and in their integration to provide a unifying framework for recursive query handling.

We list first the work relevant or even close to SLD-AL resolution. Re-using answers for new occurrences of the same subgoal is a technique already mentioned by Kowalski [31] and known in theorem proving [43]. SLD-AL resolution is also related to the extension of Earley's parsing algorithm into so-called Earley's deduction [16, 15, 44]. Lang [33, 34] has recently investigated the relationship between advanced parsing techniques and the work presented here. There is a strong similarity between his notion of items and the tuples manipulated by QoSaQ. Tamaki and Sato introduced OLDT resolution [54], similar to non-optimized SLD-AL resolution. However, OLDT does not allow logical optimization as rules are required to be pre-ordered. Tcpor and Kemp [26] have proved the completeness of the extension of SLD-AL resolution to the case where stratified databases are allowed; the semantics they give to stratified databases is the standard model [45]. Seki and Itoh have obtained a similar result [52], but working in the OLDT formalism.

Apparently, the first authors who introduced a notion similar to \textit{db-subsumption} on SLD trees were Minker and Nicolas [40], but they did not investigate the same properties as we do. Tableau containment [2] is a subcase of db-subsumption, as it applies only in the flat case. This notion is also known within the Prolog community.

Work in \textit{deductive databases} also includes Ullman's capture rules formalism [55]; however, the termination/completeness issue is not fully treated there. Van Gelder proposed a message passing framework [56]. The representation used for SLD-AL nodes as tuples (Section 5) is similar to the notions used in the Alexander method [48, 27, 28] and in the Supplementary Magic Sets method [8, 51]. \textit{Global optimization} is partially present in the counting method [4] and \textit{relation waking} in the Magic Counting method [51]. Taking as \textit{cost metrics} the number of intermediate tuples manipulated (Section 5), is already (informally) advocated in [4, 8]. It should also be seen as a \textit{global} cost metrics, as opposed to the \textit{relative} one used in [6].
2. SLD resolution

2.1. Introduction: analogy with graph-searching

The use of resolution for (recursive) query answering appears very natural as soon as one considers the analogy between graph searching and the basic operations of resolution.

Under certain conditions, conjunctions of literals and, more generally, (recursive) rules can be seen as specifying classes of paths in a data graph associated with the database. For instance, if the database contains (binary) tuples for a parent relation, then the body of the rule defining the grand-parent relationship (gp) specifies paths of length 2 in the graph associated with parent:

\[ \text{gp}(X, Y):= \text{parent}(X, Z), \text{parent}(Z, Y). \]

Similarly, the definition of the ancestor relation (anc) given in the introduction specifies the class of paths of arbitrary length in the parent graph.

2.1. Example. The definition of same-generation given in Example 1.1 specifies a rather complex class of paths. For simplicity, we will consider a simplified (no third rule) and modified (the two occurrences of parent are replaced by p and p') version of this example. The graph associated with the data (relations p and p') is given in Fig. 1.

\[ \text{sg}(X, X):= \text{hu}(X), \]
\[ \text{sg}(X, Y):= \text{p}(X, U), \text{sg}(U, V), \text{p'}(Y, V). \]
\[ \text{p}(a, b) \quad \text{p'}(e, d) \quad \text{hu}(d) \]
\[ \text{p}(a, c) \quad \text{p'}(f, e) \]
\[ \text{p}(b, d) \]
\[ \text{p}(c, d). \]

A path between two individuals \(\text{ind}_1\) and \(\text{ind}_2\) is of type sg if and only if there exist an arbitrary \(n\), an individual \(\text{ind}_3\) (labeled by \text{hu}) and two paths consisting of \(n\) consecutive edges labeled respectively by \(p\) and \(p'\) and leading respectively from \(\text{ind}_1\) to \(\text{ind}_3\) and from \(\text{ind}_3\) to \(\text{ind}_2\). In our example, we have several such paths: two

![Fig. 1. Data graph for Example 2.1.](Image)
of them lead from $a$ to $f$, with $n = 2$ and $d$ as an intermediate vertex; the first one goes from $a$ to $d$ over $b$, and the second one over $c$:

$$a \rightarrow_p b \rightarrow_p d \rightarrow_p e \rightarrow_p f,$$

and

$$a \rightarrow_p e \rightarrow_p d \rightarrow_p e \rightarrow_p f.$$

The resolution principle can be seen as a calculus describing the construction of paths obeying the conditions expressed by the rules. Given a goal $G$ (a conjunction of literals), the resolution of a literal $SG^1$ selected out of $G$ can have two effects from the graph-searching point of view. When $SG$ is resolved against a fact, one follows the corresponding edge in the graph. When $SG$ is resolved against a rule, one applies a definition and replaces $SG$, in $G$, by the body of the rule.

Consider the goal $sg(a, Y)$, asking for the individuals $Y$ that can be reached from $a$ by a path of type $sg$. Applying the second definition (resolving $sg(a, Y)$ with the second rule) one obtains the new goal: $G_1: p(a, U), sg(U, V), p'(Y, V)$. This goal indicates that, in order to answer our top goal, one should search a $p$-edge from $a$ to an individual $U$, find a path of type $sg$ from $U$ to another individual $V$, and search (backwards) a $p'$-edge from $V$ to $Y$. This leaves a total freedom in the order of searching these paths. One can decide to first search a $p$-edge leaving $a$, e.g. $a \rightarrow_p b$. Doing so, the goal reduces to the new goal: $G_2: sg(b, V), p'(Y, V)$. One can also decide to follow first a $p'$-edge between any individuals $Y$ and $V$, e.g. $e \rightarrow_p f$. In this case, $G_1$ is reduced to $G_3: p(a, U). sg(U, e)$.

This freedom corresponds to the notion of selection function in SLD resolution.

2.2. Example. The graph traversal analogy does not capture the full power of rules and resolution: rules can contain terms; predicates can be of arity greater than 2, as in the present example. The following rules state that a group where all elements are their own inverses, is an abelian group. A comparison with simple graph traversal is no longer possible. The top goal is $p(b, a, c)$ (does $b * a = c$ hold?).

$$p(e, X, X), \quad \text{since } e \text{ is the neutral element.}$$

$$p(X, e, X),$$

$$p(X, X, e),$$

$$p(a, b, c). \quad \text{"a * b = c" holds}$$

Associativity laws:

$$p(A, BC, ABC) := p(A, B, AB), p(B, C, BC), p(AB, C, ABC),$$


2.2. Notions of first-order language and of resolution

We assume some familiarity with predicate calculus [39, 10], and with the resolution principle [47, 10]. The following definitions are all standard, except maybe the instance/variant definition where we assume that the expressions being compared do not share any variable and are of same length.

Standing for subgoal; there is no relationship with our same-generation $sg$ example.
The language consists of a countable set of predicate, function, constant, and variable symbols. We borrow much from the syntax of Prolog: constants, predicate and function (resp. variables) symbols are strings starting with lower (resp. upper) case letters; logical connectives are: "¬" (not); "·" (and); "→" (logical implication: →). A term is defined recursively as a constant, a variable or as \( f(t_1, \ldots, t_n) \) where \( f \) is an \( n \)-ary function symbol and \( t_i \) is a term. If \( \text{pred} \) is an \( n \)-ary predicate symbol and the \( t_i \)'s are terms, then \( \text{pred}(t_1, \ldots, t_n) \) (resp. \( \neg \text{pred}(t_1, \ldots, t_n) \)) is a positive (resp. negative) literal. An expression is a literal, or a conjunction, or a disjunction of literals. In \( p(t_1, \ldots, t_n) \), the term-depth of \( t_i \) is 1. If \( f(t_1, \ldots, t_n) \) appears in a literal \( \text{lit} \), then the term-depth of \( t_i \) in \( \text{lit} \) is 1 plus the term-depth of \( f(t_1, \ldots, t_n) \) in \( \text{lit} \).

A clause \( C \) is a disjunction of literals. A definite clause is a clause with one positive literal and \( q \) (\( q \geq 0 \)) negative literals. We write it à la Prolog: \( \text{lit}_0; \neg \text{lit}_1, \ldots, \neg \text{lit}_q \). \( \text{lit}_0 \) is the head and \( (\text{lit}_1, \ldots, \text{lit}_q) \) is the body of the clause. A fact is a definite clause without body (\( q = 0 \)). We do not make a difference between clause or rule.

We refer to [10, 36] for the definition of substitutions, the composition of substitutions, the unification of expressions, the mgu (most general unifier) of two expressions and for unification algorithms. We use Greek letters to denote substitutions: \( \theta, \gamma, \sigma \). We compose substitutions from left to right: \( \sigma \theta \) means that \( \sigma \) is applied first, then \( \theta \).

Let \( \text{Exp} \) and \( \text{Exp}' \) be two expressions which do not share any variables. If they do, then rename the variables of \( \text{Exp}' \). \( \text{Exp} \) is an instance of \( \text{Exp}' \) iff there is a substitution \( \theta \) such that \( \text{Exp}\theta = \text{Exp}' \) (up to a reordering of the literals in \( \text{Exp} \) and \( \text{Exp}' \)). Note that this requires that \( \text{Exp} \) and \( \text{Exp}' \) are of the same length. We also say that \( \text{Exp}' \) is more general than \( \text{Exp} \). \( \text{Exp} \) and \( \text{Exp}' \) are variant iff they are instance of each other.

A (logic) database \( \text{DB} \) is a countable set of definite clauses. A deductive database is a finite and function-free database [42, 17].

A predicate \( p \) directly depends on a predicate \( q \) if \( q \) appears in the body of a clause defining \( p \). Let the dependence relationship be the transitive (not reflexive) closure of the direct dependence relationship. \( p \) is called recursive if \( p \) depends on itself. \( p \) and \( q \) are mutually recursive if \( p \) depends on \( q \) and \( q \) depends on \( p \).

A fact \( F \) is derivable from a database \( \text{DB} \) iff there is an instance \( I \) of a clause in \( \text{DB} \) such that \( F \) is identical to the head of \( I \) and the body of \( I \) (if non-empty) is a conjunction derivable from \( \text{DB} \). A conjunction is derivable from \( \text{DB} \) iff each of the facts it contains derives from \( \text{DB} \). We do not discuss here the logical adequacy of this definition. See [36, 39].

2.3. Goals, resolution and SLD trees

A goal \( G \) is a conjunction of positive literals. Without loss of generality, we assume that a top goal \( G_0 \) contains a unique positive literal. A substitution \( \theta \) is a correct answer to a goal \( G \) iff \( G\theta \) is derivable from \( \text{DB} \). We say that \( \eta \) is an answer
to $G_0$ more general than $\theta$, if both $\eta$ and $\theta$ are answers to $G_0$ and if $G_0\eta$ is more general than $G_0\theta$.

Here, we are interested in providing procedures that return, to any goal $G_0$, the complete set of correct answers to $G_0$, from any database $DB$.

The basic mechanism we consider as the core of the answering process is that of resolution, which we now define. Let "$SG, Rest$" be a goal $G$, where $SG$ is a literal (SG stands for subgoal) and $Rest$ is a conjunction of literals. Let

$$Cl: \text{lit}_0:=-\text{lit}, \ldots, \text{lit}_q$$

be a clause sharing no variable with $G$.\footnote{If it does, we rename the variables of $Cl$.} Let $SG$ and $\text{lit}_0$ be unifiable with mgu $\sigma$.

**Definition.** The *resolution* of $(G, Cl)$ on the pair $(SG, \text{lit}_0)$ produces a goal $G'$$\vphantom{\sigma}$

$$G':(\text{lit}_1, \ldots, \text{lit}_q, Rest)\sigma.$$ 

Given a top goal $G_0$, the collection of resolutions that can be performed on $G_0$ and on its descendants (i.e. the goals obtained directly or indirectly by resolution from $G_0$), builds up a search space associated to $G_0$, called an SLD tree. SLD trees are defined up to a selection function, as follows.

**Definition.** A *selection function* is a function that, given a goal $G$, selects a literal, called the subgoal selected from $G$, on which resolution is to be performed.

As the order of literals in $G$ is not significant, we are entitled to write $G$ as "$SG, Rest$", as we did above, where $SG$ is the subgoal selected out of $G$.

**Definition.** An *SLD tree* is a tree whose nodes are labeled by goals and which is constructed as follows:

- The root is labeled by a top goal $G_0$.
- Let $N$ be a node, labeled by the goal $G$ (subgoal $SG$). Let $Cl$ be a clause whose head $lit_0$ is unifiable with $SG$. Then $N$ has a child $N'$, labeled by the goal $G'$ obtained by the resolution of $G$ and $Cl$ on $(SG, \text{lit}_0)$.

A *success node* is a node labeled by an empty goal.

With a node $N$, we associate a current substitution $\eta$. We also say that $N$ returns $\eta$.

**Definition.** With the root is associated the empty substitution. If $\eta$ is the current substitution at node $N$, if the child $N'$ of $N$ is obtained by a resolution of mgu $\sigma$, then the current substitution at $N'$ is $\eta\sigma$.\footnote{If it does, we rename the variables of $Cl$.}
The completeness of SLD resolution can be stated as follows (attributed to Hill [22] and Clark [12]; see [36]). We give another proof of Theorem 2.3 in Appendix A.3.

2.3. Theorem (Completeness of SLD resolution). For any SLD tree of root \( G_0 \) built on any selection function, for any correct answer \( \theta \) to \( G_0 \), there exists a success node returning an answer more general than \( \theta \).

2.4. The descendance relationship and local selection functions

Let \( N \) be labeled by a goal \( G: \text{"SG, Rest"} \). Let \( P \) be obtained by the resolution of \( N \) against the rule \( \text{"Head:-Body"} \). \( P \) is labeled by: \( \text{"(Body, Rest)\sigma"} \) (Body can be empty). One can trace the literals in \( G' \) back to the nodes at which they were first introduced. This is done as follows.

Definition. A literal \( \text{lit}_\sigma \) in \( P \) is said to be introduced at \( N \) if \( \text{lit} \) belongs to Body. If \( \text{lit} \) belongs to Rest, \( \text{lit}_\sigma \) is said to be introduced at the node \( Q \) (higher than \( N \) on the SLD tree) if the literal \( \text{lit} \) in \( G \) was introduced at \( Q \).

A literal \( \text{lit}_\sigma \) in \( P \) is said to be most recently introduced, if \( \text{lit}_\sigma \) was introduced at a node \( Q \) and if no other literal in \( P \) was introduced at a node strictly between \( Q \) and \( P \).

\( P \) (resp. its subgoal) is said to be a direct descendant of \( Q \) (resp. of its subgoal) if the subgoal selected at \( P \) was introduced at \( Q \). The descendance relationship is the transitive closure of the direct descendance relationship.

The descendance relationship does not coincide with the property of being lower on the SLD tree. On one hand, the descendance relationship captures a logical relationship between subgoals: \( SG' \) is a descendant of \( SG \) if and only if it was (directly or indirectly) introduced by \( SG \). On the other hand, SLD trees capture some procedural aspects: a resolution (e.g. of the first literal, \( p \), of a goal \( \text{"p, q"} \)) appears higher on the SLD tree than another one (e.g. of the second literal, \( q \), of \( \text{"p, q"} \)) whenever the former is performed before the latter (\( p \) and \( q \) do not need to be related by a descendance relationship). As an example, on Fig. 2b node 3 is not a descendant of node 2, as \( s(X, b_0) \) is neither directly nor indirectly introduced by \( t(Z, c_0) \). The corresponding AND/OR tree (Fig. 3) shows the descendance relationship more explicitly.

Definition. A local selection function is a selection function that always selects one of the most recently introduced literals.

In this paper, we will focus on trees built using so-called local selection functions. Local selection functions produce SLD trees with a simple structure, where each subgoal \( SG \) (selected out of a goal \( G \), at node \( N \)) is treated as if it were yet another top goal. If \( G \) is written as \( \text{"SG, Rest"} \), then all the descendants of \( G \) will have the form \( \text{"G', Rest\eta"} \), where \( G' \) contains only literals directly or indirectly introduced by \( SG \), and \( Rest \) remains unchanged, apart from the application of the successive substitutions (captured here in \( \eta \)). As a consequence, the node \( N \) is followed by a
Choosing the least instantiated literal \( \ldots \) \( \ldots \) most instantiated.

(a) Fig. 2. Illustration of the FRD strategy.

(b) Fig. 3. AND/OR tree corresponding to Fig. 2.

compact set of nodes, called a proof segment, where all the resolution steps are geared towards the proof of one answer to \( SG \).

**Definition.** \( N-P \) is called a proof segment for \( SG \) (for \( N \)), iff \( P \) is lower than \( N \) on the tree and \( P \) is the first node on this branch that does not contain any descendant of \( SG \). If \( \eta \) is the current substitution at \( P \), \( N-P \) is said to prove the lemma \( SG_\eta \). \( P \) is a proof node for \( SG_\eta \).

The restriction to local selection functions will turn out to be needed in general for a correct definition of SLD-AL resolution (although some weakenings are possible). Also, SL-resolution [32, 37] (from which SLD resolution was derived) was initially defined with local selection functions, whereas current definitions of SLD [6, 3] use unrestricted selection functions. Although standard Prolog systems use a local selection function, more advanced systems including co-routining facilities call for unrestricted selection functions [41, 21].
2.5. Side-way information passing and focus on relevant data

It is interesting to compare, from the query processing point of view, the formal tools provided by the resolution formalism with some other tools developed in the database literature.

The first such tool is unification and the (side-way) propagation of bindings it provides. When a subgoal is resolved against a clause (fact), unification provides bindings for the variables that are both in the clause and in the subgoal. As the latter variables may be shared by other literals in the goal, those bindings also apply to these other literals. Hence, unification during resolution provides a unique, transparent way to propagate bindings, in a side-way manner, from a subgoal being answered to the remaining literals in the goal. This side-way information passing notion has been considered as a major concept in the database literature [55, 56, 8, 48].

The second notion that plays a major role is that of selection function. The freedom to select any of the (most recently introduced) literals in the goal provides a unique opportunity to control the evaluation process. This control can be performed either by the user or by a query answering optimizer.

In a Prolog-like programming environment, the order of literals in the body of a clause is significant (it is part of the procedural semantics of the language). Knowing that the system will always select the first literal to come, the user can specify the order in which subgoals will be evaluated. This is a way to give to the user the full responsibility for termination and efficiency, for which no automatic solution is possible in the general case.

In deductive and relational database systems, choosing the/a good selection function will be the task of the query optimizer: two different selection functions can often lead to evaluation costs differing by an order of magnitude. For instance, this optimizer may apply a strategy focusing on relevant data (FRD) [38, 58], that corresponds to the heuristics "make selections first", popular in database systems [4]. Informally, this consists of taking advantage of the constants appearing in the query to consider only the facts relevant to the query. In terms of SLD resolution, this corresponds to a selection function that: tries to choose a literal amongst the most instantiated ones (whatever this precisely means).

2.4. Example. Consider the following database:

\[
\begin{align*}
r(X, Y): & \leftarrow s(X, Z), t(Z, Y) \\
s & \\
s(a_0, b_0) & \leftarrow t(b_0, c_0) \\
\vdots & \\
s(a_n, b_n) & \leftarrow t(b_n, c_n). \\
\end{align*}
\]
Figures 2(b) and (a), respectively, display trees obtained by using (resp. not using) an FRD strategy to answer $r(X, c_0)$ on this database. The respective size of the trees is a sufficient illustration of our point.

3. SLD-AL resolution

3.1. Introduction

The construction of SLD trees follows a blind strategy: select a subgoal and answer it by further resolution. This strategy is blind as the same subgoal, or similar subgoals, may appear at several places of the tree. Not recognizing this similarity leads to redundant work and, potentially, to infinite computations.

For instance, if a subgoal $sg(a, X)$ has already been considered, then there is no need to restart the same work when a new subgoal $sg(a, X_1)$ is encountered. Recognizing that the second subgoal is not new and reusing the results obtained for the first one, can even avoid entering an infinite loop.

There may be an infinite branch when the second subgoal $sg(a, X_1)$ is a descendant of the first subgoal $sg(a, X)$: in this case, one can repeat, starting from $sg(a, X_1)$, exactly the same sequence of resolutions as the one that led from $sg(a, X)$ to $sg(a, X_1)$. This leads to a new similar subgoal, say $sg(a, X_2)$. Now, this can be repeated infinitely many times.

SLD-AL resolution can be seen as eliminating this subquery redundancy and rests on two simple principles. The admissibility test prevents a subgoal $SG$ similar to a previous one $SG'$ from being indefinitely solved. In the finite and function-free case, the admissibility test cuts off any infinite branch. The resolution of non-admissible subgoals against previously derived lemmas permits the production of further answers, by answering the remaining literals in the goal: the non-admissible subgoal $SG$ is thus exclusively resolved against the answers/lemmas proved for $SG'$, called its producer. In the sequel, we will call this two-fold mechanism the AL-technique.

3.1. Example (An infinite SLD tree vs. a finite SLD-AL tree). We consider the transitive closure, $tc$, of the relation $p$.

$$p(a_1, a) \quad p(a_2, a_1)$$

(1) $tc(X, Y) : \neg p(X, Y)$,

(2) $tc(X, Y) : \neg tc(Z, Y), \, p(X, Z)$.

The SLD tree displayed in Fig. 4 for the goal $tc(Z_0, a)$ is infinite as the resolution of each $tc(Z_i, a)$ against clause (2) introduces a successor $tc(Z_{i+1}, a)$. Figure 5 displays an SLD-AL tree corresponding to Fig. 4. In our figures, we will always represent a lemma resolution by a double line, whereas resolution against clauses from the database are represented by a single line. The subgoal $tc(Z_1, a)$ is not
Recursive query processing

Fig. 4. An infinite SLD tree for Example 3.1.

Fig. 5. A finite SLD-AL tree for Example 3.1.

admissible as it is a variant of the root. Hence, it is resolved against a lemma \( tc(a_1, a) \), which was previously proved at node \( P_0 \). This permits a further search of the tree, eventually producing the answer \( tc(a_2, a) \).

3.2. SLD-AL trees

This section is devoted to a proper definition of the construction of SLD-AL trees. Let us first note that, when constructing an SLD-AL tree, it is not necessary to apply the AL-technique to any type of predicates. Rather, in general, our results
on SLD-AL resolution will hold as soon as this AL-technique is applied to recursive predicates. Its application to non-recursive predicates is left open to choice. As a consequence, we isolate a subset of the predicates of our language, called the r-predicates. We suppose that this subset is given and that it contains (except stated) at least all the recursive predicates (hence the wording r-predicate). An r-literal (r-lemma, r-subgoal) is a literal (lemma, subgoal) built on an r-predicate.

Together with the tree itself, the construction of an SLD-AL tree supposes an access to the collection \( S \) of r-subgoals that have been considered so far and to the collection \( L \) of r-lemmas that have been proved so far. Both are initialized to the empty set. A subgoal in \( S \) (and the node \( N \) at which it was selected) can become a producer for other subgoals (nodes). By default, it is not a producer. The root is always considered as a producer (for the user).

A node \( N \) on an SLD-AL tree is labeled by a goal \( G \) and its selected subgoal \( SG \). A node is either admissible or not admissible. If \( SG \) is not a r-subgoal, then \( N(SG) \) is always supposed to be admissible. An SLD-AL tree of root \( G_0 \) and database \( DB \) is constructed as follows, up to a local selection function.

**Definition.** (Initial step). The root is labeled by \( G_0 \), which is admissible and a producer.

(Current step). Choose a node \( N(G, SG) \) and a resolution against \( G \) that has not been performed yet. If \( SG \) is not admissible, this resolution must be a resolution against a lemma. Otherwise, it is against a clause (or a fact) of the database.

1. If the resolution succeeds, add a child \( P \) to \( N \), labeled by the goal \( G' \) (subgoal \( SG' \)) resulting from the resolution.
2. Apply this step if \( P \) is a proof node for a r-lemma \( lem \). Add \( lem \) to \( L \), if and only if it is new, i.e. if it is not an instance of a lemma already in \( L \).
3. Apply this step if \( SG' \) is a r-subgoal. If \( SG' \) is an instance of a subgoal \( SG'' \) in \( S \), mark \( SG'' \) as non-admissible and \( SG'' \) as a producer (if it is not already). Otherwise, \( P \) is admissible and \( SG' \) is added to \( S \).

3.2. Proposition (Finiteness in the finite function-free case). If \( DB \) is a finite and function-free database, then any SLD-AL tree is finite.

Proposition 3.2 is proved in Appendix C. SLD-AL resolution was already presented in [60]. However, the definition of SLD-AL trees we give here is different from the one we gave there. The SLD-AL trees, as defined in [60], correspond to the straight SLD-AL trees that we define in Appendix D.

3.3. Search strategy and completeness

**Definition.** We call search strategy any function that is used to select the next resolution to try, at each step of the construction of an SLD-AL tree.
The notion of a search strategy is important in particular for completeness in the general case. In the general case, SLD-AL trees can be infinite. Hence, the search strategy may not construct the whole tree, but may get caught into an "infinite part" of the tree, without exploring the rest of it. A search strategy is fair if any potential resolution is selected after a finite number of steps. When the SLD-AL tree is finite (for instance if DB is finite and function-free), any search strategy is fair. Our definition of fairness is compatible with the definition given in [36] for SLD trees.

3.3. Theorem (Completeness of SLD-AL resolution). For any goal Go, for any (local) selection function and any fair search strategy, if θ is a correct answer to Go, there is a success node on the SLD-AL tree returning an answer more general than θ.

This completeness result is proved in Appendix A.

3.4. Variations on the admissibility test

Several alternative definitions of the admissibility test are possible, all of them leading essentially to the same completeness properties. We could for instance have defined SG' to be non-admissible if there was an ancestor SG" of SG' such that SG' was an instance of SG". Or we could have replaced the instance notion by the variant one (we do so in Section 5). Accordingly, this leads to four possible alternative definitions. Our completeness (Theorem 3.3) and finiteness (Proposition 3.2) results hold for any of them. However, the complexity results of Section 6 hold only when the admissibility test compares a subgoal to any previous subgoal on the tree (not only its ancestors), as defined in Section 3.2.

3.5. Further examples

Besides the example given in the introduction to this section, we give a few examples showing the behavior of SLD-AL resolution in various contexts.

3.4. Example (The Fibonacci function). Let the Fibonacci function be defined as

\[ \text{fib}(1, 1) \]
\[ \text{fib}(2, 1) \]
\[ \text{fib}(X, Y) : \neg \text{plus}(X', 1, X), \text{fib}(X', Y'), \]
\[ \text{plus}(X'', 1, X'), \text{fib}(X'', Y''), \text{plus}(Y', Y'', Y). \]

When assuming the (countably many) facts of the plus relation to be provided on request by a procedure, one can build a SLD-AL tree of top goal \( \text{fib}(n, Y) \) and whose size is linear in \( n \). Hence, \( \text{fib}(n, Y) \) can be computed by searching this SLD-AL tree in both time and space linear in \( n \). This is done while using a selection function which selects the literals in their order of appearance in the above rules.
Other selection functions may lead to infinite trees (as, for instance, if a non-instantiated subgoal, say \( \text{fib}(X', Y') \), is selected).

3.5. Example (The restriction to local selection functions is necessary). This example shows that our restriction to local selection functions is necessary. In Fig. 6, at node \( N \), the choice of \( q(X) \) is not local. The resulting instantiation of \( X \) to \( a_2 \) prevents the production of the lemma \( tc(a_1, a) \). As a consequence, this tree derives neither \( r \)-lemmas nor answers to the top goal, whereas \( n_{-}tc(a_2, a) \) is an answer to the goal.

\[
\begin{align*}
p(a_1, a) & \quad q(a_2) \\
p(a_2, a_1) & \\
n_{-}tc(X, Y) & : - tc(X, Y), q(X) \\
tc(X, Y) & : - p(X, Y) \\
tc(X, Y) & : - p(X, Z), tc(Z, Y).
\end{align*}
\]

3.6. Example (Infinite SLD-AL trees). In the general case, an SLD-AL tree can be infinite in two different ways. First, the length of a branch may become infinite, i.e. there is an infinite sequence of nodes (an infinite sequence of resolutions) such that each node is a child of the previous one. These infinite branches are similar to infinite SLD branches. Second, the width of a branch may also become infinite. If the database is infinite, this happens when a subgoal can be successfully resolved against infinitely many lemmas. If the database is finite, this may happen when a non-admissible subgoal can be resolved against infinitely many lemmas. In both cases, the node has infinitely many (direct) children. Such infinitely wide branches may occur on an SLD tree only when the database is infinite. Finally, note that these two kinds of infiniteness may coexist.

Although the second kind of infiniteness requires that there are infinitely many facts derivable from \( DB \), the first kind may appear even when there are only finitely many facts derivable from \( DB \) (even none as in Fig. 7(A)).

![Fig. 6. Incompleteness of SLD-AL for non-local selection functions.](image)
An SLD-AL tree of infinite length (Fig. 7(A)) is obtained with the goal \( p(a) \), on the database:

\[
p(X) :- p(f(X)).
\]

An SLD-AL tree of infinite width is obtained (Fig. 7(B)) with the goal \( q(Y) \) on the database:

\[
q(a) \quad q(f(X)) :- q(X).
\]

3.6. Subgoal generalization and SLD-ALG resolution

A shortcoming of SLD-AL trees is that they may become infinite even when there are finitely many facts derivable from \( DR \). The reason is that subgoals may become of increasing complexity while remaining admissible, as displayed in Fig. 7(A). In this section, we outline SLD-ALG resolution (see also [60]), obtained from SLD-AL resolution by the introduction of the subgoal generalization technique [54, 29].

The main idea is to replace a subgoal \( SG \) by a generalized subgoal \( SG' \), as soon as the complexity of \( SG \) goes beyond a given threshold. This threshold is expressed as a bound to the term-depth (Section 2.2) that a (sub)term is allowed to reach in \( SG \). The generalization \( SG' \) of \( SG \) is then obtained by replacing all (sub)terms in \( SG \) whose term-depth is greater than this threshold, by a new variable. For instance, if this threshold is 2, then the subgoal \( \text{pred}(f(X, g(a))) \) is generalized into \( \text{pred}(f(X, Z)) \).

SLD-ALG resolution can then be defined as SLD-AL resolution, except that a subgoal \( SG \) is replaced by its generalization \( SG' \) whenever the complexity of \( SG' \) goes beyond this threshold. Also, SLD-ALG resolution must check whether an answer \( \theta \) to \( SG' \) is an answer to \( SG \); this is done by unifying \( SG' \theta \) and \( SG \).

SLD-ALG resolution incorporates the following trade-off: on one hand, generalizing subgoals leads to a finite number of different subgoals and the admissibility test
cuts off any infinitely long branch. On the other hand, answers to the generalized subgoal $SG'$ are not necessarily answers to the initial one. This is a potential loss of efficiency, as irrelevant answers/lemmas will be produced.

SLD-ALG trees enjoy nice properties: they are complete search spaces (under fair search strategies) and are finite as soon as the Herbrand model of $DB$ is finite.

Formally, SLD-ALG resolution slightly generalizes OLDT resolution [54], as any local selection function is now allowed (OLDT is restricted to ordered clauses).

4. Redundancy elimination on SLD and SLD-AL trees

4.1. Presentation

Again, an easy way to obtain an intuition about redundancy elimination is to use the analogy with graph-searching, as presented in Section 2.1. Along this analogy, we identify the query answering process with the construction of those paths in the graph that meet the requirements expressed by the rules. Now, when constructing these various paths, it will happen that the same vertex, i.e. the same individual of the database, is encountered several times. Natural questions arise: are these various events redundant? Under what conditions can we stop constructing a path, because one of its vertices has already been considered?

In the resolution formalism, a goal specifies, at each stage of the computation, what kind of path remains to be constructed to answer the top query. The natural idea is thus to compare these goals and to test whether their path "specifications" are identical, variant, or if one is more general than the other.

As an example, consider the data and the rules of Example 4.1. A data graph and an SLD tree of root $q(a, Y)$ are given in Fig. 8. To answer this query, we need to return the terminal nodes of all paths of length 3, whose initial node is $a$. When constructing these paths, we meet the individual $d$ three times. The goals associated with the two first such events (nodes 1 and 2 on the SLD tree) are essentially identical: one of them is redundant and can be discarded; the goal associated with the third such event (node 3) is clearly different, as it requires two additional moves in the graph, instead of just one.

4.1. Example. Let $q(X, Y)$ be defined as

$$q(X, Y):=\neg p(X, Z), p(Z, U), p(U, Y)$$

together with the data:

$$p(a, b) \quad p(b, d) \quad p(d, e) \quad p(e, f)$$

$$p(a, c) \quad p(c, d)$$

$$p(a, d).$$

A first objective of this section is thus to provide the adequate tools to compare goals and to recognize when one of them is redundant. Formally, we will use a
subsumption-based technique [18, 37] to detect redundancy. However, this test is expensive to apply: from the practical point of view, we will only consider restricted versions of it (Section 5).

Another important aspect to take into account when detecting redundancy, is that redundancy may be relative to one given path construction subproblem, or, in other words, relative to one given ancestor in the SLD/SLD-AL tree. As an example, let us take the definition of sg and the database given in Example 2.1. Figure 9 displays both the data graph and an SLD tree of root sg(a, Y). As already discussed in Section 2.1, one can construct two different paths to answer the goal sg(a, Y) on this data. These two paths are, respectively,

\[ a \rightarrow_p b \rightarrow_p d \rightarrow_p e \rightarrow_p f, \text{ and } a \rightarrow_p c \rightarrow_p d \rightarrow_p e \rightarrow_p f. \]

The answering process essentially follows these paths: after having followed the edges \( a \rightarrow b \) (resp. \( a \rightarrow c \)), we obtain the goals "sg(b, V), p'(Y, V)" (resp. "sg(c, V), p'(Y, V)"), which specify what remains to do (nodes 3 and 9 of the SLD tree). Following the edges \( b \rightarrow d \) and \( c \rightarrow d \), one arrives twice at \( d \), with two occurrences of the goal "sg(d, V), p'(V, V), p'(Y, V)" (nodes 5 and 11). These two occurrences are clearly redundant relatively to the top goal: to go from \( a \) to \( f \), one needs to search the path \( d \rightarrow e \rightarrow f \) only once.
However, the two nodes 5 and 11 are not redundant relative to all their ancestors. Suppose for instance that we discard node 11. Then the subgoal $sg(c, V)$, selected at node 9, does not get its answer $V = e$: the path $c \rightarrow d \rightarrow e$ is not fully constructed.

More generally, we will say that node 11 is only \textit{globally redundant} (relatively to some of its ancestors), but not \textit{locally redundant}: it is not redundant relative to all its ancestors. As opposed, node 13 is locally redundant (if one decides to resolve 11 and to reach 13): it is redundant relative to its unique ancestor, namely the root.

Finally, let us take advantage of this example to contrast the AL-technique and redundancy elimination. Consider again the nodes 5 and 11 on the tree. Both have $'d, V_1')$ as selected subgoal. The AL-technique avoids answering this subgoal twice, and shares results between these two nodes. However, both nodes are further exploited; none of them is simply discarded. Hence the two branches of the tree are searched further. As opposed, redundancy elimination aims at pruning one whole branch.

\footnote{This can be more complex than in the present case!}
4.2. (Db-)Subsumption

We now define the db-subsumption test (database-subsumption) which will be used to detect the redundancy of goals. Let \( N_1 \) and \( N_2 \) be two nodes having a common ancestor \( N \), labeled by "SG, Rest". It follows that \( N_i (i = 1, 2) \) is labeled by a goal of the form "\( G_i, Rest\eta_i \)", where \( G_i \) is the set of literals descendant of \( SG \) at node \( N_i \), and \( \eta_i \) is the current substitution at node \( N_i \).

This db-subsumption test must not only guarantee that a success node can be reached from \( N_2 \) whenever one success node can be reached from \( N_1 \). It must only ensure that the bindings returned by these success nodes will give the same values to the output variables of \( G_1 \) and \( G_2 \), i.e. to those variables that carry values for \( SG \). This is the reason why the following test includes the comparison of \( SG\eta_1 \) and \( SG\eta_2 \). Example 4.3, given at the end of this section, provides more insight into output variables.

**Definition.** \( N_1 \) is db-subsumed by \( N_2 \) relatively to \( N \) if there is a substitution \( \sigma \) such that:

1. For any literal \( lit \) in \( G_2 \), \( lit\sigma \) is in \( G_1 \);
2. \( SG\eta_2\sigma = SG\eta_1 \).

\( N_1 \) is strongly db-subsumed by \( N_2 \) relatively to \( N \) if there is a substitution \( \sigma \) such that:

1. One can find a subset \( S = \{lit_i, i = 1, \ldots, n\} \) of \( G_1 \) and a one-to-one mapping between \( S \) and \( G_2 \) such that \( lit_i\sigma = lit_1 \);
2. \( SG\eta_2\sigma = SG\eta_1 \).

The node \( N_1 \) is said to be redundant relative to \( N \) if there is a node \( N_2 \), created before \( N_1 \), such that \( N_1 \) is strongly db-subsumed by \( N_2 \) relative to \( N \). \( N_1 \) is locally redundant, if it is redundant relative to all its ancestors.

Strong db-subsumption is strictly stronger than db-subsumption. For instance, if \( G_1 \) is "\( p(X, Y) \)" and \( G_2 \) is "\( p(X', Y'), p(Z', U') \)", then \( N_1 \) can not be strongly db-subsumed by \( N_2 \), whereas it may be subsumed (it remains to check the second part of the test definition). We need strong db-subsumption for completeness: see Appendices E and A. Our notion of strong subsumption is stronger than the \( \theta \)-subsumption of Loveland [37], as Loveland would only require that \( G_2 \) has fewer literals than \( G_1 \).

We note the following property, without proving it.

**4.2. Property.** If \( N_1 \) is redundant relative to one of its ancestors \( N \), then \( N_1 \) is redundant relative to all the ancestors of \( N \).

**4.3. Example.** Let us consider the following database:

1. \( p(X):-q(Y), r(X, Y) \)
2. \( p(X):-q(X), r(Y, X) \)

\( q(a) \quad r(b, a) \).
The goal $G_0 = \text{p}(X)$ admits two answers, namely $X = a$ and $X = b$. Resolving the goal $G_0$ against clauses (1) and (2) leads respectively to the goals “$q(Y_1), r(X_1, Y_1)$” (substitution $\{X/X_1, Y/Y_1\}$) and “$q(X_2), r(Y_2, X_2)$” (substitution $\{X/X_2, Y/Y_2\}$). These two goals are variant, hence subsume each other (by applying the substitution $\{X_1/Y_2, Y_1/X_2\}$ to the first goal, one obtains the second one). If we were interested only in the existence of one answer to $G_0$, we could discard one of them: whenever a success node is reached from one of them, another success node can be reached from the other one.

However, we are interested here in the completeness of the answer to $G_0$. From this point of view, these two goals are not equivalent: the first one returns $X = X_1 = b$; the second one returns $X = X_2 = a$. The reason why these two goals are not equivalent is clear: the substitution $\{X_1/Y_2, Y_1/X_2\}$ does not map the output variable ($X_1$) of the first goal onto the output variable ($X_2$) of the second one, but onto another variable ($Y_2$).

### 4.3. Optimized SLD trees

When building an SLD tree, completeness is usually sought only for the top goal. Hence, nodes that are found to be redundant relative to the root are simply not added to the tree. This yields the following definition.

**Definition (Optimized SLD trees).** Given a selection function and a (fair) search strategy, an optimized SLD tree of root $G_0$ is built as an SLD tree, except that a node which is redundant relatively to $G_0$, is not added to the tree.

**4.4. Theorem (Completeness of optimized SLD trees).** If $\theta$ is a correct answer to $G_0$, then, on any optimized SLD tree of root $G_0$, there is a success node returning an answer more general than $\theta$.

The completeness of SLD resolution (Theorem 2.3) is a corollary of Theorem 4.4. This result also holds for non-local selection functions, and is proved in Appendix A.3.

### 4.4. Local and global optimization of SLD-AL resolution

SLD-AL resolution, unlike SLD resolution, puts a certain emphasis on the answers to some local subgoals, the producers, as their answers can be shared between several subgoals. Further, in the general case, it is not possible to ensure the completeness of the answer to the top goal without ensuring the completeness of the answer to all producers: an example of such a case is given at the end of this section. In order to take this into account, our definition of optimized SLD-AL trees will guarantee that any producer gets a complete answer. This is a sufficient (but not necessary) condition to ensure completeness of the answer to the top goal.
We are now facing the following difficulty: one never knows, when creating a node $N$, whether or not it will become a producer later on. Hence, one may decide, at a given point of time, not to resolve a descendant $P$ of $N$ because $P$ is redundant relatively to the root (but not necessarily relatively to $N$). However, later on, $N$ may become a producer and require a complete answer. One needs, at that time, to further resolve $P$.

A first (simple) solution is to further resolve all the nodes that are not locally redundant. In this case, a complete answer is obtained for all local subgoals, in particular for all producers.

**Definition.** A *locally optimized* SLD-AL tree is an SLD-AL tree where a locally redundant goal is not added to the tree.

A second and more complex approach consists of discarding all locally redundant nodes and in resolving globally redundant ones only if they are needed for some producers.

**Definition.** A node $N_i$ is said to be *currently asleep* if it is redundant relatively to all its ancestors $N$ that are producers. $N_i$ is *currently active* if it is neither asleep nor locally redundant. A *globally optimized* SLD-AL tree is built as an SLD-AL tree, except that (1) a node $N$ can be chosen for further resolution if and only if it is active; (2) the step 3, admissibility test, is applied on $P$ if and only if $P$ is active.

**4.5. Theorem** (Completeness of optimized SLD-AL solution). For any goal $G_0$, for any (local) selection function and any fair search strategy, if $\theta$ is a correct answer to $G_0$, there is a success node on any globally optimized SLD-AL tree returning an answer more general than $\theta$.

This definition of globally optimized SLD-AL trees is abstract, as it does not specify any way to manage asleep/active nodes: how can we, at a given time, reconsider previously asleep nodes that have been *woken up* (made active), because one of their ancestors has become a producer? This practically complex problem has been addressed in the QoSaoQ framework [61], by Hulin [23] and in the Magic Counting method [51].

**4.6. Example.** Let us modify the data of Example 2.1 by adding a cycle in the relation $p$ (fact $p(d, c)$) and two new facts in the relation $p'(p'(g,f)$ and $p'(h, g))$. We obtain (see Fig. 10):

\[
\begin{align*}
p(a, b) &\quad p'(e, d) &\quad hu(d) \\
p(a, c) &\quad p'(f, e) \\
p(b, d) &\quad p'(g, f) \\
p(c, d) &\quad p'(h, g) \\
p(d, c). 
\end{align*}
\]
In the first case (Fig. 9), node 11 had been declared globally redundant, but not locally redundant. Now, because of the node 13 which is obtained from d (node 5) by following the edge back to c, the subgoal \(sg(c, V)\) (node 9) becomes a producer. If the lemma \(sg(c, e)\) is not produced at all, then one cannot perform the lemma resolution leading from node 13 to 16. This has the disastrous effect that the answer \(Y = h\) to the top query is not reached. The node 11 needs therefore to be woken up and further resolved, until the lemma \(sg(c, e)\) is produced. This allows the search...
of the branch producing \( Y = h \). Note that node 15 is locally redundant and can safely be discarded.

5. Application to deductive databases: the QoS\(\alpha\)Q procedure

5.1. Presentation

In this section, we apply SLD-AL resolution to the field of deductive databases [17] and we derive a general query evaluation procedure, called QoS\(\alpha\)Q, which is further described in a companion paper [61]. The essential factor to take into account when investigating query evaluation on (deductive) databases, is the potentially enormous number of facts which are stored on secondary storage. In comparison, we will consider that the number of rules remains smaller, and we make the following (classic) distinction.4

Hypothesis H1. We distinguish between base predicates (or base relations) which are exclusively defined by means of (potentially many) explicit facts (tuples), and virtual predicates, defined by means of rules.

QoS\(\alpha\)Q is obtained from SLD-AL resolution through (1) the representation of (sets of) goals by means of (sets of) tuples, and (2) the translation of the operations of SLD-AL resolution on goals onto operations on tuples.

The rationale behind this approach is the following. Clearly, we want to avoid an explicit manipulation of (sets of) goals, as goals may become very long strings. Further, the choice of a representation of (sets of) goals by means of (sets of) tuples is dictated by the lessons drawn from the implementation of relational database systems: it is now widely recognized that, in order for the query evaluator/optimizer to perform best, it is advantageous to decompose the operations to be performed onto operations on (sets of) tuples.

The two following hypotheses are both convenient and classic, although neither is, strictly speaking, necessary for the representation of goals by tuples.

Hypothesis H2. Any clause/fact in the database is range-restricted, i.e. any variable in its head also appears in its body.

The interest of H2 is that any fact in, or derivable from, a range-restricted database, is ground. As a consequence, all the intermediate tuples manipulated by QoS\(\alpha\)Q will be ground. Relaxing H2 requires making QoS\(\alpha\)Q able to handle non-ground tuples; this is a fairly easy task.

4 Although "hybrid" relations can be accommodated.
Hypothesis H3. The database is function-free, i.e. neither clauses nor facts contain any function symbols.

The interest of H3 is primarily to ensure termination of the evaluation process (see Proposition 3.2). Relaxing H3 is less easy than relaxing H2 (e.g. see [46]).

5.2. On the representation of goals

5.2.1. Stack-wise representation

We adopt a stack-wise approach to represent goals of an SLD/SLD-AL tree. Along this approach, the information contained in a goal G (node N) is split into several pieces, one piece for each ancestor of N.

Let \( (N_0, \ldots, N_n = N) \) be the sequence of ancestors of \( N \). The information to be attached to \( N_i \) is the information known at node \( N_i \) but not known at node \( N_{i-1} \). Let the goal \( G_{i-1} \) (node \( N_{i-1} \)) be of the form “\( SG_{i-1}, Rest_{i-1} \)”.

Then the goal \( G_i \) is of the form “\( SG_i, Set_i, Rest_{i-1} \sigma_i \)”, where “\( SG_i, Set_i \)” is the set of literals most recently introduced into \( G_i \) and \( \sigma_i \) the composition of the substitutions performed between \( N_{i-1} \) and \( N_i \). The information stored in \( Rest_{i-1} \) is already known at node \( N_{i-1} \). Hence, at node \( N_i \), we need to store only the information contained in \( (SG_i, Set_i, \sigma_i) \). This decomposition of \( N \) is summarized in Table 1.

| \( G_0 \) | \( SG_0 \) |
| \( N_1 \) | \( (SG_1, Set_1, \sigma_1) \) |
| \( \vdots \) | \( \vdots \) |
| \( N_i \) | \( (SG_i, Set_i, \sigma_i) \) |
| \( \vdots \) | \( \vdots \) |
| \( N = N_n \) | \( (SG_n, Set_n, \sigma_n) \) |

The goal \( G = G_n \) associated to \( N \) is:

\[
"SG_n, Set_n, \{Set_{n-1}[Set_{n-2}[\ldots Set_2[\ldots]\sigma_{n-2}]\sigma_{n-1}]\sigma_n"\]

5.2.2. Current tuple of bindings

Let \( (SG, Set, \sigma) \) be the information to be attached to a node \( N \). We want to represent this information while separating the values of the bindings, provided by \( \sigma \), from the syntactic information independent from these bindings. This syntactic

---

5 The distinction we will make between input and output arguments becomes more complex.
information can then be shared by (potentially many) goals of the same form. This is done as follows.

Let \( N \) be a direct descendant of \( N_1 \), labeled by the goal \( G_1: \text{"}SG_1, \text{ Rest}_1\text{"} \). Let \( SG_1 \) be resolved against the rule \( Cl: \text{"}\text{lit}_0:=-\text{lit}_1, \ldots, \text{lit}_q\text{"} \). Let \( SG \) be \( \text{lit}_1\sigma \). Let \( Set \) be \( \text{“(lit}_{k+1}, \ldots, \text{lit}_q\sigma)”}^6.

The syntactic information to be attached to \( N \) is \( \text{“lit}_k, \ldots, \text{lit}_q” \). The actual bindings provided by \( \sigma \) to variables of \( Cl \) are kept in a current tuple attached to \( N \), which is constructed as follows.

The binding of a variable \( X \) of \( Cl \) has to be kept in the current tuple, if this binding is already known and if it is still useful. The binding for \( X \) is already known either if it was provided as an input argument to the rule, i.e. if \( X \) appears in \( \text{lit}_0 \) and received this binding by the resolution with \( SG_1 \) (top-down propagation), or if this binding resulted as an answer to a subgoal corresponding to a previous literal of the rule, i.e. if \( X \) appears in a literal \( \text{lit}_j, j < k \) (side-way propagation). This binding for \( X \) is still useful either if it is needed as an output argument for the rule, i.e. if an occurrence of \( X \) in \( \text{lit}_0 \) corresponds to a variable in \( SG_1 \) (bottom-up propagation), or if it is needed for a subgoal yet to be resolved in the rule, i.e. if \( X \) appears in a literal \( \text{lit}_j, j \geq k \) (side-way propagation).

It is convenient to attach to \( N \) a frame \( \text{“(envir)_SG”} \), where \( \text{〈envir} \) collects the environment arguments of \( SG \). The binding of \( X \) is an environment argument of \( SG \) if and only if it is kept in the current tuple attached to \( N \) and it is not needed in \( SG \) (\( X \) does not appear in \( \text{lit}_k \)). Otherwise, i.e. if \( X \) appears in \( \text{lit}_k \), its binding is an input argument of \( SG \).

5.2.3. Bottom-up propagation of answer tuples: the lcid/lcont device

The stack-wise representation introduces an operation, otherwise unknown in SLD-AL resolution, which is called bottom-up propagation of answers. This operation is a side-effect of the stack-wise representation of goals. Let \( N_1 \) be the goal \( \text{“}SG_1, \text{ Rest}_1\text{"} \). Let \( (SG, Set, \sigma) \) be the information to be attached to a direct descendant \( N \) of \( N_1 \). The substitution \( \sigma \) contains bindings for variables of \( Rest_1 \), as the goal associated to \( N_1 \) is \( \text{“}SG, Set, Rest_1\sigma\text{”} \). Along our stack-wise representation, these bindings are a priori unknown to \( N_1 \). Eventually, these bindings have to be re-associated with \( N_1 \). This happens when all the literals remaining in \( \text{“}SG, Rest\text{”} \) have been successfully resolved, i.e. when the rules have been successfully executed.

If \( Cl(\text{“}\text{lit}_0:=-\text{lit}_1, \ldots, \text{lit}_q\text{”} \) was the rule resolved with \( SG_1 \), we will consider that the execution of the rule \( Cl \) provides an answer tuple for \( SG_1 \). This answer tuple is obtained by keeping the bindings for a variable \( X \) of \( Cl \) if and only if an occurrence of \( X \) in \( \text{lit}_0 \) corresponds to a variable in \( SG_1 \).

Finally, as the information attached to \( N_1 \) and to \( N \) is separated, we need a device to correctly return such an answer tuple to \( N_1 \). This is achieved through a so-called lcid/lcont device.

^6 Note that we arbitrarily numbered the literals in such a way that the \( (k - 1) \)st first ones have already been resolved.
Definition. To any node \( N \) whose subgoal \( SG \) is virtual, we give a local context identifier, \( \text{lcid} \). To any node \( N' \), we give a local context, \( \text{lcont} \), equal to the \( \text{lcid} \) of its direct ancestor.

Other solutions to re-associate answers with subgoals are discussed in [61].

5.2.4. Stack-wise representation of goals on an example

5.1. Example. Let us consider the following (recursive) rule (top goal \( r(a, Y) \)):

\[
r(X, Y) :- p(X, U, W), r(U, V), p'(V, W, Y)
\]

where \( p \) and \( p' \) are base relations. The resolution of \( r(a, Y) \) against this rule produces first the goal \( G \) "\( p(a, U, W), r(U, V), p'(V, W, Y) \)". In a database system, the set of \( p \)-facts matching \( p(a, U, W) \) can be large. Let \( \{ p(a, b_i, c_i) \mid i = 1, \ldots, n \} \) be this set. Resolving \( G \) against each of these facts, we obtain a set of goals:

\[
\text{"} r(b_i, V), p'(V, c_i, Y) \text{"}, \quad i = 1, \ldots, n.
\]

The syntactic information attached to these goals is identical for all of them, and is "\( r(U, V), p'(V, W, Y) \)". Their current tuples keep bindings for \( U \) and for \( W \) (the binding for \( X \) is not needed any more, and the other variables have not received bindings yet). We thus obtain a set of tuples, noted as: \( \langle * W, * U \rangle = \{ (c_i, b_i) \mid i = 1, \ldots, n \} \). The frames associated with these goals/tuples are \( \langle c_i \rangle \_r(b_i, V) \).

Now, suppose that the node \( N_i \) labeled by "\( r(b_1, V), p'(V, c_1, Y) \)" has a direct descendant \( N_i' \) of the form "\( r(b_1', V'), p'(V', c_1', V), p'(V, c_1, Y) \)"; then the information contained in this node can be decomposed, as in Table 1:

\[
N_i : \quad \langle c_i \rangle \_r(b_1, V);
N_i' : \quad \langle c_1 \rangle \_r(b_1', V').
\]

Further, the \( \text{lcont} \) of \( N_i' \) is equal to the \( \text{lcid} \) of \( N_i \).

Suppose now that further resolving the goal labeling \( N_i' \) returns the answer \( V = d_1 \) to the subgoal \( r(b_1, V) \) selected at \( N_i \). Suppose further that \( V = d_1 \) is not a correct answer to \( r(b_2, V) \). Then the answer tuple \( \langle d_1 \rangle \) must be associated with the frame \( \langle c_i \rangle \_r(b_1, V) \) to produce the subgoal \( p'(d_1, c_1, Y) \). As opposed, associating \( \langle d_1 \rangle \) with the frame \( \langle c_2 \rangle \_r(b_2, V) \), and producing the subgoal \( p'(d_1, c_2, Y) \) would be an incorrect step. By construction, \( \langle c_1 \rangle \_r(b_1, V) \) and \( \langle c_2 \rangle \_r(b_2, V) \) will be given two different \( \text{lcids} \), and the \( \text{lcont} \) of \( \langle d_1 \rangle \) will be equal to the \( \text{lcid} \) of \( \langle c_1 \rangle \_r(b_1, V) \). This will permit only the correct steps to be performed.

5.2.5. Comparison with (recursive) programming languages

There is an obvious analogy with the representation of the state of a program, and with the associated manipulation of frames, in the context of a recursive programming language. In particular, in Prolog implementations, SLD goals are also represented as a stack of frames. This analogy, however, falls short on two major aspects.
(1) We intend to be able to manipulate "sets of environments" and "sets of subgoals"; in a programming environment (e.g. Prolog) only one call to a procedure is issued at a time.

(2) The manipulations we perform on environments are not a mere combination of "push" and "pop" operations. This is due to the complex issue of returning lemmas to all the corresponding non-admissible nodes: a lemma to be resolved with a non-admissible goal $G$, may be proved only after $G$ was considered (in another branch). To solve this issue, we adopt here a so-called constructive search of the tree, where $G$ is not merely popped after it was considered, but is explicitly kept until all lemmas necessary for $G$ have been proved and used. See Appendix D for another solution.

5.3. Rule compilation

The purpose of a compilation process is to predetermine (1) the representation of the data to be manipulated during the evaluation, and (2) the operations to be performed on these data. In our case, this process predetermines the "shape" of the frames, current tuples and answer tuples, and the operations to be performed on them. This compilation has three main characteristics.

(1) For each rule defining a predicate $pred$, it may produce one compiled rule for each possible instantiation pattern for $pred$. Intuitively, two (sub)goals are said to have same instantiation patterns when they differ at most on the value of constants or on the names of their variables. For instance, $r(a, Y)$ and $r(b, Y)$ have the same instantiation patterns; as opposed, $r(X, c)$ has a different instantiation pattern.

Definition. The pattern $\omega$ of a literal $lit$ is a string of length $l$ (the arity of $lit$) over the set \{b, 1, ..., l\}. The $j$th symbol of $\omega$ is: (1) $b$ if the $j$th argument of $lit$ is a constant; (2) $i$ if the $j$th argument of $lit$ is its $i$th free variable, when free variables of $lit$ are considered in their order of appearance.

Examples. The pattern of $q(a, Y)$ is $b1$; the pattern of $t(a, Y, Y, Z)$ is $b12$.

(2) It pre-orders the literals in the body of the rule, by (pre)applying a so-called homogeneous selection function. A local selection function is homogeneous if it retains, as a unique criteria, the instantiation pattern of the subgoal against which the rule is resolved. Homogeneous selection functions do make possible a precompilation process, as it avoids making use of the actual values of bindings. We do not commit ourselves to any particular selection function. In examples, we use a "focus on relevant data" strategy (Section 2.5).

(3) It determines one so-called tuple-literal for each literal in the body of the rule, and a so-called target-list, to collect the answer tuples produced by the execution of the rule. A tuple-literal (t-literal for short) is an abstract structure that represents the "shape" of frames, and keeps the variables for which bindings should be kept. Intuitively, a t-literal can be derived from a frame $\langle envir\rangle_{SG}$ by replacing the actual binding of $X$ by a placeholder $*X$, called a *argument.
Example. \( ^*W \)_r\( ^*U, V \) is the t-literal associated with \( r(U, V) \), abstracted from, say, \( \langle c \rangle \)_r\( ^*b_1, V \) in Example 5.1.

5.3.1. One compilation process

Let \( R \) be a rule and \( \text{Head} \) be its head, built on a predicate \( \text{pred} \). Let \( \omega \) be a pattern for \( \text{pred} \). A possible compilation process of \( R \) into \( (R, \omega) \) consists of the following six stages.

1. "Apply" \( \omega \) to \( R \): if two occurrences of the same digit in \( \omega \) correspond to two different arguments of \( \text{Head} \), unify these arguments in \( R \). If this fails, there is no compiled rule \( (R, \omega) \). Otherwise, we note \( R\omega \) the resulting rule. For instance, if \( \omega = 11 \) and \( \text{Head} = \text{pred}(a, b) \), the application of \( \omega \) to \( R \) fails; if \( \omega = 11 \) and \( R: \text{pred}(X, Y):-q(X, Z), q(Z, Y) \), then \( R\omega: \text{pred}(X, X):-q(X, Z), q(Z, X) \).

2. Determine the input and output variables in \( R\omega \): a variable is an input (resp. output) variable if it appears as an argument of the head of \( R\omega \) and this argument corresponds to \( b \) (resp. to a digit) in \( \omega \). Note that a variable can be both an input and an output variable, as in: \( \omega = b1 \) and \( \text{Head} = p(X, X) \).

3. Generate the target list of \( (R, \omega) \): \( \text{Target} = \langle t_1, \ldots, t_d \rangle \), where \( d \) is the number of different digits in \( \omega \); \( t_i \) is the constant \( a \) (resp. \( *Y \)) if the \( i \)-th digit of \( \omega \) corresponds to \( a \) (resp. \( Y \)) in the head of \( R\omega \).

4. Choose an ordering for the literals in the body of \( R\omega \). Let \( \langle \text{lit}_0: \text{lit}_1, \ldots, \text{lit}_q \rangle \) be the resulting rule.

5. For each \( k = 1, \ldots, q \), construct the t-literal \( \text{tlit}_k \), of the form \( \langle \text{envir} \rangle \_* \text{lit}_k \), as follows:
   - \( * \text{lit}_k \) is obtained from \( \text{lit}_k \) by replacing a variable \( Y \) by \( *Y \) whenever \( Y \) is an input variable or appears in \( \text{lit}_l \) \((0 \leq l < k)\). The \( *Y \) are the input arguments of \( \text{tlit}_k \).
   - \( \langle \text{envir} \rangle \) is a tuple of distinct \( * \) arguments \( *X \), called the environment arguments of \( \text{tlit}_k \), where:
     1. \( X \) does not appear in \( * \text{lit}_k \)
     2. and \( X \) is an input variable or appears in \( \text{lit}_l \) \((0 < l < k)\)
     3. and \( X \) is an output variable or appears in \( \text{lit}_j \) \((j > k)\).

6. The compiled rule \( (R, \omega) \) is:

\[
(R, \omega): \text{tlit}_0 \rightarrow_t \text{tlit}_1 \rightarrow^h \ldots \rightarrow^h \text{tlit}_q \rightarrow^h \text{Target}.
\]

The ri-edge is a rule-introduction edge; h-edges are horizontal edges.\(^7\) The set \( \text{SR}(G_0) \) of compiled rules relevant to \( G_0 \), as defined as follows, is always finite.

**Definition.** A compiled rule \( (R, \omega) \) is relevant to \( G_0 \), if \( (\text{pred}, \omega) \) is relevant to \( G_0 \), i.e. if (a) \( G_0 \) is built on \( \text{pred} \) with instantiation pattern \( \omega \) or (b) there is a t-literal \( \text{tlit} \) in the body of a compiled rule relevant to \( G_0 \), such that \( \text{tlit} \) is built on \( \text{pred} \) and has \( \omega \) as instantiation pattern.

\(^7\) Horizontal edges were called sibling edges in [61].
5.1. Example (continued). The (recursive) rule of our running example:

\[ p(X, Y) :\neg p(X, U, W), r(U, V), p'(V, W, Y) \]

can be compiled onto (for the pattern b1):

\[ p(*X, Y) \rightarrow r p(*X, U, W) \rightarrow h (*W)_r r(*U, V) \rightarrow h p'(*V, *W, Y) \rightarrow h (*Y). \]

From the QoSaQ point of view, edges specify data manipulation operations. For instance, the horizontal edge

\[ (W)_r r(*U, V) \rightarrow h p'(*V, *W, Y) \]

requires, for each frame \( (C_i)_r r(b_i, V) \), the evaluation of the subgoal \( r(b_i, V) \) and, once this is done (say \( V = d_i \)), the production of a frame \( p'(d_i, c_i, Y) \) for the \( t \)-literal \( p'(*V, *W, Y) \).

The ri-edge is a triggering edge, triggering the execution of the rule for a subgoal on \( \text{pred} \), of pattern \( \omega \).

5.4. Data manipulation in QoSaQ

QoSaQ is a non-deterministic procedure, built on a few basic operations. These basic operations are obtained by translating the basic SLD-AL operations, defined on goals, onto operations on tuples. These operations manipulate different kinds of tuples, attached to a \( t \)-literal \( tlit \), that we now properly define.

**Definition.** A current tuple \( CT \) (for \( tlit \)) is a \( c \)-ary tuple, as soon as \( tlit \) contains \( c \) distinct *arguments. *arguments of \( tlit \) are considered in their order of appearance in \( tlit \).

The query tuple \( QT \) associated to the current tuple \( CT \) is the projection of \( CT \) onto the input arguments of \( tlit \).

An answer tuple \( AT \) for \( tlit \) is a \( d \)-ary tuple if \( tlit \) has \( d \) distinct free variables. Free variables are considered in their order of appearance in \( tlit \).

An intermediate tuple \( IT \) is either a current tuple or an answer tuple.

We recall that a current tuple \( CT \) for \( tlit \) represents a frame \( \langle envir \rangle_SG \), that can be obtained by replacing a *argument in \( tlit \) by its value in \( CT \). Keeping this in mind, the definitions and the basic operations of QoSaQ given in the rest of this section, should be clear.

5.4.1. Admissibility test and attribution of leids

The implementation of the AL-technique supposes that query tuples and answer tuples for \( \text{pred} \)-predicates are stored, apart from the answering process. This can be done by providing a query relation, storing query tuples, and an answer relation, storing answer tuples, for each pair \( (\text{pred}, \omega) \), where \( \omega \) is an instantiation pattern for a \( \text{pred} \)-predicate.
Definition. A query tuple QT (and the current tuple CT it derives from), attached to a t-literal tlit, is non-admissible if and only if tlit is built on a r-predicate and QT is not new when CT is generated. Otherwise, QT (and CT) are admissible.

Attribution of lcids: an admissible query tuple QT (and its current tuple CT) is given a new lcid. If QT is non-admissible, CT (QT) receives the lcid of the first occurrence of QT.\(^8\)

An intermediate tuple IT is a direct descendant of CT if and only if lcid(CT) = lcont(IT).

5.4.2. The three basic operations

Let CT be a current tuple for a t-literal tlit, built on pred, and let QT be its query tuple. Each of the three basic operations on tuples includes the generation of a current/answer tuple for a t-literal or for a target list.

If pred denotes a base relation, the horizontal operation on CT amounts to resolving a goal against a fact; this operation is essentially a join/projection. Let NEXT be the construct following tlit in its compiled rule. NEXT is indifferently a t-literal or a target list.

Definition (Horizontal operation). From a current tuple CT and a tuple T for pred which matches QT, generate an intermediate tuple IT by projecting (CT, T) over the *arguments of NEXT.\(^9\) lcont(IT) is set equal to lcont(CT).

Let pred be now a virtual predicate.

The top-down operation amounts to resolving a subgoal SG against a rule. It is executed for each compiled rule (R, w) for (pred, w). Let tlit\(_1\) be the first t-literal in (R, w).

Definition (Top-down operation). Select CT if it is admissible and if QT matches the head of (R, w). If CT is selected, generate a tuple CT\(_1\) by projection over the *arguments of tlit\(_1\).\(^10\) Set lcont(CT\(_1\)) = lcid(CT).

The bottom-up operation consists of returning answers to admissible and non-admissible current tuples. NEXT is defined as for the horizontal operation.

Definition (Bottom-up operation). From a current tuple CT and an answer tuple AT such that lcont(AT) = lcid(CT), generate an intermediate tuple IT by projecting (CT, AT) over the *arguments of NEXT. lcont(IT) is set equal to lcont(CT).

Observe that the combined definitions of the bottom-up operation and of the attribution of lcid ensures that an answer for a query tuple QT is returned to all its admissible and non-admissible occurrences.

\(^8\) This first occurrence was necessarily admissible.

\(^9\) This is possible as, by construction, any *argument *X in NEXT appears either as a *argument *X or as a free variable X in tlit.

\(^10\) This is possible as, by construction, each *argument of tlit\(_1\) is a *argument in the head of (R, w).
5.4.3. On range-restricted databases

We do not prove the following proposition, given here for the record.

5.2. Proposition. If the database is range-restricted (Hypothesis H2), then any current, query or answer tuple manipulated by QoSaQ, is ground.

5.4.4. Canonic mapping between QoSaQ tuples and SLD-AL goals

In this section, we properly describe the canonic mapping from tuples manipulated by QoSaQ to goals labeling the nodes of an SLD-AL tree.

Let \( \langle CT_0, \ldots, CT_n \rangle \) be a sequence of current tuples where \( CT_i \ (i > 1) \) is a direct descendant of \( CT_{i-1} \) and is a current tuple for \( \text{lit}_{j_i} \) in the compiled rule \( (R_i, \omega_i) \).

Let us attach each \( CT_i \) to a node \( N_i \) of an SLD-AL tree, where \( N_i \) is a descendant of \( N_{i-1} \). We want to construct the following structure:

\[
\begin{align*}
G_0: & \quad SG_0 \\
N_1: & \quad (SG_1, Set_1, \sigma_1) \\
& \quad \vdots \\
N_i: & \quad (SG_i, Set_i, \sigma_i) \\
& \quad \vdots \\
N_n: & \quad (SG_n, Set_n, \sigma_n)
\end{align*}
\]

\( Set_i \ (i = 1, \ldots, n) \) is \( Set_i \sigma_i \) where:

1. \( Set_i \ (i = 1, \ldots, n) \) is the set \( \{ \text{lit}_{j_i+1}, \ldots, \text{lit}_{4_i} \} \) of literals following \( \text{lit}_{j_i} \) in \( R_i, \omega_i \) (cf. step 4 of our compilation process).
2. \( \sigma_i \ (i = 1, \ldots, n) \) is \( \sigma^2_i \sigma^1_i \):

\[
\sigma^1_i = \{ X/t \} \text{ where: } X \text{ is the } k\text{th free variable of } \text{lit}_{j_i+1}, \text{ and } t \text{ is the constant } c \text{ (resp. the variable } Y) \text{ if the } k\text{th attribute of the target list of } (R_i, \omega_i) \text{ is } c \text{ (resp. } *Y).\]

\( \sigma^2_i \) keeps the bindings provided when linking the rules to each other:

\[
\sigma^2_i = \{ Y/d \} \text{ where: } Y \text{ is the } k\text{th } *\text{argument of } \text{lit}_{j_i},\]
and \( d \) is the \( k\text{th attribute of } CT_i \).

If \( SG_n \) is \( \text{lit}_{l_n} \sigma^2_n \), then the goal \( G_n \) associated to \( CT_n \) is

\[
"SG_n, Set_n[Set_{n-1}[Set_{n-2}[\ldots]\sigma_{n-2}]\sigma_{n-1}]\sigma_{n}"
\]

To associate an SLD-AL goal to an answer tuple \( AT \), replace \( CT_n \) by \( AT \) in the above mapping and do not include \( SG_n \) in the final goal.

\[11\] We suppose that the compiled rules \((R_i, \omega_i)\) do not share variables.
5.5. Redundancy elimination in QoSaQ

Redundancy elimination, in a process manipulating tuples, consists of throwing away some intermediate tuples that are considered to be redundant. In QoSaQ, as in SLD-AL resolution, we distinguish between local and global optimization.

5.5.1. Local optimization

Local optimization consists of discarding an intermediate tuple as soon as a complete answer can be returned to any local subquery, without further processing this tuple. Let $IT_1$ and $IT_2$ be two intermediate tuples either for the same t-literal (current tuples) or for the same target list (answer tuples) in the same compiled rule. Let $G_1$ and $G_2$ be the goals, respectively, associated with $IT_1$ and $IT_2$ by the canonic mapping of Section 5.4.4.

**Proposition** (Local optimization lemma). If (1) $IT_1$ and $IT_2$ have same lcont and (2) $IT_1 = IT_2$, then $G_1$ and $G_2$ are variant. If $IT_1(G_1)$ is generated after $IT_2(G_2)$, then $IT_2(G_2)$ is locally redundant.

**Proof.** This results from a trivial application of the canonic mapping (Section 5.4.4). □

5.5.2. Global optimization: on pure input arguments

In QoSaQ, global optimization consists of detecting when an intermediate tuple is potentially redundant relative to the top query, without being necessarily redundant relative to its (direct) ancestor(s). In the general case (i.e. in the presence of the AL-technique), the implementation of global optimization requires the introduction of a waking mechanism (see Section 4). In this section, we simply discuss how the potential global redundancy of tuples can be detected. A further discussion can be found in [61].

The detection of global redundancy relies on the distinction between the pure and non-pure input arguments of a subgoal. Consider the stack-wise decomposition of the node $N$, as displayed in Table 1. The information stored at the ancestor $N'_i$ of $N$ is $(SG_i, Rest_i, \sigma_i)$, and its frame is $\langle envir \rangle_{SG_i}$. An input argument of $SG_i$ will be called pure if it is not needed any more afterwards, i.e. neither in $Rest_i$ nor as a binding for $SG_{i-1}$. The interest of these pure arguments is that they are not appearing in the goal $G$ labeling $N$, constructed as

\[ SG_n, Set_n, [Set_{n-1}[Set_{n-2}[\ldots]\sigma_{n-2}]\sigma_{n-1}]\sigma_n \].

It follows that if two goals $G$ and $G'$ have same stack-wise decomposition, up to the pure arguments of their respective ancestors $SG_i$ and $SG'_i$, the goals $G$ and $G'$ are variant.

This property on goals carries over to the tuples manipulated by QoSaQ in the following way.
Definition (Pure input arguments). An input argument of a t-literal \( t_{li} \) in a compiled rule \( CR \) is pure, if it appears neither in t-literals following \( t_{li} \) in \( CR \) nor in the target list of \( CR \).

We consider two sequences of current tuples \( \langle CT_0 = G_0, CT_1, \ldots, CT_n \rangle \) and \( \langle CT'_0 = G_0, CT'_1, \ldots, CT'_n \rangle \). Let \( CT_i \) and \( CT'_i \) correspond to the same t-literal in the same compiled rule, labeled by \( t_{li} \). Let \( CT_i \) (resp. \( CT'_i \)) be a direct descendant of \( CT_{i-1} \) (resp. \( CT'_{i-1} \)). Let \( G \) and \( G' \) be the two goals associated respectively to \( CT_n \) and \( CT'_n \) by the canonic mapping.

5.4. Proposition (Global optimization lemma). If (1) \( CT_i \) and \( CT'_i \) do not differ, except maybe on some pure input arguments of \( t_{li} \), and if (2) \( CT_n = CT'_n \), then \( G \) and \( G' \) are variant.

Proof. This follows from a trivial application of the canonic mapping (Section 5.4.4), while taking the above comments into account. ☐

In [61], we discuss the attribution to tuples of so-called global optimization identifiers (gcid) and global context, such that the two tuples \( CT_n \) and \( CT'_n \) fulfill the above condition whenever (1) \( CT_n = CT'_n \) and (2) \( gcont(CT_n) = gcont(CT'_n) \). Testing this condition then becomes very easy.

5.5.3. On the counting methods

The implementation of global optimization becomes particularly easy under two conditions: (A) the AL-technique is not implemented (the process searches SLD trees); (B) no t-literal in \( SR(G_0) \) has neither environment nor non-pure input arguments. Note that (A) implies that termination cannot be guaranteed.

In this case, completeness can be achieved without a waking mechanism, while implementing global optimization (condition (A)). Also, the global optimization lemma is particularly easy to apply (condition (B)), as all the arguments of \( CT_i \) and \( CT'_i \) are pure input arguments. The condition to test reduces to: (1) \( CT_n = CT'_n \) and (2) they have same number of ancestors and their ancestors correspond pairwise to the same t-literal in the same compiled rule. The latter condition can be implemented by encoding into numbers the sequence of t-literals used to reach \( CT_n \) and \( CT'_n \). We refer to [61] for a discussion on these encoding schemes.

The resulting procedure corresponds to the counting procedures [5, 8, 49, 61].

5.5. Example. A compiled rule for the recursive rule given in Example 2.1 is given by (we do not write empty environment tuples):

\[
sg(*X, Y) \rightarrow^r p(*X, U) \rightarrow^h sg(*U, V) \rightarrow^h p'(Y, *V) \rightarrow^h (*Y).
\]

No t-literal has either environment or non-pure input arguments.

We can now compare the execution of QoSaQ with the search of the optimized tree displayed in Fig. 9, on the data displayed on the same figure. As direct
descendants of the root \( \text{sg}(a, Y) \), one derives the two frames \( \text{sg}(b, V) \) and \( \text{sg}(c, V) \). Both have as a direct descendant a (different) occurrence of \( \text{sg}(d, V) \). The two corresponding occurrences of the current tuple \( \langle d \rangle \) are globally redundant as the same sequence of t-literals is used to reach both of them.

The detection of global redundancy by QoSaQ does correspond to the detection of redundancy by comparing nodes 5 and 11 on the SLD tree (Fig. 9).

5.6. On QoSaQ

QoSaQ (global optimization in a QSQ approach) can be seen as the framework resulting from:

1. The non-deterministic execution of top-down, horizontal and bottom-up operations
2. The implementation of the AL-technique on a set of so-called r-predicates. This set is, by default, the set of recursive predicates; this can, however, be changed (by the user, by the compiler).
3. The possibility to discard locally redundant intermediate tuples.
4. The possibility to implement global optimization. This can be done without waking mechanism if the set of r-predicates is empty, and with waking mechanism otherwise.

The following theorem follows from the canonic mapping presented in Section 5.4.4, from the local and global optimization lemmas, and from the completeness of globally optimized SLD-AL resolution.

5.6. Theorem. QoSaQ returns a complete answer to \( G_0 \) whether or not local optimization is performed and whether or not global optimization (with adequate waking mechanism) is performed. If any recursive predicate is a r-predicate, then QoSaQ always terminates. If there is no :-predicate, then (general) global optimization can be implemented without waking mechanism (in this case, however, QoSaQ may not terminate).

The Magic Counting procedure can also be understood as an implementation of QoSaQ, valid only for linear rules [51]. The restrictions they made permit, in particular, a relatively simple waking mechanism.

6. Upper-bound for the QoSaQ procedure

6.1. Worst-case upper-bound

In this section, we provide an upper-bound to the number of tuples manipulated by the QoSaQ procedure.

Let \( G_0 \) be a query. Let \( \text{SR}(G_0) \) be the set of compiled rules relevant to \( G_0 \). We suppose that each virtual predicate is a r-predicate and that locally redundant tuples are systematically removed.
$n$ is the total number of constants in the database.

- $p$ is the number of relevant constants, i.e., of constants which appear in an intermediate tuple manipulated by QoSaQ.

- $k_1$ is the maximal number of arguments in a t-literal or in a target list in $SR(G_0)$.

- $k_2$ is the maximal number of free variables in a t-literal in $SR(G_0)$.

- $a$ is the maximal arity of a virtual predicate.

The following result should be compared with the inherent exponential complexity of SLD resolution.

### 6.1. Theorem

1. The number $N_{ad}$ of admissible query tuples is $O(p^a)$.
2. The number $N_q$ (resp. $N'_q$) of current tuples (resp. not redundant) is $O(p^{a+k_1+k_2})$ (resp. $O(p^{a+k_1})$).
3. The number $N_{ans}$ (resp. $N'_{ans}$) of answer tuples (resp. non-redundant) is $O(p^{a+k_1+k_2})$ (resp. $O(p^a)$).
4. The total number $N$ of tuples manipulated while answering $G_0$ along $SR(G_0)$ is $O(p^{a+k})$, where $k' = k_1 + k_2$.

**Proof.**

1. The number of query or answer relations for the each r-predicate is bounded by a constant (depending on its arity). Each query or answer relation has less than $a$ attributes and contains at most $p^a$ different tuples. Hence, $N_{ad}$ and $N'_{ans}$ are $O(p^a)$.

2. Let us count the number of direct descendants of an admissible query tuple along one compiled rule. First, there is exactly one current tuple for $tlit_1$. This yields at most $p^{k_2}$ current tuples for $tlit_2$, as $tlit_1$ has at most $k_2$ free variables. Among these current tuples, at most $p^{k_1}$ are not redundant, as $tlit_2$ has at most $k_1$ arguments. Inductively, suppose that there are at most $p^{k_1}$ non-redundant current tuples for $tlit_k$, then there are at most $p^{k_1+k_2}$ current tuples for $tlit_{k+1}$, among which at most $p^{k_1}$ are not redundant. The same reasoning applies if $tlit_k$ is the last t-literal (before the target list) of the rule, and there are at most $p^{k_1+k_2}$ answer tuples for each admissible query tuple along one rule, among which at most $p^{k_1}$ are not redundant.

As the length of the rule is bounded by a constant, the number of direct (tuple) descendants of an admissible tuple along a rule is $O(p^{k_1+k_2})$. The number of non-redundant such tuples is $O(p^{k_1})$.

3. As the number of rules defining a given predicate is bounded, we derive from (1) and (2) that $N_q$ is $O(p^{a+k_1+k_2})$, that $N'_q$ is $O(p^{a+k_1})$ and $N_{ans}$ is $O(p^{a+k_1+k_2})$.

4. From (1), (2) and (3), one derives that $N$ is $O(p^{a+k})$. \(\square\)

### 6.2. On focusing on relevant data

The above result is of independent interest as it validates, from the complexity point of view, the interest of a procedure able to focus on relevant data. Query evaluation procedures which interpret rules as specifying a fixed-point computation, or in other words which execute rules in a "plain" bottom-up fashion, are not

---

12 That is, without applying rewrite rules as done in, e.g. the Magic Sets approach [8].
able to focus on relevant data [1] but the number of intermediate tuples they manipulate is always polynomial in $n$ [9, 24]. Let us consider, for instance, a recursive predicate defined exclusively in terms of itself and of base predicates. An upper-bound derives from Immerman’s paper and is $O(n^{2a+k})$ where $k$ is the maximal number of variables in a rule that do not appear in the head.

A first way [6] to exploit this result is to try to rewrite a set of rules into another set of rules whose maximal arity is strictly smaller than $a$. However, this would not ensure a smaller upper-bound, as the constant $k$ may increase meanwhile. Further, it was shown in [6] that such a rewriting is in general not possible.

Our approach is different: instead of decreasing the exponent of the upper-bound, we aim at decreasing the factor: the number of relevant individuals $p$ is always less than the number $n$ of individuals in the database. Further, as $a + k_1 + k_2$ is, syntactically, always less than $2a + k$, our upper-bound is always less than or equal to Immerman’s.

Further, most database queries are connected, in the sense that when answering a top query that is partially instantiated, one can avoid answering totally uninstantiated queries. Technically, we say that $SR(G_0)$ is connected if and only if, for any $t$-literal $(\ldots)_\ast \text{lit}$ it contains, $\ast \text{lit}$ has at least one $\ast$-argument. In this case, the set of relevant individuals can be strictly included in the set of individuals in the database. In most database cases, $p$ is actually much less than $n$. It follows that our upper-bound is much less than Immerman’s in most database cases.

The following proposition expresses the relationship between connected queries and relevant data.

**6.2. Proposition (Focusing on relevant data).** Let $SR(G_0)$ be connected. Let $DB$ be a database with $b$ base relations $base_i$. Then, for any $b$-ary tuple of integers $(s_1, \ldots, s_b)$, there is a database $DB'$ such that $\text{size}(base_i) > s_i$, and such that the number of intermediate tuples manipulated is the same when evaluating $G_0$ over $DB'$ as over $DB$.

**Proof.** $DB'$ is obtained from $DB$ by adding tuples which exclusively contain new constants, and by adding as many of them as is necessary for $base_i$ to contain more than $s_i$ tuples. As $SR(G_0)$ is connected, any subgoal on a base predicate formed by instantiating $\ast$-arguments to “old” constants (i.e. already in $DB$) do not match any of these new tuples. One checks that this implies that no new constant ever appears in an intermediate tuple. Therefore, the number of intermediate tuples manipulated is the same when evaluating $G_0$ over $DB'$ as over $DB$. □

**6.3. On the cost metrics**

One may question whether the number of SLD-AL nodes is a faithful approximation of the real cost of evaluating a query. We note first that it has been advocated elsewhere [4, 8] and we further discuss it.
First, we do not include the cost of finding a/the next successful resolution (searching cost), nor the cost of checking that there are no/no more Is (failure cost). These costs depend on the data structure used: they can be nearly constant, or logarithmic in \( n \), or polynomial in \( n \). Further, these costs are not necessarily uniform, in particular in a page-based environment.

Second, we do not include the cost of checking redundancy or admissibility. As above, this depends on the data structure.

Notwithstanding the importance of the other factors (data structures, memory management), we believe that the number of intermediate tuples being manipulated is a very interesting estimation of the cost of processing a query, as it captures the logical part of the total cost.

7. Conclusion

We addressed the problem of the evaluation of (recursive) queries in deductive databases [17]. First, we developed abstract search spaces (in particular, optimized SLD-AL trees), which are particularly suited for query answering situations. Then, we investigated the techniques that can be used to efficiently represent and manipulate nodes of SLD-AL trees, in the context of deductive databases.

From the practical point of view, a most interesting result of this research is to present a set of basic techniques with which we have been able, in this paper and in the companion paper [61], to draw a geography of some well-known methods for recursive query processing. This geography is obtained by classifying the methods along several criteria: (1) the nature of the search space they construct (either SLD or SLD-AL trees); (2) the optimization strategy they adopt (local vs. global optimization); (3) the nature of the strategy adopted to search the associated search space (constructive vs. iterative-deepening, Appendix D). This geography is displayed in Table 2.

<table>
<thead>
<tr>
<th>Table 2. A geography for recursive query processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>QSQR [58]:</td>
</tr>
<tr>
<td>• Iterative deepening search of SLD-AL trees</td>
</tr>
<tr>
<td>• Local optimization</td>
</tr>
<tr>
<td>Magic sets [5, 8, 50] (or [48]):</td>
</tr>
<tr>
<td>• Constructive search of SLD-AL trees</td>
</tr>
<tr>
<td>• Local optimization</td>
</tr>
<tr>
<td>Counting methods [5, 8, 20] (may not terminate):</td>
</tr>
<tr>
<td>• Search of SLD trees</td>
</tr>
<tr>
<td>• Global optimization</td>
</tr>
<tr>
<td>QoSaq (Section 5, see [61]):</td>
</tr>
<tr>
<td>• Constructive search of SLD-AL trees</td>
</tr>
<tr>
<td>• Global optimization (with waking mechanism)</td>
</tr>
</tbody>
</table>
Further, we have been able to develop the general query answering procedure QoSaQ, which provides a unique framework where a few basic techniques (AL-technique, set-oriented manipulation of nodes, local and global optimization) are given as basic tools, and can be combined according to the desire of the compiler or of the user. QoSaQ is at the core of the DedGin* system [35].

Appendix A. Completeness of SLD-AL resolution

The standard method to prove the completeness of a resolution procedure is first to prove it in the ground case and then to use an appropriate lifting lemma [47, 36, 3]. This method cannot be applied in our case, as the admissibility test applies on non-ground literals. Hence, its outcome is likely to be different when one studies the derivation of a ground answer or of a non-ground answer.

We cannot rely either on the proof of completeness of OLDT resolution [54], as the assumption of a fixed order of literals, which is made there, simplifies the proof a lot. The proofs given by Kerisit for the Alexander method [27, 28] (see also [45]) are remotely related to ours, but do not include global optimization.

A.1. Tasks and ranks

A.1.1. Definition

The very idea of SLD/SLD-AL resolution is to replace the task of proving the answer \( \theta \) to a goal \( G \), by the task of returning an answer \( \theta' \) to a goal \( G' \): this new task is supposed both to be simpler, and to be a necessary step to achieve the initial task.

We define a task \( T_i \) as being a triple \( (Q_i, \theta_i, N) \), where \( N \) is a producer for a \( r \)-subgoal \( SG \), \( Q_i \) is a descendant of \( N \), \( \theta_i \) is a correct answer to the set \( G_i \) of descendants of \( SG \) in \( Q_i \) (if \( Q_i \) is equal to \( N \), we take \( G_i = "SG" \)). \( \eta_i \) will be the current substitution at node \( Q_i \). \( \text{lit}_i \) is supposed to be the selected literal out of \( Q_i \). We will omit the subscript/superscript whenever the context allows this simplification. Intuitively, this triple represents the task of proving, starting from node \( Q_i \), the answer \( \theta_i \) to the producer \( N \).

The complexity of returning an answer \( \theta \) to a goal \( N \) can be measured by the rank of the expression \( G \theta \). Intuitively, the rank of a fact \( F \) (of a conjunction \( C \)) derivable from the database, captures the minimal number of SLD resolution steps necessary to prove it.

Definition. The rank of \( F \) (of \( C \)) is defined by:

- the rank of \( F \) is \( \min_{I \in S} [1 + \text{rank(body}(I))] \), where \( S \) is the set of instances \( I \) of clauses in \( DB \) whose body is derivable and whose head is identical to \( F \);
- the rank of \( C \) is \( \sum_{j=1}^{p} r_j \), if \( C \) is a conjunction of \( p \) facts of respective ranks \( r_j \). The rank of \( (G, \theta) \) is the rank of \( G \theta \), whenever \( \theta \) is a correct answer to \( G \). The rank of a task \( T_i \) is the rank of \( (G_i, \theta_i) \).
Observe that the above double induction is initialized by the instances of facts in $DB$, whose rank is 1.

In the rest of this section, we show how one can decrease the complexity of a task $(Q, \theta, N)$ by: (1) resolving $Q$ against a database clause; (2) resolving $Q$ against a lemma; (3) relying on a previous task $(Q', \theta', N)$, whenever $Q'$ strongly db-subsumes $Q$ relatively to $N$.

A.1. Task, ranks and resolution

The following proposition guarantees that one can always reduce the complexity of an (admissible) task by an SLD resolution.

**A.1. Proposition.** Let $T = (Q_1, \theta_1, N)$ be a task such that $\text{rank}(T_1) > 1$ and $Q_1$ is admissible. Then, there is a clause $Cl(lit_0'=-lit_1', \ldots, lit_n')$ and a task $T_2 = (Q_2, \theta_2, N)$ such that

- $lit_0'$ and $lit_0'$ are unifiable;
- $Q_2$ is obtained from $Q_1$ by resolution against $Cl$;
- $SG\eta_2\theta_2 = SG\eta_1\theta_1$;
- $\text{rank}(T_2) = \text{rank}(T_1) - 1$.

**Proof.** As $lit_0'\theta_1$ is derivable from the database, and by definition of the rank, there is a clause $Cl$ and a substitution $\gamma$ such that $lit_0'\gamma = lit_0'\theta_1$ and $\text{rank}(\text{body}(Cl\gamma)) = \text{rank}(lit_0'\theta_1) - 1$.

As $Cl$ and $\eta_1$ do not share any variable, the domains of $\gamma$ and of $\theta_1$ can be considered as disjoint. We can thus consider their union: let $\theta_2' = \gamma \cup \theta_1$. It follows that $lit_0'\theta_2' = lit_0'\theta_2'$, there exist a mgu $\sigma$ of $lit_0'\theta_2'$ and $lit_0'\theta_2$ and a substitution $\theta_2$ such that $\theta_2' = \sigma\theta_2$. Let us show that $\theta_2$ is the substitution we are looking for.

Any literal of $G_2$ is of the form $lit_j\sigma (j = 1, \ldots, p)$ or $lit_j\sigma (j = 1, \ldots, q)$. Using the properties of the substitutions we introduced, we have:

$$lit_j^1\sigma\theta_2 = lit_j^1\theta_2 = lit_j^1\theta_1, \quad lit_j^1\sigma\theta_2 = lit_j^1\theta_2 = lit_j^1\gamma.$$

First, it follows that $lit_j^1\sigma\theta_2 (j = 1, \ldots, p)$ and $lit_j^1\sigma\theta_2 (j = 1, \ldots, q)$ are derivable from $DB$. Hence, $\theta_2$ is a correct answer to $G_2$.

Second, the ranks obey the following sequence of equalities:

$$\text{rank}(G_2\theta_2) = \sum_{j=1}^{p} lit_j^1\sigma\theta_2 + \sum_{j=1}^{q} lit_j^1\sigma\theta_2$$

$$= \text{rank}(lit_0'\theta_1) - 1 + \sum_{j=1}^{p} lit_j^1\theta_1$$

$$= \text{rank}(G_1\theta_1) - 1.$$

Third, as $SG\eta_1$ does not share any variable with $Cl$, $SG\eta_1\theta_1 = SG\eta_1\theta_2'$. By definition of $\eta_2$ as being $\eta_1\sigma$, we obtain $SG\eta_1\theta_1 = SG\eta_2\theta_2$. \(\square\)
A.1.3. Tasks, ranks and the AL-technique

The following proposition guarantees that the AL-technique does not increase the complexity of achieving a task.

A.2. Proposition. Let $T_1 = (Q_1, \theta_1, N)$ and $Q_1$ be non-admissible. Then, there exists a task $T_2 = (Q_2, \theta_2, Q_2)$ such that:

1. $Q_2$ is a producer; the literal $lit_0^2$ selected at $Q_2$ is admissible and more general than $lit_0^1$; $Q_2$ is active;
2. $lit_0^2 \theta_2 = lit_0^1 \theta_1$;
3. $\text{rank}(T_2) \leq \text{rank}(T_1)$.

Proof. Point (1) is given by the definition of (globally) optimized SLD-AL resolution. As for point (2), $lit_0^1 \theta_1$ is derivable from the database; hence, as $lit_0^2$ is more general than $lit_0^1$, there exists an answer $\theta_2$ to $lit_0^2$ such that $lit_0^2 \theta_2 = lit_0^1 \theta_1$. Point (3) follows from the fact that the rank of $(G_1, \theta_1)$ is necessary greater than or equal to the rank of $lit_0^1 \theta_1$, hence of $lit_0^2 \theta_2$, i.e. of $(G_2, \theta_2)$. □

The following proposition can be proved as Proposition A.1.

A.3. Proposition. Let $T_1 = (Q_1, \theta_1, N)$ be a task such that $\text{rank}(T_1) > 1$ and $Q_1$ is non-admissible. If lem is a lemma more general than $lit_0^1 \theta_1$, then there is a task $T_2 = (Q_2, \theta_2, N)$ such that

1. $Q_2$ is obtained from $Q_1$ by resolution against lem;
2. $SG\eta_2 \theta_2 = SG\eta_1 \theta_1$;
3. $\text{rank}(T_2) < \text{rank}(T_1)$.

A.1.4. Task, ranks and redundancy

Let $T_1 = (Q_1, \theta_1, N)$ be a task where $Q_1$ is redundant relatively to $N$. The following proposition guarantees that not resolving the node $Q_1$ does not increase the complexity of returning $\theta_1$ to $N$.

A.4. Proposition. There exists a task $T_2 = (Q_2, \theta_2, N)$ such that:

1. $Q_2$ was created before $Q_1$; $Q_2$ strongly db-subsumes $Q_1$ relatively to $N$; $Q_2$ is active;
2. $SG\eta_2 \theta_2 = SG\eta_1 \theta_1$;
3. $\text{rank}(T_2) \leq \text{rank}(T_1)$.

Proof. By definition of "redundancy relative to $N"", there exists a set of nodes $Q$ which were created before $Q_1$ and strongly db-subsume $Q_1$ relatively to $N$. To pick
up an active one, say $Q_2$, it is enough to take the first such node that has been created: $Q_2$ is necessarily active, as otherwise there would exist a node $Q'$ created before $Q_2$ (hence before $Q_1$) and that would strongly db-subsume $Q_2$ (hence $Q_1$) relative to $N$. Point (1) is thus proved. Let $G_2$ be “$\text{lit}_1^2, \ldots, \text{lit}_n^2$”. By definition of strong db-subsumption, there is a subset of the literals in $G_1$, subset that we note “$\text{lit}_1^1, \ldots, \text{lit}_n^1$”, and a substitution $\sigma$ such that $\text{lit}_i^1 = \text{lit}_i^2 \sigma$ ($i = 1, \ldots, n$) and $SG_{\eta_1} = SG_{\eta_2} \sigma$. Let $\theta_2 = \sigma \theta_1$. It follows that $\theta_2$ is an answer to $Q_2$ such that $SG_{\eta_2} \theta_2 = SG_{\eta_1} \theta_1$. Further, the rank of $(G_2, \theta_2)$ is less than or equal to the rank of $(G_1, \theta_1)$, as the facts in $G_2 \theta_2$ are in a one-to-one correspondence with the facts of a subset of $G_1 \theta_1$. □

**On strong subsumption rather than mere subsumption**

Note that strong db-subsumption, rather than mere db-subsumption, is critical, as it guarantees that $\text{rank}(T_2)$ remains less than or equal to $\text{rank}(T_1)$. This cannot be guaranteed with mere db-subsumption. See also the comments of Appendix E.

**A.2. Completeness of globally optimized SLD-AL resolution**

Theorem 4.5 follows from the following proposition by taking $N = Q_1 = G_0$.

**A.5. Proposition.** For any task $T_1 = (Q_1, \theta_1, N)$ on a globally optimized SLD-AL tree, there is a node $P$, descendant of $N$ on the tree, proving a lemma for $SG$, more general than $SG_{\eta_1} \theta_1$.

**Proof.** The proof is made by induction on the rank of a task. According to the definition of globally optimized SLD-AL resolution, we consider that only active nodes can be admissible/non-admissible.

**Initial step**

Let $\text{rank}(T_1) = 1$.

(a) $Q_1$ is admissible (hence active). As $\text{lit}_0 \theta_1$ is of rank 1, there is a fact $F$ in the database that is more general than $\text{lit}_0 \theta_1$. The resolution of $\text{lit}_0$ against $F$ proves a lemma for $SG$ that is more general than $SG_{\eta_1} \theta_1$.

(b) $Q_1$ is non-admissible (hence active). As in Proposition A.2, one constructs a task $T_2 = (Q_2, \theta_2, \eta_2)$. We have $\text{rank}(T_2) \leq \text{rank}(T_1)$; hence, they are both equal to 1. Further $Q_2$ is admissible and active. By (a) above, there is a node $P_2$ lower than $Q_2$ returning a lemma $\text{lem}$, more general than $\text{lit}_0 \theta_2$, i.e. than $\text{lit}_1 \theta_1$. The resolution of $\text{lit}_0$ against $\text{lem}$ proves a lemma for $SG$ that is more general than $SG_{\eta_1} \theta_1$.

(c) $Q_1$ is asleep. As in Proposition A.4, one constructs a task $T_2 = (Q_2, \theta_2, N)$, such that $Q_2$ is active and $\text{rank}(T_2) = 1$. By (a) and (b) above, there is a success node $P$ lower than $N$ on the tree, returning an answer to $SG$ more general than $SG_{\eta_2} \theta_2$, i.e. than $SG_{\eta_1} \theta_1$. 


Induction step

(a) Let $Q_1$ be admissible (hence active). Hence, there is a task $T_2 = (Q_2, \theta_2, N)$ which is constructed as in Proposition A.1 such that $\text{rank}(T_2) = \text{rank}(T_1) - 1$. By induction hypothesis, there is a success node $P$ on the tree, returning an answer more general than $SG\eta_2\theta_2$, i.e. than $SG\eta_1\theta_1$.

(b) Let $Q_1$ be non-admissible (hence active). As in Proposition A.2, one constructs a task $T_2 = (Q_2, \theta_2, Q_2)$, where $\text{rank}(T_2) \leq \text{rank}(T_1)$ and $Q_2$ is both admissible and active. By (a) above, there is a node $P_2$ lower than $Q_2$ returning a lemma $\text{lem}$, more general than $\text{lit}_2^2\theta_2$, i.e. than $\text{lit}_1^1\theta_1$. As in Proposition A.3, there is a task $T_3 = (Q_3, \theta_3, N)$ such that $\text{rank}(T_3) < \text{rank}(T_1)$. By induction hypothesis, there is a success node $P$ lower than $N$, returning an answer more general than $SG\eta_3\theta_3$, i.e. than $SG\eta_1\theta_1$.

(c) $Q_1$ is asleep. As in Proposition A.4, one constructs a task $T_2 = (Q_2, \theta_2, N)$, such that $Q_2$ is active and $\text{rank}(T_2) < \text{rank}(T_1)$. By (a) or (b) above, or by induction hypothesis, there is a success node $P$ lower than $N$ on the tree, returning an answer to $SG$ more general than $SG\eta_2\theta_2$, i.e. than $SG\eta_1\theta_1$. \[
\]

A.3. Completeness of SLD, optimized SLD

An optimized SLD tree is a (globally) optimized SLD-AL tree where no node is ever non-admissible. Accordingly, the proof of the completeness of optimized SLD-AL resolution holds for optimized SLD resolution. In this case, only the root is ever a producer. Point (b) in the initial step and in the induction step are to be omitted in the proof.

The completeness of SLD resolution derives from the completeness of optimized SLD.

Appendix B. Soundness of SLD, SLD-AL resolution

B.1. Proposition. For any proof segment $N(SG)-P$, returning a lemma $\text{lem}$, on any SLD or SLD-AL tree, $\text{lem}$ is derivable from $DB$.

Proof. We introduce a measure $\text{comp}$ of the “complexity” of $N-P$ and we prove the result by induction on $\text{comp}$.

- If $SG$ is non-admissible and resolved with $\text{lem}'$, $\text{comp}(N-P)$ is 1 plus the $\text{comp}$ of the first proof segment proving a lemma more general than $\text{lem}'$.
- If $SG$ is resolved against “$C:\text{lit}_0$--$\text{lit}_1$, . . . , $\text{lit}_q$”. $N-P$ can be divided into $q$ proof segments $N_{i-1} - N_i$ ($0 < i \leq q$). $\text{Comp}(N-P)$ is the sum of the $\text{comps}$ of $N_{i-1} - N_i$ plus 1.

If $\text{comp} = 1$, $SG$ is resolved against a fact in $DB$. Hence, by soundness of SLD, $\text{lem}$ derives from $DB$.

Suppose all results proved on segments of $\text{comp}$ strictly less than $c$ are correct. Let $N-P$ be a proof segment of $\text{comp}$ equal to $c$. If $SG$ is non-admissible, let $\text{lem}'$...
be the lemma on which SG is resolved between N and P. lem' derives from DB by induction hypothesis. As lem is less general than lem', it also derives from DB. Finally, suppose SG is admissible. Let lem be the lemma proved by each of the $N_{i-1} - N_i$s. By induction hypothesis, lem, derives from DB. lem can be obtained by resolving C against these q lemmas. Hence, lem derives from DB. □

Appendix C. Finiteness of SLD-AL trees in the finite function-free case

Proof. Let the d-depth of a node denote its number of ancestors (N is an ancestor of M iff M is a descendant of N). Let a branch B be any sequence of nodes $<N_0, \ldots, N_p>$ where $N_0$ is the root and $N_i$ is a child of $N_{i-1}$ on the SLD-AL tree. The number of nodes of d-depth d in a branch B, is bounded by $l^d$, where l is the maximal number of literals in the body of a clause (by induction on d):

- $d = 1$. Let $G_0$ be resolved against the clause C in B. B contains at most one node of d-depth equal to 1 for each literal in the body of C. Hence B contains less than $l$ nodes of d-depth equal to 1.

- Induction step. A node P is of d-depth d, if and only if its direct ancestor N is of d-depth $d - 1$. By the same argument as for $d = 1$, N has at most l direct descendants in B. By induction hypothesis, there are at most $l^{d-1}$ nodes of d-depth $d - 1$; hence there are at most $l^d$ nodes of d-depth d.

The d-depth of literals in B is bounded by a constant. First, note that, whenever the database is finite and function-free, there is no infinite set S of literals such that no literal in S is an instance of another literal in S. Let s be the size of the largest such set. If there were a node P of d-depth greater than s, then this node would have more than s ancestors. Hence, by definition of s, one of these ancestors would be non-admissible. However, non-admissible nodes do not have any direct descendants (no descendants at all), as they are resolved against lemmas. Hence, there is no node of d-depth greater than s.

It follows that the length of a branch B is bounded (by $l^s$).

Finally, each node N has a finite (bounded) number of children (N can be resolved only against a finite number of clauses/lemmas). Hence, the number of nodes on an SLD-AL tree is finite. □

Appendix D. Constructive vs. depth-first searches of SLD-AL trees: QSQR

SLD-AL trees are complex structures to search, as a lemma lem to be resolved against a non-admissible goal G may be proved only after G was created. Therefore, two solutions can be adopted.

The first one, adopted in QoSaQ, consists in constructively searching the SLD-AL tree. This requires that the goal G be stored until all the relevant lemmas lem have been produced and resolved against G. The storage techniques that we have developed in Section 5 for SLD/SLD-AL goals, allow such a constructive search for QoSaQ.
The second solution, adopted in QSQR and implemented in DedGin, tries to take maximal advantage of depth-first techniques (and of their ease of implementation) to search SLD and SLD-AL trees. In this approach, the goal $G$ is not stored until all the lemmas have been found. Rather, $G$ is repeatedly regenerated to be resolved each time, potentially with new lemmas.

The search spaces associated with such a strategy are called here straight SLD-AL trees. The construction of a straight SLD-AL tree differs from that of an SLD-AL tree as follows: a non-admissible goal $G$ is resolved only against the lemmas available at the time $G$ is created; then $G$ is no more considered during the construction of this straight SLD-AL tree; hence $G$ can be discarded. It follows that depth-first search strategies can be used to search straight SLD-AL trees. The completeness of the associated answering procedure is achieved by constructing a sequence of straight SLD-AL trees: each tree in the sequence makes use of lemmas proved on any of the previous trees in the sequence.

This solution, which, for short, we call an iterative-deepening search involves redundant computation, as the same goal must be reconstructed again and again. On the other hand, it is economical of space, as we do not need to store the goal $G$.

Definition. A straight SLD-AL tree is constructed as an SLD-AL tree, except that (i) the set of lemmas $L$ is initialized to an arbitrary finite set $L_{lem}$; (ii) a non-admissible goal $G$ is exclusively resolved against lemmas already in $L$ when $G$ is created.

A sequence of SLD-AL trees is a sequence of straight SLD-AL trees built with initial set of lemmas $L_i$ ($i = 0, \ldots$), where $L_0$ is empty and $L_n$ ($n > 0$) is the set of lemmas proved on the $(n - 1)st$ tree.

D.1. Theorem (Completeness of straight SLD-AL resolution). For any goal $G_0$, for any sequence of straight trees, for any correct answer $\theta$ to $G_0$, there is a success node in a straight SLD-AL tree of the sequence which returns an answer to $G_0$, more general than $\theta$.

Further, if $L_{n_0}$ is equal to $L_{n_0 - 1}$, then the $n_0$th straight SLD-AL is complete: for any correct answer $\theta$ to $G_0$, there is a success node on the $n_0$th straight SLD-AL tree that returns an answer more general than $\theta$.

The following points are worth mentioning: (i) each straight SLD-AL tree in a sequence can be built using different search strategies and selection functions; (ii) a “limit” $n_0$ exists as soon as the database is finite and function-free; (iii) a straight SLD-AL tree is finite as soon as the database is finite and function-free. Theorem D.1 can be proved along the techniques provided in Appendix A. The QSQR algorithm [58] performs a mere search of a sequence of locally optimized SLD-AL trees.

13 Called SLD-AL trees in [60].
14 Although it is not quite the same.
Definition (The QSQR procedure). Given a goal $G_0$, search a sequence of SLD-AL trees of root $G_0$, using an FRD-based selection function and not adding locally redundant nodes. Stop as soon as $L_n$ is equal to $L_{n-1}$.

This definition of the QSQR algorithm corrects a mistake in the presentation of QSQR in [58]. The comparison of subgoals must be local to each straight SLD-AL tree, and not global as said in [58]. See [60] for a modification.

Appendix E. On the optimality of redundancy detection

We investigate how optimal is redundancy elimination based on db-subsumption. Let $N_1$ (goal $G_1$) and $N_2$ (goal $G_2$) be two descendants of $N$. Let $\eta_1$ and $\eta_2$ be their respective current substitutions.

E.1. Proposition. The two following statements are equivalent:

1. $N_1$ is db-subsumed by $N_2$ relative to $N$;
2. for any database $DB$, whenever $G_1$ admits an answer $\theta_1$ from $DB$, $G_2$ admits an answer $\theta_2$ from $DB$, such that $SG_{\eta_2, \theta_2}$ is more general than $SG_{\eta_1, \theta_1}$.

The proof of $(2 \rightarrow 1)$ uses the fact that any database $DB$ is allowed, or in other words, that no knowledge at all is assumed about $DB$. This assumption is extreme, as when considering $N_1$ and $N_2$, one knows at least that $C_1$ and $C_2$ (where $C_i : SG_{\eta_i} : -G_i$) are derivable from the database. For instance, let $G_0$ be $p(U, V, W)$ and let the database contain the two clauses $C_1 : "p(X, Y, Z):-p(Z, X, Y)."$ and $C_2 : "p(X, Y, Z):-p(Y, Z, X)."$. By resolving $G_0$ against $C_1$ and $C_2$, one obtains nodes labeled by $p(W, U, V)$ and $p(V, W, U)$ ($X/U, Y/V, Z/W$) which do not db-subsume each other. However, knowing that $C_1$ and $C_2$ are equivalent, one could safely discard one of them.

This remark relativizes the scope of Proposition E.1. However, this result indicates that further redundancy elimination on SLD trees cannot be based on a mere comparison of the goals/nodes, but should include more general knowledge about clauses in the database.

Proof. We prove only $(2 \rightarrow 1)$ as $(1 \rightarrow 2)$ is classical (see [37]). See also Appendix A.

Let $Y_1, \ldots, Y_n$ be the variables of $G_1$. Let $\theta_1$ be the substitution $\{Y_i \mid a_i\}$ where the $a_i$s are new, distinct, constants. Let $DB$ be $\{lit_i^1 \theta_1 \mid lit_i^1\}$ is a literal of $G_1$. Clearly, $G_1$ admits the answer $\theta_1$ from $DB$. By hypothesis, $G_2$ admits an answer $\theta_2$ from $DB$, such that $SG_{\eta_2, \theta_2}$ is more general than $SG_{\eta_1, \theta_1}$. Let $\gamma$ be a substitution such that $SG_{\eta_2, \theta_2} \gamma = SG_{\eta_1, \theta_1}$.

As $G_2$ admits the answer $\theta_2$, we have the following property: for any literal $lit_k^2$ in $G_2$, $lit_k^2 \theta_2$ derives from $DB$. From the definition of derivability, and as $DB$ contains only ground facts, one concludes that $lit_k^2 \theta_2$ is ground and that there exists a literal $lit_k^1$ in $G_1$ such that $lit_k^2 \theta_2 = lit_k^1 \theta_1$. 
Let \( \theta'_2 \) be the substitution \( \theta_1 \gamma \). From the two above paragraphs, we conclude that

1. \( S \eta_2 \theta'_2 = S \eta_1 \theta_1 \), and
2. \( \text{lit}^+_2 \theta'_2 = \text{lit}^+_1 \theta_1 \).

Let \( Z_1, \ldots, Z_p \) be all the variables in \( G_2 \) and \( S \eta_2 \). Let \( t[a_i | Y_i] \) be the term obtained from \( t \) by replacing any occurrence of \( a_i \) (\( i = 1, \ldots, n \)) by \( Y_i \). Let \( \sigma \) be the substitution: \( \{Z_j | t[j[a_i | Y_i] \text{ if } Z_j | t \text{ is in } \theta'_2 \} \). From the two properties of \( \theta'_2 \), we easily derive the two following properties of \( \sigma \):

1. \( S \eta_2 \sigma = S \eta_1 \) and
2. \( \text{lit}^+_2 \sigma = \text{lit}^+_1 \).

We have thus exhibited the substitution \( \sigma \) that fulfills the two requirements for proving that \( N_1 \) is db-subsumed by \( N_2 \) relative to \( N \). \( \Box \)

If one could fully rely on db-subsumption to discard redundant nodes on SLD and SLD-AL trees, Proposition E.1 would ensure that this redundancy is optimal in the sense discussed above. Unfortunately, relying on the most general definition we gave may still lead to incompleteness, and we had to use strong db-subsumption.

Intuitively, this incompleteness is due to a mismatch between the sequential way resolution proceeds to answer a goal and the "sort of order" between goals induced by (db-)subsumption between goals (see Example E.2). On the one hand, resolution proceeds by replacing a goal (or a subgoal) by another one which will need less steps to answer. On the other hand, a goal \( G_1 \) may be subsumed by \( G_2 \), and yet \( G_2 \) may be more complex to answer (whatever this means) than \( G_1 \). For instance, "p(a)" is subsumed by "p(X), p(Y)". However, if the database contains the fact \( p(a) \), then answering "p(a)" can be done in one step, whereas answering "p(X), p(Y)" cannot be done in less than two steps. Strong db-subsumption solves this mismatch by requiring that \( \sigma \) contains more literals than \( \eta_2 \).\(^{15}\)

E.2. Example

\[
\begin{align*}
p(X) & : -r(Y), r(Y'), s(Z), q(X) \\
p(X) & : -s(U), r(U'), s(V), q(X) \\
r(W) & : -q(W) \\
s(W) & : -q(W) \\
q(d) & .
\end{align*}
\]

We can construct the following SLD tree, where nodes are numbered according to the order of search:

\[
\begin{align*}
(1) & \quad p(X) \\
(2) & \quad r(Y), r(Y'), s(Z), q(X) \\
(3) & \quad s(U), r(U'), s(V), q(X) \\
(4) & \quad q(Y), r(Y'), s(Z), q(X) \\
(5) & \quad q(U), r(U'), s(V), q(X) \\
\vdots & \quad \vdots \\
X = d & \quad X = d
\end{align*}
\]

\(^{15}\) The use of factorization to solve this problem [37] is beside the scope of this paper.
Both (4) and (5) are db-subsumed by both (2) and (3). Thus, if they were declared redundant, the answer $X = d$ would not be retrieved and the tree would not contain any success node.

Acknowledgment

It is a pleasure to acknowledge the support and the advice of Jean-Marie Nicolas. I would like also to thank Herve Gallaire, and my colleagues at ECRC, especially in the KB group, for the nice and stimulating environment they provide. Special thanks go to Alexandre Lefebvre, whose work on the DedGin* system has increased our familiarity with the notions presented here. Exchanges with H. Tamaki and R. Ramakrishnan have helped clarifying the relationship respectively between SLD-AL and OLDT resolutions, and between QoSaQ and GSMS/GSC. Comments from and discussions with Bernard Lang have proved particularly helpful.

References


[22] R. Hill, LUSH-resolution and its completeness, DCL memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974; cited in [36].


Recursive query processing


