Extension of a Datalog Reasoner with Top-Down Evaluation

Bachelor Thesis

Christoph Fuchs
Ing.-Sigl-Strae 26
A-6020 Innsbruck
Matrikelnr.: 0416596

Supervised by Dr. Elena Simperl
and co-supervised by Uwe Keller, Barry Bishop

Innsbruck, September 22, 2008
Abstract

The Integrated Rule Inference System IRIS is a Datalog reasoner capable of different bottom-up evaluation strategies. This thesis focus is on extending the reasoner with three new evaluation strategies which take a very different approach compared to the bottom-up evaluation strategies. The introduced strategies are SLD, SLDNF and OLDT, which are all top-down strategies.

It will be elucidated how the evaluation of a Datalog program works, with focus on the top-down evaluation strategies. The different strengths and weaknesses of each algorithm will be pointed out and put in contrast. To get a better understanding of the procedures, many examples are used throughout the thesis.
Acknowledgements

I would like to thank my co-supervisor Uwe Keller, who helped me understand the basics of top-down reasoning, and Barry Bishop who had always time for me when I got stuck at implementing one of the algorithms.
1 Introduction

1.1 Problem Statement ........................................... 1

1.2 Prolog ......................................................... 1

1.2.1 Horn clauses ............................................... 3

1.2.2 Syntax Notes .............................................. 3

1.3 Datalog ......................................................... 4

1.3.1 Datalog$^R$ ................................................ 4

1.3.2 Datalog$^-$ .................................................. 5

1.3.3 Datalog$^f$ .................................................. 6

1.4 Evaluation Strategies implemented in IRIS .................. 6

1.4.1 Naive Evaluation ........................................... 6

1.4.2 Semi-Naive Evaluation .................................... 8

1.5 Contribution of the Project .................................. 8

2 Solution

2.1 Top-Down Evaluation .......................................... 11

2.1.1 Literal Selection ............................................ 11

2.1.2 Substitution and Unification ............................... 13

2.1.3 Basic Top-Down Algorithm ............................... 15

2.1.4 Graphical Representation .................................. 16

2.2 SLD Resolution ................................................ 17

2.2.1 Definition: SLD tree ...................................... 18

2.2.2 Example: SLD Resolution .................................. 18

2.3 SLDNF Resolution ............................................. 21

2.3.1 Definition: SLDNF tree .................................... 21

2.3.2 Example: SLDNF Resolution ............................... 22

2.4 OLDT Resolution ............................................... 25

2.4.1 Memo Predicates ............................................ 26

2.4.2 Definition: OLDT structure ............................... 26

2.4.3 Example: OLDT Resolution ............................... 28
3 Conclusions 33
  3.1 Bottom-up versus top-down 33
  3.2 Implementation Issues 34
  3.3 Final Words 35
Chapter 1

Introduction

1.1 Problem Statement

This thesis describes the process of extending the Datalog reasoner IRIS\(^1\) with the top-down evaluation strategies SLDNF\(^2\) and OLDT\(^3\). IRIS is an extensible reasoning engine for expressive rule-based languages which currently supports naive and semi-naive evaluation of Datalog programs [2].

Since IRIS evaluates Datalog programs, a good understanding of Prolog and Datalog is necessary to understand the evaluation strategies. The following chapter will give a short introduction to Prolog and Datalog. Further, the bottom-up evaluation strategies which are already implemented in IRIS are presented.

1.2 Prolog

Prolog is a logic programming language that has its roots in formal logic. It is a general purpose language often associated with artificial intelligence and computational linguistics. Unlike many other programming languages, Prolog is declarative: The program logic is expressed in terms of relations, and execution is triggered by running queries over these relations. Relations and queries are constructed using Prolog's single data type, the term. Relations are defined by clauses. Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses, a Turing-complete subset of first-order predicate logic. [9]

\(^1\)Integrated Rule Inference System
\(^2\)Linear resolution with selection function for definite clauses augmented with negation as failure [1]
\(^3\)Ordered selection strategy with linear resolution for definite clauses with tabulation [1]
Prolog defines two types of clauses: Rules and facts. A rule consists of a head and a body. The head of the rule is true if the body of the rule is true. A rule's body consists of calls to predicates, which are called the rule's goals or sub-goals. [9]

Head :- Body.

The predicate calls in a rule's body form atoms. An atom consists of a predicate symbol and a tuple, which can contain terms. A term is either a variable a constant or a constructed term (function symbol applied to a list of terms). In propositional logic all tuples are empty, since there are no terms. In this case the empty tuple () is simply omitted and only the predicate symbols are denoted. A literal is either a positive or a negative atom. Since we are only dealing with Horn clauses, multiple literals form either a rule body or a query.Clauses with empty bodies are called facts. An example of a fact is:

geek('Iain').

which says 'Iain is a geek' and is equivalent to

geek('Iain') :- true.

Queries are rule bodies. In IRIS, a ?- is put before the literal(s) to differentiate between rules and queries.

?- geek('Iain').
true

The above query reads: 'Is Iain a geek?'

Given the fact from above this query yields back an empty tuple which is interpreted as true. If there are variables in the query, the evaluated tuples contain the answers.

?- geek(?X).
X = 'Iain'

The conjunction of literals is indicated by a comma, as in

computer-geek(?X) :- geek(?X), loves-computers(?X).

where the two predicates geek and loves-computers are in conjunction to form the new predicate computer-geek. Disjunction is sometimes denoted by a semi-colon. In this thesis, multiple clauses for the same literal are used as disjunction, which is equivalent.

Figure 1.1 explains the different notations using an example rule. Remember that a clause can either be a rule (as shown in this figure) or a fact, and a query simply is a rule body. Also remember that a term is not necessarily a variable, but can also be a constant or a constructed term.
1.2. PROLOG

A Horn clause is a clause with at most one positive literal. They are named after the logician Alfred Horn, who first pointed out the significance of such clauses in 1951, in an article published in the Journal of Symbolic Logic. [8]

A Horn clause with exactly one positive literal is a definite clause. A Horn clause has the following form:

\[ \neg p \lor \neg q \lor \ldots \lor \neg t \lor u \]  

Using logical transformations, a Horn clause can be rewritten such that the positive literal holds if all negative literals hold:

\[ (p \land q \land \ldots \land t) \rightarrow u \]  

In this thesis the reverse notation is used, such that the positive literal is written first:

\[ u \leftarrow (p \land q \land \ldots \land t) \]  

Note that the clauses in (1), (2) and (3) are equivalent. They all behave as the procedure: to show \( u \), show \( p \) and show \( q \) and \ldots and show \( t \).

In IRIS, this clause would be written as follows:

\[ u :- p, q, \ldots, t. \]

Horn clauses can be propositional or first order, depending on whether we consider propositional or first-order literals. [8]

A Horn clause with no positive literals is called a goal clause or query. Since the head of a Horn clause is the only positive literal, a Horn clause with no positive literals is the body of a Horn clause. For an example query please see section 1.2.

1.2.2 Syntax Notes

The syntax used in the examples is the actual syntax used by IRIS, which is equivalent to the formal definitions of logic programs. To preserve readability of definitions, formal writing is used when it comes to defining procedures and
expressions. An assignment ← is represented by :- in most cases. If the head or the body of the clause is empty, the arrow is replaced by ?- or omitted respectively. Variables are denoted by a preceding ?. Please see Table 1.1 for syntax details.

<table>
<thead>
<tr>
<th></th>
<th>Formal</th>
<th>IRIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td>( q(X) \leftarrow r(X) )</td>
<td>( q(?X) :!- r(?X) ).</td>
</tr>
<tr>
<td>Query</td>
<td>( \leftarrow q(X) )</td>
<td>(?!- q(?X) ).</td>
</tr>
<tr>
<td>Fact</td>
<td>( r(3) \leftarrow )</td>
<td>( r(3) ).</td>
</tr>
<tr>
<td>Conjunction</td>
<td>( a(X) \land b(X) )</td>
<td>( a(?X), b(?X) ).</td>
</tr>
<tr>
<td>Disjunction</td>
<td>( a(X) \lor b(X) )</td>
<td>( a(?X), b(?X) ).</td>
</tr>
<tr>
<td>Negation</td>
<td>( \neg b(X) )</td>
<td>( \text{not } b(?X) ).</td>
</tr>
</tbody>
</table>

### 1.3 Datalog

Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog. In contrast to Prolog, it

- disallows complex terms as arguments of predicates, e.g. \( P(1, 2) \) is admissible but not \( P(f(1), 2) \),
- imposes certain stratification restrictions on the use of negation and recursion (see chapter 1.3.2), and
- only allows range restricted variables, i.e. each variable in the conclusion of a rule must also appear in a not negated clause in the premise of this rule. [7]

Datalog is allowing recursion and negation. Since SLD resolution can not handle negation, we define two subsets of Datalog: \( \text{Datalog}^R \), which allows recursively defined rules, but no negation. \( \text{Datalog}^- \), which allows recursion and negated literals. Another Datalog variant is an extension to normal Datalog: \( \text{Datalog}^f \), which allows function symbols.

The following sections will give a short overview of the three Datalog variants. In chapter 2 the proper resolution for each variant will be presented.

#### 1.3.1 Datalog\(^R\)

Recursion is an essential part of logic programs and deductive databases. Many things can be modelled elegantly using recursive rules. To underline the power
of recursive rules, let’s have a look at the following example, which calculates the $N$th Fibonacci number $F$.

\begin{verbatim}
1. fib(0, 1).
2. fib(1, 1).
\end{verbatim}

Note that for the program to work correctly, the two facts fib(0, 1). and fib(1, 1). are essential to terminate the recursion.

Another popular example to show the power of recursion is the ancestor relationship. The ancestor relationship models a family tree by subsequently resolving the parents of the seed until there are no facts left (e.g. the father of Alice’s great-great-grandfather is not in the knowledge base or not known anymore).

\begin{verbatim}
1. ancestor(?X,?Y) :- parent(?X,?Y).
2. ancestor(?X,?Y) :- parent(?X,?Z), ancestor(?Z,?Y).
\end{verbatim}

The query \texttt{?- ancestor(?X, ‘Alice’)} would return all known ancestors of Alice.

### 1.3.2 Datalog

When it comes to negation, some limitations need to be mentioned. Since we are dealing with Horn clauses, negated literals can only appear in the body of a rule, not in the rule head (see section 1.2.1). The evaluation of negated literals is done by negation as failure, as described in section 2.3. Negation as failure works when we assume that everything that is not known is false. This assumption is called the closed world assumption.

**Closed World Assumption**

The closed world assumption is the presumption that what is not currently known to be true is false. The opposite of the closed world assumption is the open world assumption, stating that lack of knowledge does not imply falsity. [6]

Consider the following data, which describes married couples and gender:

\begin{verbatim}
1. playing-game(‘Iain’, ‘Poker’).
2. playing-game(‘Manuel’, ‘Warcraft 3’).
\end{verbatim}

In the closed world assumption, the query \texttt{?- not playing-game(?X, ‘Poker’)} could be evaluated and would yield back (‘Manuel’). However, in the open world assumption, the query could not return any answer. Since the table does not contain every possible name-game tuple, the answer is not known.
Another limitation is the simultaneous positive and negative dependency of a predicate symbol to another predicate symbol. Programs that have such dependencies cannot be evaluated. Such programs are called not stratified.

Stratification
The evaluation strategies presented in this thesis require stratified Datalog programs. If a program is stratifiable, it is considered as safe. Non-stratified programs may have several minimal fixed points containing a given input. For example, the propositional program \( P_1 = \{ p \leftarrow \neg q, q \leftarrow \neg p \} \) has two minimal fixed points (containing the empty instance): \( \{ p \} \) and \( \{ q \} \). Another problem are programs that do not have any fixed point. For example, the propositional program \( P_2 = \{ p \leftarrow \neg p \} \) has no fixed point. In such cases, the SLD, SLDNF and ODLT resolutions will fail, because those strategies try to prove \( p \). Since the resulting proof tree is an infinite failure tree, the proof fails.

1.3.3 Datalog\(^f\)
An extension to Datalog which allows complex terms (also called function symbols or constructed terms) as arguments of predicates is Datalog\(^f\). Function symbols are problematic when rules generate new rules, as in \( p(X) \leftarrow p(f(X)) \). For example, even if a fact \( f(f \ldots f(x) \ldots) \) exists, the query \( \leftarrow p(X) \) will generate an infinite set of sub-goals. In this example, \( \leftarrow p(X) \) will lead to \( \leftarrow p(f(X)) \), which will lead to \( \leftarrow p(f(f(X))) \), and so on. The evaluation strategies presented in this thesis will only terminate if the program has a finite minimal model.

1.4 Evaluation Strategies implemented in IRIS
This section gives a short introduction to the evaluation strategies that are already implemented in IRIS. There are two evaluation strategies present, naive and semi-naive evaluation, which both are bottom-up evaluation strategies.

The basic principle of bottom-up evaluation is to start from known facts, and build the minimal model by applying rules. The output of a bottom-up evaluation is the least fixed point solution to the Datalog equations [5]. This means that a bottom-up evaluation strategy always computes the full minimal model, regardless of the query. Building this model is the most expensive part of the evaluation. This is why a top-down approach is potentially faster in some cases. Bottom-up evaluation is also called forward-chaining, since the direction of the evaluation is forward - from rule head to body.

1.4.1 Naive Evaluation
The following pseudo-code shows the naive evaluation as implemented in IRIS. The algorithm is very similar to the one presented in [5].
1.4. EVALUATION STRATEGIES IMPLEMENTED IN IRIS

```java
while (true) {
    for (rule : stratum) {
        delta = rule.evaluate();
        if (delta != null && delta.size() > 0) {
            if (facts.get(ruleHead).addAll(delta))
                continue;
        }
    }
    break;
}
```

The following example will show informally how the algorithm works.

Example

The simple graph shown in figure 1.2 consists of five edges, $E = \{a, b, c, d, e, f, g\}$. The following program describes the graph:

```prolog
edge('a', 'b').
edge('b', 'c').
edge('d', 'e').
edge('e', 'f').
edge('f', 'g').
```

The knowledge base of this example program consists of the edges $E$ and the set of paths to the neighboring edges $P = \{(a, b), (b, c), (d, e), (e, f), (f, g)\}$. The goal of the computation is to build the minimal model of this program, such that queries like 'Is there a path from a to f' can be evaluated.

![Graphical representation of the example knowledge base](image)

Figure 1.2: Graphical representation of the example knowledge base

By applying the naive algorithm, new tuples are calculated iteratively until no new tuples are added to the set. The results of the iterative computation are shown in table 1.2. Since the edges $E$ remain unchanged, they are not shown.
in the table. Note that a short notation for the tuples is used: \(ab\) instead of \((a,b)\).

As shown in table 1.2, the parts of the results of each iteration are recomputed. This unnecessary computation is eliminated by the semi-naive evaluation.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>1</td>
<td>(ab) bc de ef fg</td>
</tr>
<tr>
<td>2</td>
<td>(ab) bc de ef fg ac df eg</td>
</tr>
<tr>
<td>3</td>
<td>(ab) bc de ef fg ac df eg dg</td>
</tr>
<tr>
<td>4</td>
<td>(ab) bc de ef fg ac df eg dg</td>
</tr>
</tbody>
</table>

### 1.4.2 Semi-Naive Evaluation

A ‘smarter’ bottom-up evaluation strategy is the semi-naive evaluation. It extends the naive evaluation strategy by removing the redundant computation of the previous tuples.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>1</td>
<td>(ab) bc de ef fg</td>
</tr>
<tr>
<td>2</td>
<td>ac df eg</td>
</tr>
<tr>
<td>3</td>
<td>dg</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

The detailed algorithm will not be presented in this thesis, since its focus is on top-down evaluation. For details on the implementation, please refer to [5] or have a look at the IRIS source code\(^4\).

### 1.5 Contribution of the Project

During this project, three new evaluation strategies have been implemented in IRIS. Each strategy lies in a separate package \(\text{topdown.sld, topdown.sldnf}\)

\(^4\)Available online via [http://www.iris-reasoner.org/download](http://www.iris-reasoner.org/download)
1.5. CONTRIBUTION OF THE PROJECT

and `topdown.oldt` respectively). This leads to five different evaluation strategies which can be used for evaluation. Two additional variants are formed by including the `magic sets` transformation and `well-founded` semantics, which are part of IRIS too. Chapter 2 is about top-down evaluation in general. The three newly implemented evaluation strategies get special attention.
Chapter 2

Solution

The following chapter will give a deeper insight into the newly implemented top-down evaluation strategies. First, the basic principles of top-down evaluation will be explained. Section 2.2 is about the SLD resolution algorithm and its implementation. Section 2.3 will describe the SLDNF resolution which builds up on SLD resolution, extending it with \( NAF \)\(^1 \) which allows evaluation of Datalog programs. Section 2.4 describes OLDT, which is the third and final resolution algorithm. OLDT adds tabling to the resolution, which enables the algorithm to handle circular dependencies.

2.1 Top-Down Evaluation

As described earlier in section 1.4, bottom-up evaluation builds the minimal model starting from the facts. Top-down evaluation takes a different approach, starting from the query. The initial query is used to build up a proof tree, which contains all possible sub-goals in respect of the initial query, which is the root (see section 2.1.4). This means that top-down techniques do not necessarily build the minimal model but any those parts which are able to answer the query.

Before the algorithms are explained in detail, the basic principle of the evaluation strategies is explained using an example. To get a better and easier understanding, a graphical representation for top-down evaluation is introduced.

2.1.1 Literal Selection

Since queries can consist of several literals, a query is evaluated step by step for each literal. To define which literal will be processed first, a literal selection rule has to be defined. A literal selection rule \( R \) takes a query \( G \equiv L_1 \land \ldots \land L_n \)

\(^1\)Negation as failure
with \( n > 0 \) and selects a literal \( L_i \) where \( 1 \leq i \leq n \).

There are many possibilities for literal selection. The examples in this thesis use the standard literal selection rule, which always selects the first literal of a query. SLD resolution using the standard literal selection rule is called OLD\(^2\) resolution. An extension of the OLD resolution is presented in chapter 2.4.

**Example: Literal selection**

The following example shows that literals can be selected via different selection rules. Let \( R_1 \) be the standard literal selection rule and \( R_2 \) a selection rule which selects the last literal (the most right-handed literal).

```plaintext
1 q(\{X\}) :- qa(\{X\}), qb(\{X\}).
2 r(\{X\}) :- ra(\{X\}), rb(\{X\}).
3 qa(1). qb(1). ra(2). rb(2).
4 :- q(\{X\}), r(\{X\}).
```

\( R_1 \) will select \( q(\{X\}) \) first, which leads to \( qa(\{X\}), qb(\{X\}) \). Then \( qa(\{X\}) \) will be selected before \( qb(\{X\}) \) and \( r(\{X\}) \), which leads to \( ra(\{X\}) \) and \( rb(\{X\}) \). Using \( R_1 \), the literals in order of selection are

```
q(\{X\}) qa(\{X\}) qb(\{X\}) r(\{X\}) ra(\{X\}) rb(\{X\})
```

Using \( R_2 \), the literals in order of selection are

```
r(\{X\}) rb(\{X\}) ra(\{X\}) q(\{X\}) qb(\{X\}) qa(\{X\})
```

Different literal selection rules can lead to different proof trees (see chapter 2.1.4), which can lead to differing execution times of the algorithm, depending on the tree structure and the way it is traversed.

The evaluation result is independent from the literal selection, as long as safe selection rules are used. Safe selection rules always select a literal and always select the same literal with the same query given (no random effects). In our example, the result of the initial query would be a relation containing two tuples: (1) (2).

**Safe literal selection in Datalog\(^*\)**

The literal selection rules used in SLDNF resolution must not select negative literals that contain variables. The selected literal has to be a positive literal or a negative grounded literal.

The following example shows why this is necessary. Let \( R \) be the standard selection rule. Consider the following Datalog\(^*\) programs \( D_1 \) and \( D_2 \), which are

\(^2\)Ordered selection strategy with linear resolution for definite clauses
2.1. TOP-DOWN EVALUATION

Let $p$ be a query. While the evaluation of $p$ and $D_1$ via $R$ succeeds and yields back true, the evaluation of $p$ and $D_2$ via $R$ yields back false. This is because the standard selection rule $R$ selects $\neg r(x)$ as first literal while evaluating $D_2$. Using negation as failure, $r(x)$ will be the next query, which succeeds immediately. Consequently $\neg r(x) \land q(x)$ fails. [1]

2.1.2 Substitution and Unification

When it comes to evaluating rules, substitution and unification are essential tools. Substitution and unification are used by all evaluation strategies, no matter if the strategies work bottom-up or top-down. A substitution describes a mapping from a set of variables to a set of terms. Unification is a special form of substitution and can be compared with parameter passing in conventional programming. The two concepts of substitution and unification will be described in the following sections.

Substitution

As stated before, a substitution is used whenever it comes to replace variables with terms. Note that a substitution can only change a finite set of variables at once. The result of an applied substitution $\sigma(t)$ of a substitution $\sigma$ on a term $t$ is computed by replacing simultaneously all variables of $t$ with the terms provided by the variable mappings in $\sigma$.

**Example:** Let $\sigma_1 = \{X/f(Y), Y/X\}$ be a substitution, and $p(X, Y)$ a atom. The applied substitution $\sigma_1(p(X, Y))$ produces $p(f(Y), X)$ by substituting $X$ with $f(Y))$, $Y$ with $X$ respectively. Note that the two substitutions happen simultaneously. If the substitutions would be applied one after another, e.g. first $Y/X$ then $X/f(Y)$, the resulting atom would be $p(f(Y), f(Y))$.

**Example:** Let $\sigma_2 = \{X/A, Y/4\}$ be a substitution, and $p(f(X), f(g(Y)), g(X), Y)$ a atom. The applied substitution $\sigma_2(p(f(X), f(g(Y)), g(X), Y))$ would produce $p(f(A), f(g(4)), g(A), 4)$.
A substitution can be either a simple substitution in the form of \( \sigma = \{ \text{variable/term} \} \), the identity substitution \( \phi = \{ \} \) or a concatenation of simple substitutions \( \sigma = (\sigma_1 \circ \ldots \circ \sigma_n) \), a composition.

A composition of two substitutions \( (\sigma_1 \circ \sigma_2) \) is a substitution in the form of \( \sigma_1(\sigma_2) \) (or \( \sigma_2(\sigma_1) \)), since this operation is symmetric).

**Example:** Let \( \sigma_1 = \{ X/Y, A/B \} \) and \( \sigma_2 = \{ B/C, U/V \} \) be substitutions. The resulting composition \( (\sigma_1 \circ \sigma_2) \) is \( \{ X/Y, A/C, U/V \} \).

**Unification**

Unification is the process of creating a substitution for two terms, so that one term is equal to the other one. A mapping to unify two terms is called a unification. If a unification with the stated criteria exists, the two terms are unifiable. Since a simple unification is required for efficient computing, the most general unification or mgu is computed by an unification algorithm. A unification \( \sigma \) unifies two terms \( t_1 \) and \( t_2 \) if \( \sigma(t_1) \equiv \sigma(t_2) \).

A unification \( \sigma \) of \( t_1 \) and \( t_2 \) is a most general unification if for every unification \( \theta \) of \( t_1 \) and \( t_2 \) there is a substitution \( \rho \), such that \( \sigma(t_1) \equiv \theta(\sigma(t_1)) \) and \( \sigma(t_2) \equiv \theta(\sigma(t_2)) \). [4]

**Example:** Let \( A_1 \equiv p(f(X), Y, Z) \) and \( A_2 \equiv p(A, B, Z) \) be atoms. Then the mgu \( \sigma \) of \( A_1 \) and \( A_2 \) is \( \{ A/f(X), Y/B \} \).

**Unification Algorithm**

Definition: *Disagreement set* \( D \). Locate the leftmost symbol position at which expressions “disagree”. Obtain the set \( D \) of corresponding sub-expressions. The following two examples show the disagreement sets \( D_1 \) and \( D_2 \) of the corresponding sets of atoms \( Q_1 \) and \( Q_2 \).

\[
Q_1 = \{ g(A, l(Y)), g(A, X), g(A, l(W)) \} D_1 = \{ l(Y), X, l(W) \}
Q_2 = \{ g(A, l(Y)), g(Z, A), g(A, l(W)) \} D_2 = \{ A, Z \}
\] (2.1)

The algorithm to determine the mgu of a set of atoms \( Q \) consists of the following steps:

1. Put \( k = 0 \) and \( \sigma_0 = \{ \} \) identity substitution.
2. IF \( \sigma_k(Q) \) contains only one atom
   THEN stop; \( \sigma_k \) is the mgu for \( Q \).
   ELSE find \( D_k \) the disagreement set of \( \sigma_k(Q) \)
3. IF there exists a variable \( v \) and a term \( t \) in \( D_k \) such that \( v \) does not occur in \( t \) (occurs check)
   THEN form \( \sigma_{k+1} = \sigma_k \circ v/t \); increment \( k \) and goto step 2.
   ELSE stop and report that \( Q \) is not unifiable.
2.1. TOP-DOWN EVALUATION

The unification algorithm is non-deterministic (there may be several choices for \{v/t\} at step 3) but always terminates since there are only a finite number of variables in \(Q\) and one variable is eliminated at each step. The most basic example for a failing occurs check is the unification of the two atoms \(g(X)\) and \(g(f(X))\). The unification fails because the variable \(X\) occurs in the term \(f(X)\).

**Example:** \(Q = \{p(a,x,f(y)), p(u,v,w), p(a,r,f(c))\}\)

i) \(\sigma_0 = \{\}\)

ii) \(D_0 = \{a,u\}, \sigma_1 = \{u/a\}; \sigma_1(Q) = \{p(a,x,f(y)), p(a,v,w), p(a,r,f(c))\}\)

iii) \(D_1 = \{x,v,r\}, \sigma_2 = \{u/a, x/v\}; \sigma_2(Q) = \{p(a,v, f(y)), p(a,v,w), p(a,r,f(c))\}\)

iv) \(D_2 = \{v,r\}, \sigma_3 = \{u/a, x/r, v/r\}; \sigma_3(Q) = \{p(a,r, f(y)), p(a,r,w), p(a,r,f(c))\}\)

v) \(D_3 = \{f(y), w, f(c)\}, \sigma_4 = \{u/a, x/r, v/r, w/f(c), y/c\}; \sigma_4(Q) = \{p(a,r, f(y)), p(a,r,f(c))\}\)

vi) \(D_4 = \{y,c\}, \sigma_5 = \{u/a, x/r, v/r, w/f(c), y/c\}; \sigma_5(Q) = \{p(a,r, f(c))\}\)

vii) stop; \(mgu = \{u/a, x/r, v/r, w/f(c), y/c\}\)

2.1.3 Basic Top-Down Algorithm

In contrast to bottom-up evaluation, the starting point of the SLD resolution is the query. With the query given, a literal is selected and a matching (unifiable) clause in the program knowledge base is looked up. If a matching clause is found, the substitution of this step is applied to all literals of the query. The selected literal can then be replaced or removed. Those steps are repeated until there are no matching clauses left, or the empty clause was derived.

The basic concept stays the same, no matter which Datalog variant is evaluated. The basic algorithm can be described in the following five steps. In section 2.3 and 2.4 the different nuances of each evaluation strategy will be pointed out.

**Literal selection** Selects a literal from the query which will be processed.

There exist multiple ways to select a literal. The most common variant is to select to first literal in the query, from the left. For more details about literal selection see section 2.1.1.

**Scan knowledge base** If a literal was selected, the knowledge base is searched for matching facts and/or rules for this literal.

**Unification** A clause matches the literal if the clause and the literal are unifiable (see section 2.1.2 for details on unification). The unification process creates variable mappings. These variable mappings are stored and deliver the query result (which is, in fact, a relation) when the resolution algorithm terminates.
**Remove/replace literal** If the selected literal matches a fact, remove the literal from the query. If the selected literal matches a rule, replace the literal with the rule body.

**Substitution** If the query is not empty by now, apply variable substitution by the variable mappings from unification.

After the substitution the literal selection is started again, until the empty clause is derived or no more matching clauses are found.

### 2.1.4 Graphical Representation

The resolution steps can be represented graphically as a tree, the *proof tree*. The initial query forms the root of the tree. All the other nodes of the *proof tree* are sub-goals, which again are queries. Each sub-goal can have a variable mapping which is written beside the line that connects the related queries. The empty clause is indicated as □ and is called *success node*. All other leafs are called *failure nodes*.

#### Example

For example, the proof tree of the following Datalog program is shown in figure 2.1.

1. `geek(?X) :- not - athletic(?X).`
2. `computer - geek(?X) :- geek(?X), loves - computers(?X).`
3. `not - athletic('Iain').`
4. `loves - computers('Iain').`
5. `?- computer - geek('Iain').`

---

![Sample proof tree (SLD tree): Is Iain a computer geek?](image)

Another, more complex example of a *SLD tree* is shown in figure 2.2. Since the following Datalog program describes multiple rules for the literal *pro-gamer(?X)*, the resulting *SLD tree* consists of multiple branches.
2.2 SLD RESOLUTION

The most basic top-down evaluation strategy is the SLD resolution. SLD stands for linear resolution with selection function for definite clauses. SLD resolution is only complete if the Datalog program contains no circular dependencies and no negation.

In this example every single branch of the two branches leads to a success node, which results in two correct relations for the initial query `pro-gamer(?X)`. ('Manuel') and ('Iain').

Table 2.1 gives an overview of the following evaluation strategies and their capabilities.

2.2 SLD Resolution

The most basic top-down evaluation strategy is the SLD resolution. SLD stands for linear resolution with selection function for definite clauses. SLD resolution is only complete if the Datalog program contains no circular dependencies and no negation.

\begin{verbatim}
geek(?X) :- not-athletic(?X).
computer-geek(?X) :- geek(?X), loves-computers(?X).
pro-gamer(?X) :- computer-geek(?X), playing-game(?X, ?Y).
not-athletic('Iain').
loves-computers('Iain').
playing('Iain', 'Poker').
earning-money('Manuel', 'Warcraft 3').
playing('Manuel', 'Warcraft 3').
game('Warcraft 3').
game('Poker').
?- pro-gamer(?X).
\end{verbatim}

Figure 2.2: Sample SLD tree: Who is a pro-gamer?

Table 2.1 gives an overview of the following evaluation strategies and their capabilities.
Table 2.1: Top-down evaluation strategies

<table>
<thead>
<tr>
<th></th>
<th>SLD</th>
<th>SLDNF</th>
<th>OLDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Always terminates recursions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

2.2.1 Definition: SLD tree

Let \( D \) be a normal Datalog program, \( G \) a query and \( R \) a literal selection rule. A **SLD tree** of \( D \) and \( G \) via \( R \) is a tree with the following properties [1]:

1. Every node is labeled with a query
2. The root is \( G \)
3. Let \( G' \equiv ← L_1 ∧ \ldots ∧ L_i ∧ \ldots ∧ L_n \) with \( n > 0 \) be a query (sub-goal of \( G \)) and \( L_i \) the selected literal of \( G' \) via \( R \).
   - For every clause \( (A ← W) ∈ D \) where a unification \( φ = \text{mgu}(Aσ,L_i) \) exists, there is a sub-goal of \( G' \), namely
     \[
     ← (L_1 ∧ \ldots ∧ L_{i-1} ∧ Wσ ∧ L_{i+1} ∧ \ldots ∧ L_n)φ
     \]
     where \( σ \) is the variable renaming which renames all variables in \( A ← W \) with new ones\(^3\). \( φ \) is the **substitution** used in this step.
   - if no such clause exists \( G' \) is a leaf and called **failure node**
4. If \( G' \equiv ← L_1 ∧ \ldots ∧ L_n \) with \( n > 0 \) can’t be executed, \( G' \) is a leaf.
5. The empty clause is always a leaf and is called **success node**.

The **answer of a query** \( Q \) is a relation \( R = \{t_1, \ldots, t_n\} \), where every tuple \( t_i \) represents a path from the root of the proof tree \( Q \) to a success node. A tuple \( t_i \) is computed by applying the composition of every substitution used in the respecting path to the variables of \( Q \). If one or more success nodes exist, \( Q \) succeeds. If no success node exists \( R \) is empty and \( Q \) fails.

2.2.2 Example: SLD Resolution

The example program shown here is very simple. Nevertheless, it shows all important steps of the SLD resolution algorithm. The program is as follows:

\(^3\)Variable renaming assures that the set of variables of the input clause ← \( L_i \) and the set of variables of the output clause \( A ← W \) do not intersect.
2.2. SLD RESOLUTION

The literal selection rule used in this example is the standard literal selection rule. The proof tree is expanded breadth-first.

Initialization The SLD resolution starts by adding the query of this program - which is \( \text{?-} w(?Y, ?X). \) to the proof tree. The query is hereby the root of the SLD tree.

\[ ?- w(?Y, ?X) \]

Iteration 1 Now a literal is selected. Since there is only one literal in the query, \( w(?Y, ?X) \) is selected (marked blue). After that, the program clauses are looked up for potential matches. The only clause that matches the selected literal is the rule \( w(?X, ?Y) :- s(?X), p(?Y). \) Since \( w(?Y, ?X) \) is unifiable with \( w(?X, ?Y) \), the selected literal is replaced with the rule body and the substitution \([?X/?X1, ?Y/?Y1]\) is saved.

\[ ?- w(?Y, ?X) \]
\[ \begin{array}{c}
\text{?-} s(?X1), p(?Y1) \\
\end{array} \]
Iteration 2 Again, a literal is selected. Since the standard literal selection rule always selects the most left hand literal, \( s(?X1) \) is selected. Matching clauses for \( s(?X1) \) are \( s(1) \) and \( s(2) \). Since the matching clauses are facts, the selected literal can be removed. The substitution is stored again. Note that for easier understanding, the tree is expanded breadth-first in this example. The actual implementation does a depth-first search.

\[
\neg w(Y, X)
\]

\[
\neg w(Y, X) \\
\{ \gamma / \gamma_1 \ \gamma / \gamma_1 \}
\]

\[
\neg s(?X1), p(?Y1) \\
\{ ?X1 / 1 \} \quad \{ ?X1 / 2 \}
\]

\[
\neg p(?Y1) \quad \neg p(?Y1)
\]

Iteration 3 With a single identical literal left in both branches, \( \neg p(?Y1) \) is selected. Matching clauses are again facts, namely \( p(3) \) and \( p(4) \). By substituting the literals with the matching clauses the empty clause is derived.

\[
\neg w(Y, X)
\]

\[
\neg w(Y, X) \\
\{ \gamma / \gamma_1 \ \gamma / \gamma_1 \}
\]

\[
\neg s(?X1), p(?Y1) \\
\{ ?X1 / 1 \} \quad \{ ?X1 / 2 \}
\]

\[
\neg p(?Y1) \quad \neg p(?Y1)
\]

\[
\{ ?Y1 / 3 \} \quad \{ ?Y1 / 4 \} \quad \{ ?Y1 / 4 \} \quad \{ ?Y1 / 3 \}
\]

\[
\emptyset \quad \emptyset \quad \emptyset \quad \emptyset
\]

To get the answer tuples of the query, the stored variable mappings are substituted with the query variables. This substitution leads to the correct answer of the query: \((1, 3), (1, 4), (2, 3)\) and \((2, 4)\) (see page 21).

As stated before, the SLD resolution is limited to positive Datalog programs. Further, it may gets stuck in an infinite loop when the program contains recur-
sive rules. Depending on the literal selection, the program may terminate or may not. To eliminate the restriction to positive Datalog programs, the SLD resolution can be extended with negation as failure.

## 2.3 SLDNF Resolution

The SLDNF resolution extends the SLD resolution with *negation as failure*. Negation as failure, or *NAF*, is used to evaluate negative literals, i.e. literals that contain negated atoms. NAF simply derives \( \neg A \) from failure to derive \( A \). If \( A \) fails, \( \neg A \) succeeds. If \( A \) succeeds, \( \neg A \) fails.

The basic changes which form the extension of SLD begin with the literal selection. If the selected literal is negative (\( \neg A \)), a new SLDNF evaluation with only the positive atom (\( A \)) as query is created. If the evaluation fails with any substitution \( \sigma \), \( \neg A \) is successful and the current sub-goal is substituted with the identical substitution \( \epsilon \). If the evaluation is successful, \( \neg A \) fails. In this case the current sub-goal is a failure node.

Note that NAF be only used for verification, never to generate new nodes. NAF is used to proof whether a negative ground literal (a literal is ground when it contains no variables) is true or false.

### 2.3.1 Definition: SLDNF tree

Let \( D \) be a normal Datalog program, \( G \) a query and \( R \) a literal selection rule. A *SLDNF tree* of \( D \) and \( G \) via \( R \) is a tree with the following properties [1]:

1. Every node is labeled with a query
2. The root is \( G \)
3. Let \( G' \equiv L_1 \land \ldots \land L_i \land \ldots \land L_n \) with \( n > 0 \) be a query (sub-goal of \( G \)) and \( L_i \) the selected literal of \( G' \) via \( R \).

   (a) \( L_i \) is a positive literal
       For every clause \( (A \leftarrow W) \in D \) where a unification \( \phi = mgu(A \sigma, L_i) \) exists, there is a successor node of \( G' \), namely
\[ \left( L_1 \land \ldots \land L_{i-1} \land W \sigma \land L_{i+1} \land \ldots \land L_n \right) \phi \]

where \( \sigma \) is the variable renaming which renames all variables in \( A \leftarrow W^4 \) and \( \phi \) is the substitution used in this step.

\( G' \) is a leaf and called failure node if no such clause exists.

(b) \( L_i \equiv \lnot A \) is a negative ground literal

Node \( G' \) has a successor node

\[ \leftarrow L_1 \land \ldots \land L_{i-1} \land L_{i+1} \land \ldots \land L_m \]

if there exists a finite SLDNF failure tree for \( D \) and \( \leftarrow A \) via any literal selection rule. The substitution used in this step is the identical substitution \( \epsilon \).

\( G' \) is a leaf and is called a failure node, if there exists a SLDNF success tree for \( D \) and \( \leftarrow A \) (see below).

4. If \( G' \equiv \leftarrow L_1 \land \ldots \land L_n \) with \( n > 0 \) can’t be executed, \( G' \) is a leaf.

5. The empty clause is always a leaf and is called success node.

If all leaves of a finite SLDNF tree are failure nodes, the tree is called finite SLDNF failure tree. All SLDNF trees with one or more success nodes are called SLDNF success trees.

### 2.3.2 Example: SLDNF Resolution

Consider the following illustrative example:

```prolog
1 r(1, 2).
2 r(2, 3).
3 r(3, 4).
4 r(4, 5).
6 ?- p(?X, ?Y).
```

The literal selection rule used in this example is the standard literal selection rule. The SLDNF tree is traversed breadth-first.

**Initialization** The initial query contains no negated literals and thus is equivalent to the initialization of the SLD evaluation. The query \(?- p(?X, ?Y)\) forms the root of the proof tree.

\(?- p(?X, ?Y)\)

---

\(^4\)see SLD algorithm in chapter 2.2
2.3. **SLDNF RESOLUTION**

**Iteration 1** As in the previous example, the current node consists of only one literal, so this literal is selected. The only matching clause is the rule \( p(?X, ?Y) :- r(?X, ?Y), ?X < 2, \text{not } p(2, ?Y) \). The rule head \( p(?X, ?Y) \) is equal to the selected literal. The two literals are hereby unifiable and no substitution is needed.

\[
\neg p(?X, ?Y) \\
\downarrow
\neg r(?X, ?Y), ?X < 2, \text{not } p(2, ?Y)
\]

**Iteration 2** The standard literal selection rule selects \( r(?X, ?Y) \) as the next literal. For every matching clause, the substitution is saved and the current node is expanded by substituting the variable mappings and removing the selected literal.

\[
\neg p(?X, ?Y) \\
\downarrow
\neg r(?X, ?Y), ?X < 2, \text{not } p(2, ?Y)
\]

\[
\begin{align*}
\neg 1 &< 2, \text{not } p(2, 2) \\
\neg 2 &< 2, \text{not } p(2, 3) \\
\neg 3 &< 2, \text{not } p(2, 4) \\
\neg 4 &< 2, \text{not } p(2, 5)
\end{align*}
\]

**Iteration 3** The selected literals branch by branch are \( 1<2, 2<2, 3<2 \) and \( 4<2 \) respectively. The only literal that evaluates to true, is \( 1<2 \). Consequently, this is the only branch that will be expanded (node \( \neg 1<2, \text{not } p(2,2) \)).

**Iteration 4 (NAF)** Now the selected literal is \( \text{not } p(2,2) \), which is a *negative ground literal*. This starts the *negation as failure* procedure. A new SLDNF resolution is started, with the new query \( p(2,2) \). The selected literal \( p(2,2) \) matches the rule for \( p \). The node is expanded to \( \neg r(2,2), 2<2, \text{not } p(2,2) \). Again, a literal is selected (\( r(2,2) \)). Since the selected literal does not match any clause, the node can not be expanded any further. The SLDNF sub-proof for \( p(2,2) \) has terminated.
CHAPTER 2. SOLUTION

Iteration 5 As we have seen the SLDNF resolution for \( p(2,2) \) fails. From this it follows that \( \neg p(2,2) \) succeeds. Since \( \neg p(2,2) \) was the last executable literal, the SLDNF resolution for \( ?- p(?X,?Y) \) terminates (see page 25). The resulting tuple is \((1,2)\) and is again calculated by remapping the query variables (see SLD example).

A small change to the above example points out the importance of the literal order and selection. If the literals of the rule are reordered, e.g. from

\[
p(?X,?Y) :- r(?X,?Y), ?X < 2, \neg p(2,?Y).
\]

to

\[
p(?X,?Y) :- r(?X,?Y), \neg p(2,?Y), ?X < 2.
\]
the SLDNF resolution would not terminate anymore using the standard literal selection rule. Before the more restrictive literal \(?X < 2\) could be evaluated, the SLDNF resolution would end up in an infinite loop, trying to prove \(-p(2, 2)\). The same effect could appear by keeping the original literal ordering but using a different literal selection rule.

To avoid this non-terminating behaviour, a top-down resolution strategy has to remember which nodes are already being calculated or have been calculated previously. The OLDT resolution presented in the next section seizes this approach.

### 2.4 OLDT Resolution

The acronym OLDT stands for Ordered selection strategy with linear resolution for definite clauses with tabulation [1]. Whereas SLD resolution allows different literal selection rules, OLD only allows the standard literal selection rule. This resolution is an extension to the already presented SLD resolution. Like the SLD resolution, it can be extended with negation as failure (OLDTNF). Since the NAF extension for OLDT is very similar to the one for SLD - and for simplicity reasons - the following section will present the pure OLDT resolution and focus on tabulation.

Tabulation, or tabling, remembers which predicates have already been computed. To avoid saving all processed predicates, only so-called memo predicates are saved.
2.4.1 Memo Predicates

Before the actual resolution starts, some predicates are tagged as memo predicates. Tagging of recursive defined predicates is precondition for termination and completeness of the OLDT resolution. Beneath recursive defined predicates, other predicates may be tagged for performance reasons (i.e. if a built-in literal takes a long time for evaluation, e.g. the lookup of an URL). If no predicates are tagged, the OLDT will behave like the OLD resolution [1].

The OLDT examples in this thesis will only tag recursive defined predicates. Currently, the standard predicate tagger implemented in IRIS tags recursive defined predicates only.

Predicate Tagging Example

Consider the following rules:

\[ 1\ s(?X, ?Y) :- r(?X, ?Y). \]
\[ 2\ s(?X, ?Y) :- s(?X, ?Z), r(?Z, ?Y). \]

Since the only interesting part for predicate tagging is the predicate symbol, the first rule can be read as \( s \rightarrow r \). \( r \) does not lead anywhere else, so \( s \) is not tagged because of this rule.

The second rule can be read as \( s \rightarrow s, r \). Since \( s \) is defined recursively here, \( s \) is tagged as a memo predicate.

2.4.2 Definition: OLDT structure

An OLDT structure consists of a OLD-Tree and two tables: the memo table and the link table.

1. A node \( \leftarrow A_1 \wedge \ldots \wedge A_n \) in the OLD tree is called memo node, if the predicate symbol from \( A_1 \) is a memo predicate. A memo node is either a result- or a link node.

2. For every result node \( \leftarrow A_1 \wedge \ldots \wedge A_n \) the memo table is updated. Every result is appended to the answer list of \( A_1 \).

3. For every link node \( \leftarrow A_1 \wedge \ldots \wedge A_n \) the link table is updated. For every link node a pair consisting of the link node and a pointer (which is pointing to the answer list of an atom \( A \) which is unifiable with \( A_1 \)) is added to the link table. [1]

Every memo node of a OLDT structure is classified as a result node or a link node before expansion. Dependent on this classification, the corresponding table is updated.
2.4. OLDT RESOLUTION

Table registration

Let $T$ be an OLDT structure and $G \equiv A_1 \land \ldots \land A_n$ a memo node in $T$. By applying one of the following steps we get a new OLDT structure $T'$.

1. $G$ is a link node. Consequently, $A_1$ is an instance of an entry $A$ in the memo table. By adding a new (node, pointer) pair to the link table, we get $T'$. The pointer in the newly added pair points to the beginning of the answer list of $A$.

2. $G$ is a result node. That means that $A_1$ is not an instance of an entry $A$ in the memo table. By adding $A_1$ with an empty answer list to the memo table, we get $T'$. [1]

Let $D$ be a Datalog program and $Q$ a query. The initial table structure consists of one entry $(A_1, [\,])$ in the memo table (where $[\,]$ is the empty answer list and $A_1$ is the first atom of $Q$) and the empty link table.

With the initial table structure given, the actual OLDT resolution can be started. Each resolution step extends the OLDT structure by either the OLD extension or the link extension.

OLD Extension

Choose a node $G' \equiv A_1 \land \ldots \land A_n (n \geq 1)$ from $T$ which has no successor node and is not a link node.

1. For every clause $A \leftarrow W$ add a child node

$$\leftarrow (W\sigma \land A_2 \land \ldots \land A_n)\text{mgu}(A\sigma, A_1)$$  \hspace{1cm} (I)

of $G'$ to the OLD tree, where $\sigma$ replaces all variables in $A \leftarrow W$ with unused ones$^5$.

2. Do the table registration for every new node

3. For every new node that leads to a success node, add a answer to the appropriate answer list. [1]

Link Extension

Choose a link node $G' \equiv A_1 \land \ldots \land A_n$. The answer list where the pointer of the link node points to must not be empty. Let $A$ be the atom which the pointer refers to. Bend the pointer to the next element of the answer list.

1. If $A$ and $A_1$ are unifiable, add

$$\leftarrow (A_2 \land \ldots \land A_n)\text{mgu}(A, A_1)$$  \hspace{1cm} (II)

as a child node of $G'$.

$^5$see SLD algorithm in chapter 2.2
2. For every new node that leads to a success node, add a answer to the appropriate answer list. [1]

The OLDT resolution terminates when no node $G'$ can be selected for neither the OLD nor the link extension. The result tuples are resolved like in the SLD resolution.

### 2.4.3 Example: OLDT Resolution

Consider the following example, where the predicate `substitutes` is defined recursively. The facts of the program say that 1 and 2 are exchangeable, which leads to a cyclic graph.

```prolog
3 replaces(1, 2).
4 replaces(1, 3).
5 replaces(2, 1).
6 replaces(2, 4).
```

An example query for the above program would be

```prolog
?- substitutes(1, ?Y).
```

which says *'Give me all parts that can be replaced with 1.'* For space reasons abbreviations are used instead of the full predicates: $s$ for `substitutes` and $r$ for `replaces`.

**Memo Predicates** Before the OLDT resolution gets started, recursively defined predicates are tagged as memo predicates. Since there is only one recursively defined predicate, $s(?X, ?Y)$ is the only memo predicate.

**Initialization** The first atom $s(1, ?Y)$ of the query is stored in the memo table. With this atom, the (currently empty) answer list is stored. The initial query forms the root of the OLD tree.

```
?- s(1, ?Y)

Memo table:
s(1, ?Y): []
Link table:
```

**Loop detection** By resolution with the two clauses for `substitutes` two child nodes for $s(1, ?Y)$ are added to the OLD tree. The left branch is evaluated first. The first atom of the left branch node is $s(1, Z)$, which is an instance of the atom in the memo table $s(1, Y)$. From this it follows that this atom is already being computed somewhere else in the OLD tree. This node is not expanded, but an entry to the link table is made. The entry contains the node and a pointer to the beginning of the answer list.
More answers As the answer list was filled in the previous step, the paused
expansion of the left branch can be continued now. The first element
in the answer list of \( s(1, ?Y) \) substitutes the appropriate variable. In
the case of the node \( ?- s(1, ?Z), r(?Z, ?Y) \) and the stored atom \( s(1, ?Y) \)
the appropriate variable is \( ?Z \). By expanding the node with \( ?Z/2 \), a new
answer for \( s(1, ?Y) \) is derived. The new answer \( (1) \) is appended to the
answer list.
The node \(?-s(1, ?Z), r(?Z, ?Y)\) is expanded for every answer tuple in the answer list of \(s(1, ?Y)\). The new answers are added to the answer list. Note that the answer list contains unique tuples, so recomputed answers won't be added to the list. By resolving the node \(?-s(1, ?Z), r(?Z, ?Y)\) with the last answer (4) the expansion of the node is finished. No further answers are added to the answer list. The OLD tree is complete (see page 31).

In the example above, the OLD extension is favored over the link extension. This has to be the case to guarantee the completeness of the OLDT resolution.
2.4. OLDT RESOLUTION

Figure 2.5: Complete OLD tree and tables for the query \(?- s(1, ?Y)\).

If the link extension is preferred, the OLDT resolution loses its completeness for definite programs, as the following example shows [1]:

1. \(p(?X) :- q(?X), r.\)
2. \(q(f(?X)) :- q(?X).\)
3. \(q('a').\)
4. \(r.\)

Let \(?- q(?X)\) be the query and \(q, p\) the only predicates that are tagged as memo predicates. If the link extension is favored over the OLD extension, the resulting OLD tree gets infinitely broad by expanding the node \(?- q(?X1), r\).
Chapter 3

Conclusions

The last chapter points out the most prominent differences between bottom-up and top-down evaluation as well as some issues that occurred during research and implementation.

3.1 Bottom-up versus top-down

In the following section strengths and weaknesses of each resolution strategy are presented. Figure ?? outlines the most prominent difference between top-down and bottom-up evaluation strategies. A top-down approach does not necessarily derive the complete model.

**Bottom-up** Two bottom-up strategies have been introduced in this thesis: the naive and the semi-naive evaluation. These strategies are pure bottom-up strategies. Other strategies like “Magic Sets” are bottom-up strategies with a top-down component.

+ Once the model is built, further queries can be evaluated quickly, since rebuilding the model is only necessary if the knowledge base changes.
+ Always terminates if there exists a finite minimal model.
− All possible facts are derived, even if the query is very simple.
− New tuples have to be saved temporary
− Before a tuple is added to the relation it has to be checked for uniqueness. This is a rather expensive operation.

**Top-down** SLD and SLDNF are pure top-down evaluation strategies. Since the OLDT resolution saves intermediate results (bottom-up component) it is a mixed strategy even if the top-down component is the dominant element. The following list relates to pure top-down strategies.

+ Focus on the query which leads to a goal-oriented evaluation.
3.2 Implementation Issues

The first sticking point was getting to know IRIS. I had no previous experience with reasoning, nor with this particular reasoner. Understanding the inner structure of IRIS and the logical background of reasoning, Datalog, and especially the bottom-up evaluation strategies took several months. After understanding the mechanics I was able to begin with the actual implementation, which lead to the next problem.

Since IRIS uses bottom-up strategies for evaluation, namely the naive and the semi-naive approach, most of the code used for evaluation was impractical for the top-down evaluation strategies. This lead to writing a lot of code in form
3.3. **FINAL WORDS**

Another issue was testing and benchmarking the new evaluation strategies. As we have seen in section 3.1, the most expensive operation using bottom-up evaluation strategies is building the model. When the model is calculated, running queries against it is very fast. On the other hand, top-down evaluation strategies construct a proof tree for every query. This fact makes it very difficult to evaluate the speed of the strategies, since the results differ greatly depending on how the program looks.

### 3.3 Final Words

The extension of IRIS by the three top-down evaluation strategies *SLD, SLDNF* and *OLDT* features a contrast to the bottom-up evaluation strategies. The most interesting strategy of this three is the *OLDT* evaluation strategy, since it eliminates the shortcomings of SLD and SLDNF. Depending on the evaluated program, the new strategies may show a significant speedup. This leads to the improvement of IRIS and programs that make use of this reasoner.
Bibliography


