I4-D18

Chaining with Memory in Xcerpt

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D18/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Thomas Eiter and Jose Alferes
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	29 February 2008
Actual submission date:	10 March 2008

Abstract

Moving from single-rule Xcerpt programs as described in previous deliverables to full Xcerpt programs requires to address the issue of efficient rule chaining. In this deliverable, we first survey existing approaches for efficient rule chaining (using some form of memoization) in logic programming and then briefly outline first results and challenges when extending these results to Xcerpt.

Keyword List

chaining, recursion, logic programming, resolution, OLDT, SLG, magic set, Xcerpt, subsumption

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Chaining with Memory in Xcerpt

Benedikt Linse[1], **Norbert Eisinger**[1], **Clemens Ley**[2], **Tim Furche**[1], **François Bry**[1]

[1] Institute for Informatics, University of Munich, Germany
http://pms.ifi.lmu.de/

[2] Oxford University Computing Laboratory, England
http://web.comlab.ox.ac.uk/oucl/

10 March 2008

Abstract
Moving from single-rule Xcerpt programs as described in previous deliverables to full Xcerpt programs requires to address the issue of efficient rule chaining. In this deliverable, we first survey existing approaches for efficient rule chaining (using some form of memoization) in logic programming and then briefly outline first results and challenges when extending these results to Xcerpt.

Keyword List
chaining, recursion, logic programming, resolution, OLDT, SLG, magic set, Xcerpt, subsumption

Contents

1 Efficient Chaining in Logic Programming: A Survey **3**
- 1.1 Positive Rule Sets . . . 4
  - 1.1.1 Semi-naive Evaluation of Datalog Programs . . . 4
  - 1.1.2 The Magic Templates Transformation Algorithm . . . 6
    - 1.1.2.1 Adornment of Datalog Programs. . . . 6
    - 1.1.2.2 Goal-Directed Rewriting of the Adorned Program. . . . 8
  - 1.1.3 The Rete Algorithm . . . 9
  - 1.1.4 Basic Backward Chaining: SLD-Resolution . . . 10
  - 1.1.5 Backward Chaining with Memorization: OLDT-Resolution . . . 12
  - 1.1.6 The Backward Fixpoint Procedure . . . 14
- 1.2 Rule Sets with Non-monotonic Negation . . . 21
  - 1.2.1 Computation of the Iterative Fixpoint Semantics for Stratified Logic Programs with Negation as Failure . . . 22
  - 1.2.2 Magic Set Transformation for Stratified Logic Programs . . . 22
  - 1.2.3 Computation of the Stable Model Semantics . . . 24
  - 1.2.4 A More Efficient Implementation of the Stable Model Semantics . . . 25
  - 1.2.5 Computation of the Well-Founded Model Semantics . . . 26
  - 1.2.6 Computing the Well-Founded Semantics by the Alternating Fixpoint Procedure 28
  - 1.2.7 Other Methods for Query Answering for Logic Programs with Negation . . . 29

2 Chaining and Memoization in Xcerpt: An Outlook **31**
- 2.1 Outline . . . 32
- 2.2 Introduction . . . 32
- 2.3 Ad-hoc-search Use Cases . . . 34
- 2.4 Related work . . . 35
- 2.5 Answering the SSSPP for Foaf Profiles with Xcerpt^{RDF} . . . 36
  - 2.5.1 Crawling FOAF Documents to Answer SSSPP Problems . . . 36
  - 2.5.2 Formulation of SSSPP as an Unstratified Logic Program . . . 38
  - 2.5.3 Efficient Evaluation of Xcerpt^{RDF} Multi-Rule Programs . . . 41
- 2.6 Decidability of Xcerpt Query Term Subsumption . . . 43
- 2.7 Conclusion . . . 48

Overview of this Deliverable

This deliverable is comprised of two chapters: the first presents a survey (derived from [9]) over existing techniques for efficient chaining of logic programs. This includes the magic set transformation (for implementing a backward chaining, goal-driven evaluation in a system with a fixpoint operator), the rete algorithm employed extensively in production rule systems, several advanced forms of resolution (SLD, OLDT) and a discussion how these techniques apply to rule sets with negation.

The second chapter focuses on Xcerpt and the challenges when applying these techniques in this context. Though Xcerpt shares many characteristics with basic logic programming languages where chaining is concerned, its novel form of *incomplete* terms pose a previously not considered challenge for subsumption and memoization of intermediary results that lies at the heart of all the techniques discussed above.

The efficient chaining in Xcerpt is a topic of ongoing research under the lead of Benedikt Linse. The current prototype contains mostly naive (fixpoint) operators for rule chaining. We plan to extent it with the techniques developed in the further course of this study.

Chapter 1

Efficient Chaining in Logic Programming: A Survey

1.1 Positive Rule Sets

1.1.1 Semi-naive Evaluation of Datalog Programs

The fixpoint semantics of a positive logic program P directly yields an operational semantics based on canonical forward chaining of the immediate consequence operator $\mathbf{T}_P$ introduced in Section **??** (Definition **??**) until the least fixpoint is reached.

Let us quickly recapitulate the definition of the fixpoint of a datalog program P given the example program in Listing 1.1_1.

```
feeds_milk(betty).
lays_eggs(betty).
has_spines(betty).

monotreme(X) ← lays_eggs(X), feeds_milk(X).
echidna(X) ← monotreme(X), has_spines(X).
```

Listing 1.1_1: An example program for fixpoint calculation

The *intensional* predicate symbols of a datalog program P are all those predicate symbols that appear within the head of a rule, as opposed to the *extensional* predicate symbols which appear only in the bodies of rules of a program. With this definition `feeds_milk`, `lays_eggs` and `has_spines` are extensional predicate symbols, whereas `monotreme` and `echidna` are intensional predicate symbols. The set of all extensional and intensional predicate symbols of a datalog program P (denoted $ext(P)$ and $int(P)$ respectively) is called the schema of P. An *instance* over a schema of a logic program is a set of sets of tuples $s_1, \ldots, s_k$, where each set of tuples $s_i, 1 \leq i \leq k$ is associated with a predicate symbol p in the schema and s_i is the extension of p. The set of base facts of the program 1.1_1 corresponds to an instance over the extensional predicate symbols, where the set $\{betty\}$ is the set associated with each of the symbols `feeds_milk`, `lays_eggs` and `has_spines`.

Based on these definitions the semantics of a logic program P is defined as a mapping from extensions over $ext(P)$ to extensions over $int(P)$. There are several possibilities to define this function. The

fixpoint semantics uses the *immediate consequence operator* $\mathbf{T}_P$ for this aim.

Given a datalog program P and an instance I over its schema $sch(P)$, an atom A is an *immediate consequence* of P and I if it is either already contained in I or if there is a rule $A \leftarrow cond_1, \ldots, cond_n$ in P where $cond_i \in I \ \forall 1 \leq i \leq n$. The immediate consequence operator $\mathbf{T}_P(I)$ maps an instance over the schema $sch(P)$ to the set of immediate consequences of P and I.

A fixpoint over the operator $\mathbf{T}_P$ is defined as an instance I such that $\mathbf{T}_P(I) = I$. It turns out that any fixpoint for a datalog program is a model of the (conjunction of clauses of the) program. Furthermore, the model-theoretic semantics $P(I)$ of a logic program P on an input instance I, which is defined as the minimum model of P that also contains I, is the minimum fixpoint of $\mathbf{T}_P$.

As mentioned above, this fixpoint semantics for datalog programs, which may be extended to non-datalog rules[26], gives directly rise to a constructive algorithm to compute the minimum model of a program.

Consider again Listing 1.1_1. The set of immediate consequences of this program with the initial instance[1] $I_0 = \{\{\}_{f\_m}, \{\}_{l\_e}, \{\}_{h\_s}, \{\}_{mon}, \{\}_{ech}\}$ is $I_1 := \mathbf{T}_P(I_0) = \{\{betty\}_{f\_m}, \{betty\}_{l\_e}, \{betty\}_{h\_s}, \{\}_{mon}, \{\}_{ech}\}$. The second application of the fixpoint operator yields $I_2 := \mathbf{T}_P^2(I_0) = \{\{betty\}_{f\_m}, \{betty\}_{l\_e}, \{betty\}_{h\_s}, \{betty\}_{mon}, \{\}_{ech}\}$.

I_3 is defined analogously and the extension of `echidna` is set to $\{betty\}$. Finally the application of $\mathbf{T}_P$ to I_3 does not yield any additional facts such that the condition $I_3 = I_4$ is fulfilled, and the fixpoint is reached.

The above procedure can be implemented with the pseudo-algorithm in Listing 1.1_2, which is called *naive evaluation* of datalog programs, because for the computation of I_i all elements of I_{i-1} are recomputed. As suggested by its name, the function `ground_facts` returns all the ground facts of the program which is to be evaluated. The function `instantiations` takes as a first argument a rule R, which may contain variables, and as a second argument the set of facts I_{i-1} which have been derived in the previous iteration. It finds all instantiations of the rule R which can be satisfied with the elements of I_{i-1}.

```
I_0 := ∅
I_1 := ground_facts(P)
i := 1
while I_i ≠ I_{i-1} do
  i := i + 1
  I_i := I_{i-1}
  while (R = Rules.next())
    Insts := instantiations(R, I_{i-1})
    while (inst = Insts.next())
      I_i := I_i ∪ head(inst)
return I_i
```

Listing 1.1_2: Naive evaluation of a datalog program P

The central idea underlying the so-called *semi-naive* evaluation of datalog programs is that all facts that can be newly derived in iteration i must use one of the facts that were newly derived in iteration $i-1$ – otherwise they have already been derived earlier. To be more precise, the rule instantiations that justify the derivation of a new fact in iteration i must have a literal in their rule body which was derived in iteration $i-1$. In order to realize this idea one must keep track of the set of newly derived facts

[1] the predicate symbols in subscript position indicate that the first set is the extension of `feeds_milk`, the second one the one of `lays_eggs`, and so on.

in each iteration. This method is also called *incremental forward chaining* and is specified by Listing 1.1_3. In line 2 the increment `Ink` is initialized with all facts of the datalog program. In line 4 the set `Insts` of instantiations of rules that make use of at least one atom of the increment *Ink* is computed at the aid of the function `instantiations`. The function `instantiations` does not yield ground rules that are justified by the set `KnownFacts` only, such that the call `instantiations({ p(a), q(a)}, { })` for a program consisting of the rule `r(x)← p(X), q(X)` would *not* yield the instantiation $i_1 :=$ `r(a)← p(a), q(a)`. In contrast, i_1 would be returned by the call `instantiations({ p(a)}, { q(a)})` with respect to the same program.

Once these fresh rule instantiations have been determined, the distinction between facts in the increment and older facts is no longer necessary, and the two sets are unified (line 5). The new increment of each iteration is given by the heads of the rule instantiations in `Insts`.

```
KnownFacts := ∅
Ink := { Fact | (Fact ← true) ∈ P }
while (Ink ≠ ∅)
  Insts := instantiations(KnownFacts, Ink)
  KnownFacts := KnownFacts ∪ Ink
  Ink := heads(Insts)
return KnownFacts
```

Listing 1.1_3: Semi-naive evaluation of a datalog program P

Although the semi-naive evaluation of datalog programs avoids a lot of redundant computations that the naive evaluation performs, there are still several ways of optimizing it.

- In the case that besides a program P also a query q is given, it becomes apparent that a lot of computations, which are completely unrelated to q, are carried out. This is a general problem of forward chaining algorithms when compared to backward chaining. However, it is possible to write logic programs that, also when executed in a forward-chaining manner, are in a certain sense goal-directed. In fact it is possible to transform any datalog program P and query q into a logic program P' such that the forward chaining evaluation of P' only performs computations that are necessary for the evaluation of q. In Section 1.1.2 these so-called *magic templates transformations* are presented.

- A second source of inefficiency is that in each iteration i, it is tested from scratch whether the body of a rule is satisfied. It is often the case that a rule body completely satisfied in iteration i was almost completely satisfied in iteration $i-1$, but the information about which facts contributed to the satisfaction of rule premises in iteration $i-1$ must be recomputed in iteration i. It is therefore helpful to store complete and partial instantiations of rules during the entire evaluation of the program.

- Storing partial instantiations of rule bodies gives rise to another optimization if the rules of the program share some of their premises. In this case, the partial rule instantiations are shared among the rules. Both this and the previous optimization are realized by the Rete algorithm, which is introduced in Section 1.1.3.

1.1.2 The Magic Templates Transformation Algorithm

The magic templates algorithm [21] is a method of introducing a goal directed search into a forward chaining program, thereby benefiting both from the termination of forward chaining programs and from

the efficiency of goal directed search. It is important to emphasize that the evaluation of the transformed program is performed in the ordinary forward chaining way or can be combined with the semi-naive algorithm as described above.

The magic templates rewriting transforms a program P and a query q in two steps: a transformation of the program into an adorned version, and a rewriting of the adorned program into a set of rules that can be efficiently evaluated with a bottom up strategy.

1.1.2.1 Adornment of Datalog Programs.

In the first step, the program is rewritten into an *adorned* version according to a *sideways information passing strategy*, often abbreviated sip.

A sideways information passing strategy determines how variable bindings gained from the unification of a rule head with a goal or sub-goal are passed to the body of the rule, and how they are passed from a set of literals in the body to another literal. The ordinary evaluation of a Prolog program implements a special sideways information passing strategy, in which variable bindings are passed from the rule head and all previously occurring literals in the body to the body literal in question. There are, however, many other sips which may be more convenient in the evaluation of a datalog or Prolog program. In this survey, only the standard Prolog sip is considered, and the interested reader is referred to [5] for a more elaborate discussion of sideways information passing strategies.

The construction of an adorned program is exemplified by the transformation of the transitive closure program in Listing 1.1_4 together with the query `t(a,Answer)` into its adorned version in Listing 1.1_5. In order to better distinguish the different occurrences of the predicate `t` in the second rule, they are labeled `t-1`, `t-2` and `t-3`, but they still denote the same predicate.

```
t(X,Y) ← r(X, Y).
t-3(X,Z) ← t-1(X, Y), t-2(Y, Z).

r(a, b).
r(b, c).
r(c, d).
```

Listing 1.1_4: Transitive closure computation

When evaluated in a backward chaining manner, the query Q :=`t(a,Answer)` is first unified with the head of the first rule, generating the binding `X=a` which is passed to the rule body. This sideways information passing can be briefly expressed by `t` $\hookrightarrow_X$ `r`. The query$Q$ is also unified with the head of the second rule, generating once more the binding `X=a`, which would be used to evaluate the literal `t-1(X,Y)` by a Prolog interpreter. In the remaining evaluation of the second rule, the binding for `Y` computed by the evaluation of `t-1(X,Y)` is passed over to the predicate `t-2`. This can be briefly expressed by the sips `t-3` $\hookrightarrow_X$ `t-1` and `t-1` $\hookrightarrow_Y$ `t-2`.

From this information passing strategy an adorned version of Listing 1.1_4 can be derived. Note that all occurrences of the predicate `t` (and its numbered versions) are evaluated with the first argument bound and the second argument free when Q is to be answered. In the magic templates transformation it is important to differentiate between different call-patterns for a predicate. This is where adornments for predicates come into play. An adornment a for a predicate p of arity n is a word consisting of n characters which are either 'b' (for bound) or 'f' (for free). Since the first argument of `t` is always bound in the program and the second argument is always free, the only adornment for `t` is bf. Since the evaluation of literals of *extensional* predicates amounts to simply looking up the appropriate values, adornments are only introduced for *intensional* predicate symbols.

It is interesting to note that the choice of the information passing strategy strongly influences the resulting adorned program. In the case that one chooses to evaluate the literal `t-2` before `t-1`, both arguments of `t-2` would be unbound yielding the sub-query $\texttt{t-2}^{ff}$, and thus an additional adorned version of the second rule would have to be introduced for this sub-query. This additional adorned rule would read $\texttt{t-3}^{ff}\texttt{(X,Z)} \leftarrow \texttt{t-1}^{fb}\texttt{(X, Y), t-2}^{ff}\texttt{(Y, Z)}$.. For the sake of simplicity, the following discussion refers to the shorter version depicted in Listing 1.1_5 only.

```
t^bf(X,Y) ← r(X, Y).
t-3^bf(X,Z) ← t-1^bf(X, Y), t-2^bf(Y, Z).

r(a, b).
r(b, c).
r(c, d).
```

Listing 1.1_5: The adorned version of the program in Listing 1.1_4

1.1.2.2 Goal-Directed Rewriting of the Adorned Program.

Given an adorned datalog program P^{ad} and a query q, the general idea of the magic templates rewriting is to transform P^{ad} into a program P_m^{ad} in a way such that all sub-goals relevant for answering q can be computed from additional rules in P_m^{ad}. Slightly alternated versions of the original rules in P^{ad} are included in P_m^{ad}, the bodies of which ensure that the rule is only fulfilled if the head of the rule belongs to the set of relevant sub-goals.

Hence the magic template transformation generates two kinds of rules: The first set of rules controls the evaluation of the program by computing all relevant sub-goals from the query, and the second set of rules is an adapted version of the original program with additional premises in the bodies of the rules, which ensure that the rules are only evaluated if the result of the evaluation contributes to the answer of q.

The functioning of the magic templates transformation and the evaluation of the transformed program is again exemplified by the transitive closure computation in Listings 1.1_4 and 1.1_5.

1. In a first step, a predicate $\texttt{magic\_p}^a$ of arity $nb(p^a)$ is created for each adorned predicate $\texttt{p}^a$ that occurs in the adorned program, where $nb(p^a)$ denotes the number of bound arguments in p^a – in other words the number of 'b's in the adornment a. Thus for the running example, the predicate $\texttt{magic\_t}^{bf}$ with arity one is introduced. The intuition behind magic predicates is that their extensions during bottom-up evaluation of the program, often referred to as *magic sets*, contain all those sub-goals that need to be computed for p^a. In the transitive closure example, the only initial instance of $\texttt{magic\_t}^{bf}$ is $\texttt{magic\_t}^{bf}\texttt{(a)}$, which is directly derived from the query `t(a,Answer)`. This initial magic term is added as a seed[2] to the transformed program in Listing 1.1_6.

2. In a second step, rules for computing sub-goals are introduced reflecting the sideways information passing within the rules. Let r be a rule of the adorned program P^{ad}, let h^a be the head of r, and $l_1 \ldots l_k$ the literals in the body of r. If there is a query that unifies with the head of the rule, if queries for $l_1 \ldots l_i$ $(i < k)$ have been issued and if they have been successful, the next step in a backward chaining evaluation of P^{ad} would be to pursue the sub-goal l_{i+1}. Thus a control rule

[2] it is called the *seed*, because all other magic terms are directly or indirectly derived from it.

l_{i+1} ← magic_h^a, $l_1 \ldots l_i$ is included in P_m^{ad}. For the running example the rule magic_t^{bf}(Y) ← magic_t^{bf}(X), t(X,Y) is added.

3. In a third step, the original rules of P^{ad} are adapted by adding some extra conditions to their bodies in order to evaluate them only if appropriate sub-goals have already been generated by the set of control rules. Let r be a rule in P^{ad} with head h^a and with literals $l_1, \ldots, l_n$. r shall only be evaluated if there is a sub-goal *magic_h*^{a} for the head, and if there are sub-goals for each of the derived predicates of the body. For the adorned version of the transitive closure program (Listing 1.1_5) both the first and the second rule must be rewritten. Since there is no derived predicate in the first rule, the only literal which must be added to the rule body is magic_t$^{bf}(X)$, yielding the transformed rule t^{bf}(X,Y) ← magic_t$^{bf}(X)$, r(X,Y). With the second rule having two derived predicates in the rule body, one might expect that three additional magic literals would have to be introduced in the rule body. But since t-1 and t-3 have the same adornment and the same variables for their bound arguments, they share the same magic predicate.

The evaluation of the magic transitive closure program is presented in Listing 1.1_7 for the goal t(a,Answer). Note that in contrast to the naive and semi-naive bottom-up algorithms, only those facts are derived, which are potentially useful for answering the query. In particular, the facts r(1, 2), r(2, 3), and r(3, 1) are never used. Moreover the sub-goal r(d,X) corresponding to the magic predicate magic_t^{bf}(d) is never considered.

```
magic_t^bf(Y) ← magic_t^bf(X), t(X,Y).
t^bf(X,Y) ← magic_t^bf(X), r(X,Y).
t-3^bf(X,Z) ← magic_t^bf(X), t-1^bf(X,Y), magic_t^bf(Y), t-2^bf(Y,Z).
magic_t^bf(a). // the seed

r(a, b). r(b, c). r(c, d).
r(1, 2). r(2, 3). r(3, 1).
```

Listing 1.1_6: The magic templates transformation of the program in Listing 1.1_5 for the query t(a,X)

```
t(a,b) // derived by the seed and rule 2
magic_t^bf(b) // derived by the seed, t(a,b) and rule 1
t(b,c) // derived by magic_t^bf(b), and rule 2
magic_t^bf(c) // derived by magic_t^bf(b), t(b,c) and rule 1
t(c,d) // derived by magic_t^bf(c) and rule 2
t(a,c) // derived by rule 3
t(a,d) // derived by rule 3
t(b,d) // derived by rule 3
```

Listing 1.1_7: Evaluation of program 1.1_6

1.1.3 The Rete Algorithm

The Rete algorithm [27, 23] was originally conceived by Charles L. Forgy in 1974 as an optimized algorithm for inference engines of rule based expert systems. Since then several optimizations of Rete have been proposed, and it has been implemented in various popular expert systems such as Drools, Soar, Clips, JRules and OPS5.

The Rete algorithm is used to process rules with a conjunction of conditions in the body and one or more actions in the head, that are to be carried out when the rule fires. These rules are stored in a so-called *production memory*. The other type of memory that is used by the Rete algorithm is the *working memory*, which holds all the facts that make up the current configuration the rule system is in. A possible action induced by a rule may be the addition of a new fact to the working memory, which may itself be an instance of a condition of a rule, therefore triggering further actions to be carried out in the system.

Avoiding redundant derivations of facts and instances of rule precedents, the Rete algorithm processes production rules in a so-called *Rete network* consisting of *alpha-nodes*, *beta-nodes*, *join-nodes* and *production-nodes*.

Figure 1.1 illustrates the way a Rete network is built and operates. It serves as an animal classification system relying on characteristics such as `has wings`, `has spikes`, `is poisonous`, etc. The example rules exhibit overlapping rule bodies (several atomic conditions such as `X lays eggs` are shared among the rules).

For each atomic condition in the body of a rule, the Rete network features one alpha-node containing all the elements of the working memory that make this atomic condition true. Alpha-nodes are distinguished by shaded rectangles with round corners in Figure 1.1. Although the same atomic condition may occur multiple times distributed over different production rules, only one single alpha node is created in the Rete network to represent it. Therefore the condition `X lays eggs`, which is present in the conditions of all rules except for `p2`, is represented by a single alpha-node in Figure 1.1.

While alpha-nodes represent single atomic conditions of rule bodies, beta-nodes stand for conjunctions of such conditions, and hold sets of tuples of working memory elements that satisfy them. Beta-nodes are depicted as ovals with white background in Figure 1.1.

In contrast to alpha and beta nodes, join nodes do not hold tuples of or single working memory elements, but serve computation purposes only. For each rule in the rule system, there is one production node (depicted as rectangles with grey background in Figure 1.1) holding all the tuples of working memory elements that satisfy all the atomic conditions in its body.

Alpha- and beta-nodes are a distinguishing feature of Rete in that they remember the state of the rule system in a fine grained manner. With beta-nodes storing instantiations of (partial) rule bodies, there is no need of reevaluating the bodies of all rules within the network in the case that the working memory is changed.

Besides *storing* derived facts and instantiations of (partial) rule premises, the Rete network also allows information *sharing* to a large extent. There are two ways that information is shared among rules in the network. The first way concerns the alpha-nodes and has already been mentioned above. If an atomic condition (such as `X feeds milk`) appears within more than one rule, this alpha node is shared among both rules. Needless to say, this is also the case if both conditions are variants (equivalent modulo variable renaming) of each other. The second way that information is shared within the Rete network is by sharing partial rule instantiations between different rules. In Figure 1.1, the conjunction of atomic conditions `(X lays eggs), (X feeds milk)` is common to the rules `p3`, `p4` and `p5`. In a Rete network, instantiations of these partial rule bodies are computed only once and saved within a beta node which is connected (possibly via other beta nodes) to the production nodes of the affected rules.

1.1.4 Basic Backward Chaining: SLD-Resolution

Resolution proofs are refutation proofs, i.e. they show the unsatisfiability of a set of formulas. As it holds that the set of formulas $P \cup \{\neg\varphi\}$ is unsatisfiable iff $P \models \varphi$, resolution may be used to determine entailment (compare Theorem **??**). Observe that a goal $\leftarrow a_1, \ldots, a_n$ is a syntactical variant of the first

order sentence $\forall x_1 \ldots x_m(\bot \leftarrow a_1 \wedge \ldots \wedge a_n)$ where $x_1, \ldots, x_m$ are all variables occurring in $a_1, \ldots a_n$. This is equivalent to $\neg\exists x_1 \ldots x_m(a_1 \wedge \ldots \wedge a_n)$. If we use SLD-resolution[3] to show that a logic program P and a goal $\leftarrow a_1, \ldots, a_n$ are unsatisfiable we can conclude that $P \models \exists x_1 \ldots x_m(a_1 \wedge \ldots \wedge a_n)$.

Definition 1 (SLD Resolvent) *Let C be the clause $b \leftarrow b_1, \ldots, b_k$, G a goal of the form $\leftarrow a_1, \ldots, a_m, \ldots, a_n$, and let θ be the mgu of a_m and b. We assume that G and C have no variables in common (otherwise we rename the variables of C). Then G' is an* SLD resolvent of G and C using θ *if G' is the goal $\leftarrow (a_1, \ldots a_{m-1}, b_1, \ldots b_k, a_{m+1}, \ldots a_n)\theta$.*

Definition 2 (SLD Derivation) *A* SLD derivation *of $P \cup \{G\}$ consists of a sequence $G_0, G_1, \ldots$ of goals where $G = G_0$, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that G_{i+1} is a resolvent from G_i and C_{i+1} using θ_{i+1}. An* SLD-refutation *is a finite SLD-derivation which has the empty goal as its last goal.*

Definition 3 (SLD Tree) *An* SLD tree T w.r.t. a program P and a goal G *is a labeled tree where every node of T is a goal and the root of T is G and if G is a node in T then G has a child G' connected to G by an edge labeled (C, θ) iff G' is an SLD-resolvent of G and C using θ.*

Let P be a definite program and G a definite goal. A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition of $\theta_1, \ldots \theta_n$ to the variables occurring in G, where $\theta_1, \ldots \theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$.

Observe that in each resolution step the selected literal a_m and the clause C are chosen non-deterministically. We call a function that maps to each goal one of its atoms a *computation rule*. The following proposition shows that the result of the refutation is independent of the literal selected in each step of the refutation.

Proposition 1 (Independence of the Computation Rule) *[35] Let P be a definite Program and G be a definite goal. Suppose there is an SLD-refutation of $P \cup \{G\}$ with computed answer θ. Then, for any computation rule R, there exists an SLD-refutation of $P \cup \{G\}$ using the atom selected by R as selected atom in each step with computed answer θ' such that $G\theta$ is a variant of $G\theta'$.*

The independence of the computation rule allows us to restrict the search space: As a refutation corresponds to a branch of in an SLD-tree, to find all computed answers we need to search all branches of the SLD-tree. The independence of the computation rule allows us to restrict our search to branches constructed using some (arbitrary) computation rule.

Lemma 1 *Consider the logic program 1.1_8 with query q =* `← t(1,2)`*:*

```
t(x,y) ← e(x,y).
t(x,y) ← t(x,z), e(z,y).
e(1,2) ← .
e(2,1) ← .
e(2,3) ← .
← t(1,2) .
```

Listing 1.1_8: Transitive Closure

An SLD-tree for program 1.1_8 and q is shown in the following figure. We label the edges of an SLD tree with the number of a rule instead of a rule. We denote by (n') the rule number n where each variable x occurring in rule n is replaced by x'.

[3] SLD is an acronym for Selected Literal Definite Clause

If we want to compare the operational semantics of a program P to its declarative semantics we need a declarative notion of an answer of program P. A *correct answer* for a program P and goal G is a substitution θ such that $P \models G\theta$. Using this notion we can define the soundness and completeness of logic programming.

Proposition 2 (Soundness and Completeness of Logic Programming) *[35] Let P be a program and let Q be a query. Then it holds that*

- *every computed answer of P and G is a correct answer and*
- *for every correct answer σ of P and G there exists a computed answer θ such that θ is more general that σ.*

Observe that to find a computed answers of a program P and goal G operationally one has to visit the leaf of a finite branch in the SLD-tree w.r.t. P and G. The order in which we visit these nodes is not determined by the definition of an SLD-refutation. We call such an order a *search strategy*. An *SLD-procedure* is a deterministic algorithm which is an SLD-resolution constrained by a computation rule and a search strategy.

As SLD-trees are infinite in general, the completeness of an SLD-procedure depends on the search strategy. To be complete, an SLD-procedure must visit every leaf of a finite branch of an SLD-tree within a finite number of steps. A search strategy with this property is called *fair*. Obviously not every search strategy is fair. For example the depth first search strategy used by Prolog is not fair. An example of a fair search strategy is breath first search.

1.1.5 Backward Chaining with Memorization: OLDT-Resolution

As stated in the previous section not every search strategy is complete. This is due to the fact that an SLD-tree is infinite in general. As we only consider finite programs, an SLD-tree may only be infinite if it has an infinite branch. As a branch in an SLD-tree corresponds to an SLD-derivation we denote a branch as $[G_1, G_2, \ldots]$ where $G_1, G_2 \ldots$ are the goals of the corresponding derivation.

A branch $B = [G_1, G_2, \ldots]$ in an SLD-tree may be infinite if there is a sub-sequence $[G_{i_1}, G_{i_2}, \ldots]$ ($i_j < i_k$ if $j < k$) of B such that

- for all $j,k \in \mathbb{N}$ G_{i_j} and G_{i_k} contain an equal (up to renaming of variables) atom or
- for all $j \in \mathbb{N}$ G_{i_j} contains an atom which is a real instance of an atom in $G_{i_{j+1}}$.

Non-termination due to the first condition is addressed by a evaluation technique called *tabling* or *memorization*. The idea of tabling is the idea of dynamic programming: store intermediate results to be able to look these results up instead of having to recompute them. In addition to the better termination properties, performance is improved with this approach.

The OLDT algorithm [48] is an extension of the SLD-resolution with a left to right computation rule. Like SLD-resolution, it is defined as a non-deterministic algorithm.

A subset of the predicate symbols occurring in a program are classified as *table predicates*. A goal is called a *table goal* if its leftmost atom has a table predicate. Solutions to table goals are the intermediate results that are stored. Table goals are classified as either *solution goals* or *look-up goals*. The intuition is that a solution goal 'produces' solutions while a look-up goal looks up the solutions produced by an appropriate solution goal.

An *OLDT-structure* (T, T_S, T_L) consists of an SLD-tree T and two tables, the solution table T_S and the look-up table T_L. The *solution table* T_S is a set of pairs $(a, T_S(a))$ where a is an atom and $T_S(a)$ is a

list of instances of a called the *solutions* of a. The *look-up table* T_L is a set of pairs $(a, T_L(a))$ where a is an atom and p is a pointer pointing to an element of $T_S(a')$ where a is an instance of a'. T_L contains one pair $(a, T_L(a))$ for an atom a occurring as a leftmost atom of a goal in T.

The *extension of an OLDT structure* (T, T_S, T_L) consists of three steps:

1. a resolution step,
2. a classification step, and
3. a table update step.

In the resolution step a new goal is added to the OLDT-tree, in the classification step this new goal is classified as either non-tabled goal or solution goal or look-up goal and in the table update step the solution table and the update table are updated. While step one is equal for non-tabled and solution goals, step two and three are equal for tabled nodes while there is nothing to do in these steps for non-tabled nodes.

Let (T, T_S, T_L) be an OLDT structure and $G = \leftarrow a_1, \ldots, a_n$ a goal in T. If G is a non-tabled goal or a solution goal then in the resolution step a new goal G' is added to T which is connected to G with an edge labeled (C, θ) where G' is the SLD-resolvent of G and C using θ. If G is a look-up node then in the resolution step the new node G' is added to T with an edge labeled $(a \leftarrow, \theta)$ where a is the atom in the solution table that the pointer $T_L(a_1)$ points to and the substitution θ is the mgu of a and a_1. Finally the pointer $T_L(a_1)$ is set to point to the next element of the list it points to.

In the classification step the new goal G' is classified as a non-table goal if its leftmost atom is not a table predicate and a table goal otherwise. If G' is a table goal then G' is classified as a look-up node if there is a pair $(a, T_S(a))$ in the solution table and a is more general than the leftmost atom a' of G'. In this case a new pair (a', p) is added to the look-up table and p points to the first element of $T_S(a)$. If G' is not classified as a look-up node then it is classified as a solution node and a new pair $(a', [])$ is added to the solution table.

In the table update step new solutions are added to the solution table. Recall that the problem we want to tackle here is the recurrent evaluation of equal (up to renaming of variables) atoms in goals. Therefore the 'solutions' we want to store in the solution table are answers to an atom in a goal.

In SLD-resolution the term answer is defined only for goals. This notion can be extended to atoms in goals in the following way. OLDT-resolution uses a left to right computation rule. If the derivation of a goal $G = \leftarrow a_1, \ldots, a_n$ is finite, then there is a finite number n of resolution steps such that the nth resolvent G_n on G is $\leftarrow a_2, \ldots, a_n$. We call the sequence $[G_1, \ldots, G_n]$ a *unit sub-refutation of* a_1 and the restriction of $\theta_1 \ldots \theta_n$ to the variables occurring in a_1 is called an *answer for* a_1.

Now if the goal G produced in the resolution step is the last goal of a unit sub-refutation of a with answer θ then the update step consists in adding θ to the list $T_S(a)$.

Lemma 2 *Reconsider the program from Example 1*

```
t(x,y) ← e(x,y) .
t(x,y) ← t(x,z), e(z,y) .
e(1,2) ← .
e(2,1) ← .
e(2,3) ← .
← t(1,2) .
```

Listing 1.1_9: Transitive Closure

After a sequence of OLDT-resolutions of solution goals or non-tabled goals the OLDT-tree in Figure 1.3 is constructed. To indicate which nodes are solution nodes and which are look-up nodes we prefix solution nodes with 'S:' and look-up nodes with 'L:'.

As the left branch is a unit sub-refutation of $t(1,a)$ with solution $\{a/2\}$ the entry $t(1,2)$ is added to the solution table. As $t(1,a)$ is more general than the leftmost atom of the goal $t(1,z'),e(z',a)$ this goal is classified as a look-up node. Instead of using resolution to compute answers for the first atom of this goal we use the solutions stored in the solution table. The final OLDT-tree is depicted in 1.4:

Observe that the program of example 2 does not terminate with SLD-resolution while it does terminate with OLDT-resolution. The following example shows that OLDT-resolution is not complete in general.

Lemma 3 *Consider the program 1.1_10 and query $q =$* ← p(x)

```
p(x) ← q(x), r .
q(s(x)) ← q(x) .
q(a) ← .
r ← .
← p(x) .
```

Listing 1.1_10: Program for which OLDT resolution is incomplete

After a sequence of OLDT-resolution steps the OLDT-tree in Figure 1.5 is constructed

In the next step the solution $q(a)$ can be used to generate the solution $q(s(a))$ (see Figure 1.6).

It is easy to see that if reduction steps are only applied to the node L:← $q(x'),r$ then no solutions for $p(x)$ will be produced in finite time. Therefore OLDT is not complete in general.

This problem was addressed by the authors of OLDT. They specified a search strategy called *multistage depth-first strategy* for which they showed that OLDT becomes complete if this search strategy is used. The idea of this search strategy is to order the nodes in the OLDT-tree and to apply OLDT-resolution-steps to the nodes in this order. If the node that is the biggest node with respect to that ordering is reduced then a stage is complete and a new stage starts where reduction is applied to the smallest node again. Therefore it is not possible to apply OLDT-steps twice in a row if there are other nodes in the tree which are resolvable.

In the above example it would therefore not be possible to repeatedly apply reductions to the node L:← $q(x'),r$ without reducing the node ← r which yields a solution for $p(x)$.

1.1.6 The Backward Fixpoint Procedure

The last sections have shown bottom up and top down methods for answering queries on Horn logic programs. While the naive and semi-naive bottom up methods suffer from an undirected search for answering queries, the top down methods such as SLD resolution (Section 1.1.4) may often not terminate although the answer can be computed in finite time by a bottom up procedure. Non-termination of top down procedures is addressed by tabling (storing the encountered sub-queries and their solutions) in OLDT resolution (Section 1.1.5) and other advanced top down methods such as QSQ or SLDAL-Resolution[52], the ET^* and ET_{interp}[22, 25] algorithms and the RQA/FQI[37] strategy. The problem of undirected search in forward chaining methods is solved by rewriting the rules such that special atoms representing encountered sub-goals are represented by custom-built atoms and by requiring an

appropriate sub-goal to be generated before a rule of the original program is fired. Two representatives of this second approach are the Alexander[33] and the Magic Set methods (Section 1.1.2).

In [8] a sound and complete query answering method for recursive databases based on meta-interpretation called *Backward Fixpoint Procedure*, is presented, and it is shown that the Alexander and Magic Set methods can be interpreted as *specializations* of the Backward Fixpoint Procedure (BFP) and that also the efficient top down methods based on SLD resolution implement the BFP. Studying the BFP reveals the commonalities and differences between top down and bottom up processing of recursive Horn logic programs and is thus used to top of this chapter.

The backward fixpoint procedure is specified by the meta interpreter in Listing 1.1_11, which is intended to be evaluated by a bottom up rule engine. Facts are only generated in a bottom up evaluation of the interpreter if a query has been issued for that fact or if an appropriate sub-query has been generated by the meta-interpreter itself (Line 1). Sub-queries for rule bodies are generated if a sub-query for the corresponding rule head already exists (Line 2). Sub-queries for conjuncts are generated from sub-queries of conjunctions they appear in (Line 3 and 4). The predicate `evaluate` consults the already generated facts, and may take a single atom or a conjunction as its argument, returning true if all of the conjuncts have already been generated. It must be emphasized that using a *bottom up* rule engine for evaluating the BFP meta-interpreter for an object rule program is equivalent to evaluating this object program *top down*, thereby not generating any facts which are irrelevant for answering the query. For the correctness of the meta-interpreter approach for fixpoint computation and for an example for the evaluation of an object program with the BFP meta-interpreter see [8].

```
fact(Q) ← query_b(Q) ∧ rule(Q ← B) ∧ evaluate(B)
query_b(B) ← query_b(Q) ∧ rule(Q ← B)
query_b(Q_1) ← query_b(Q_1 ∧ Q_2)
query_b(Q_2) ← query_b(Q_1 ∧ Q_2) ∧ evaluate(Q_1)
```

Listing 1.1_11: The backward fixpoint procedure meta interpreter

A direct implementation of the meta-interpreter may lead to redundant computations of facts. To see this consider the application of the meta-interpreter to the object program `p ← q, r` and the query `p`. The relevant instantiated rules of the meta-interpreter contain the ground queries `evaluate(q,r)` and `evaluate(q)`, thereby accessing the fact `q` twice. Getting rid of these redundant computations is elegantly achieved by specifying a bottom up evaluation of a binary version of the predicate `evaluate` as shown in Listing 1.1_12. The first argument of `evaluate` contains the conjuncts which have already been proved, while the second argument contains the rest of the conjunction. Therefore a fact $\texttt{evaluate}(\emptyset, Q)$ represents a conjunction which has not yet been evaluated at all, while $\texttt{evaluate}(Q, \emptyset)$ represents a completely evaluated conjunction. With this new definition of `evaluate` the atoms $\texttt{evaluate}(B)$ and $\texttt{evaluate}(Q_1)$ in Listing 1.1_11 must be replaced by $\texttt{evaluate}(B, \emptyset)$ and $\texttt{evaluate}(Q_1, \emptyset)$, respectively. With this extension, the BFP meta-interpreter of Listing 1.1_11 becomes redundant. A non-redundant version is obtained by only considering rules (1, 5 to 11) of Listings 1.1_11, 1.1_12 and 1.1_13. For the proofs for the redundancy of the rules (1 to 9) on the one hand and for the equivalence of the interpreters made up of rules (1 to 4) and (1, 5 to 11) on the other hand see [8].

```
evaluate(∅, B) ← query_b(Q) ∧ rule(Q ← B)
evaluate(B_1, B_2) ← evaluate(∅, B_1 ∧ B_2) ∧ fact(B_1)
evaluate(B_1 ∧ B_2, B_3) ← evaluate(B_1, B_2 ∧ B_3) ∧ B_1 ≠ ∅ ∧ fact(B_2)
evaluate(B, ∅) ← fact(B)
evaluate(B_1 ∧ B_2, ∅) ← evaluate(B_1, B_2) ∧ B_1 ≠ ∅, fact(B_2)
```

Listing 1.1_12: Implementation of the predicate `evaluate`

```
10 query_b(B_2) ← evaluate(B_1,B_2) ∧ B_2 ≠ (C_1 ∧ C_2)
   query_b(B_2) ← evaluate(B_1,B_2 ∧ B_3)
```

Listing 1.1_13: Replacement rules for the rules 2 to 4 in Listing 1.1_11

The implementation of the backward fixpoint procedure gives rise to two challenges exemplified by the sub-goal $\texttt{query}_b\texttt{(r(x,a))}$, which may be generated during the evaluation of a program by the BFP meta-interpreter. The sub-goal is both nested and non-ground. The main problem with non-ground terms generated by the application of the BFP meta-interpreter to an object program is that for deciding whether a newly derived fact is a logical duplicate of an already derived fact it is not sufficient to perform string matching, but full unification is needed. With *specialization*[28], a common partial evaluation technique used in logic programming, one can get rid of these problems.

Specialization is applied to the BFP meta-interpreter with respect to the rules of the object program. For each rule of the meta-interpreter that includes a premise referring to a rule of the object program, one specialized version is created for each rule of the object program. The first rule of the BFP meta-interpreter (Listing 1.1_11) specialized with respect to the rule $\texttt{p(x)} \leftarrow \texttt{q(x)} \wedge \texttt{r(x)}$ results in the following partially evaluated rule:

```
fact(p(x)) ← query_b(p(x)) ∧ evaluate(q(x) ∧ r(x))
```

Similarly, the specialization of the second rule of the meta-interpreter with respect to the same object rule yields the partially evaluated rule $\texttt{query}_b\texttt{(q(x)} \wedge \texttt{r(x))} \leftarrow \texttt{query}_b\texttt{(p(x))}$.

Another specialization which can be applied to the BFP meta-interpreter is the specialization of the $\texttt{query}_b$ predicate with respect to the predicates of the object program, transforming a fact $\texttt{query}_b\texttt{(p(a))}$ into the fact $\texttt{query}_b\texttt{-p(a)}$ and eliminating some of the nested terms generated during the evaluation. With this transformation the first rule of the BFP is further simplified to:

```
fact(p(x)) ← query_b-p(x) ∧ evaluate(q(x) ∧ r(x))
```

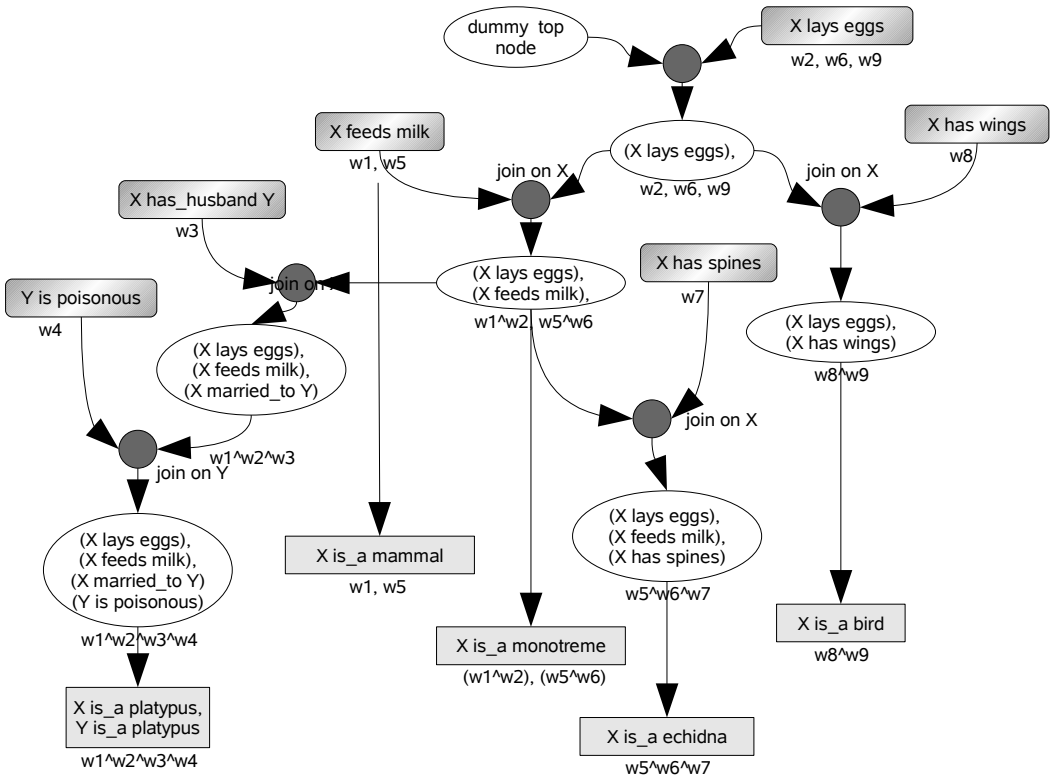
Getting rid of non-ground terms can also be achieved by specialization resulting in an adorned version of the program. Adornment of logic programs is described in the context of the magic set transformation in Section 1.1.2. [8] also discusses the *faithfulness* of representations of sub-queries as facts. Adornments of sub-queries are not completely *faithful* in the sense that the adornment of the distinct queries `p(X,Y)` and `p(X,X)` results in the same adorned predicate $\texttt{p}^{ff}$. As described in Section 1.1.2, multiple adorned versions for one rule of the original program may be generated.

In [8] it is shown that the above specializations of the meta-interpreter made up of the rules (1, 5 to 11) with respect to an object program P and the omittance of the meta-predicates `evaluate` and `fact` yields exactly the supplementary magic set transformation and the Alexander method applied to P. Therefore both of these methods implement the BFP.

Not only can bottom up processing methods be specialized from the BFP, but it can also be used to specify top down procedures such as ET^*, ET_{interp}, *QSQ*, etc. The difference between SLD resolution and the BFP is explained in [8] in terms of the employed data structures. SLD resolution uses a hierarchical data structure which relates sub-queries and proved facts to the queries they belong to. On the other hand, the BFP employs relations – i.e. a flat data structure – for memorizing answers to queries, and therefore allows to share computed answers between queries that are logically equivalent.

SLD resolution and the hierarchical resolution tree can be simulated by the BFP by introducing identifiers for queries, thus allowing to relate facts and sub-queries with queries to the answers of which they contribute. See [8] for details. This simulation prevents sharing of answers between queries and thus makes the evaluation less efficient. One conclusion that can be drawn from the BFP is that it does not make sense to hierarchically structure queries according to their generation. In contrast it makes sense to rely on a static rewriting such as the Alexander or Magic Set rewriting and process the resulting rules with a semi-naive bottom-up rule engine.

Figure 1.1 A Rete Network for Animal Classification



Working memory:
w1: anna feeds milk
w2: anna lays eggs
w3: anna married_to pierre
w4: pierre is poisonous
w5: betty feeds milk
w6: betty lays eggs
w7: betty has spines
w8: tux has wings
w9: tux lays eggs

Production memory:
p1: X lays eggs, X has wings ==> X is_a bird
p2: X feeds milk ==> X is_a mammal
p3: X feeds milk, X lays eggs ==> X is_a monotreme
p4: X feeds milk, X lays eggs, X has_spines ==> X is_a echidna
p5: X feeds milk, X lays eggs, X married_to Y, Y is poisonous
==> X is_a platypus, Y is_a platypus

Figure 1.2 An SLD tree for program 1.1_8

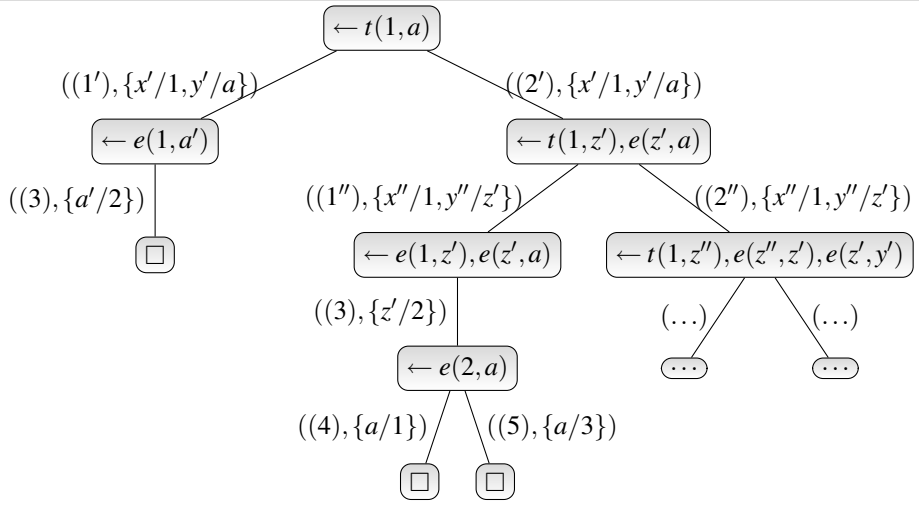


Figure 1.3 An intermediary OLDT tree for program 1.1_9

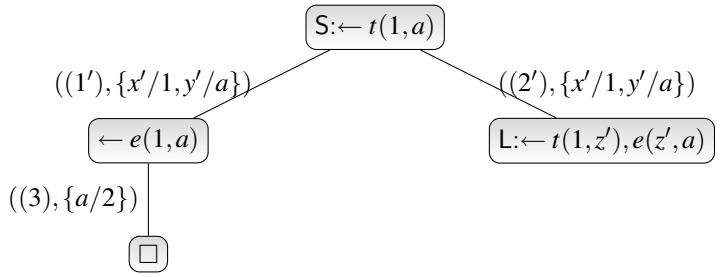


Figure 1.4 The final OLDT tree for program 1.1_9

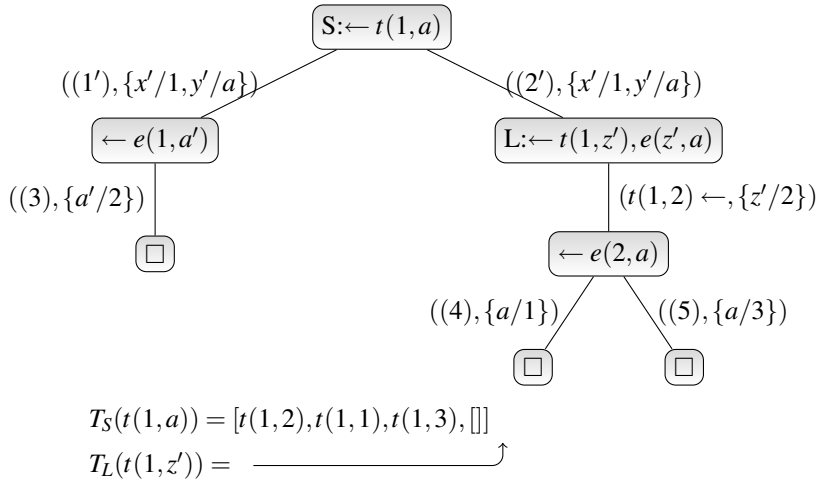


Figure 1.5 An intermediary OLDT tree for program 1.1_10

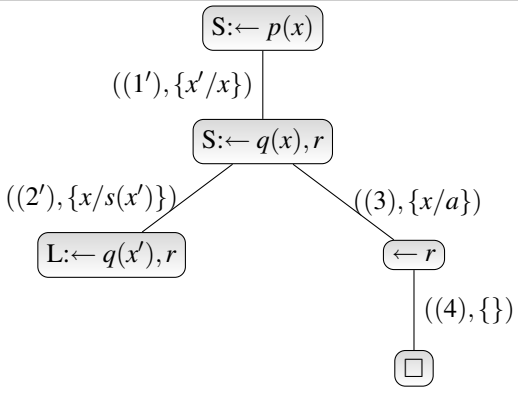
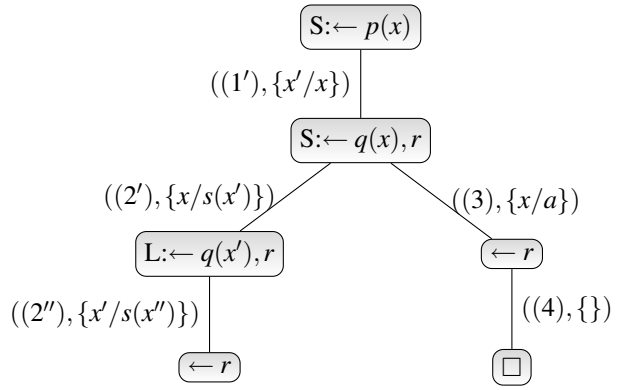


Figure 1.6 An intermediary OLDT tree for program 1.1_10



$T_S(p(x)) = []$

$T_S(q(x)) = [q(a), q(s(a))]$

$T_L(q(x')) =$ ⟶↑

1.2 Rule Sets with Non-monotonic Negation

In the previous chapter evaluation methods for logic programs without negation are examined. This chapter considers a more general form of logic programs, namely ones that use negation. The rules considered in this chapter are all of the form

$$A \leftarrow L_1, \ldots, L_n$$

where the $L_i, 1 \leq, i \leq n$ are literals, and A is an atom. Thus negative literals are allowed in the bodies of rules, but not in their heads. It is important to note that augmenting logic programming with negation increases expressivity and is necessary for deriving certain information from a database.

In Section **??** two important semantics for logic programming with negation have been described: The *stable model semantics* (See Section **??**) and the *well-founded model semantics* (Section **??**). However, this declarative semantics does not provide an easy to implement algorithm neither for computing the entailments of a logic program with negation nor for answering queries with respect to such programs. In fact the greatest unfounded sets in the definition of the well-founded semantics and the stable models in the stable models theory must be guessed.

In this section constructive algorithms for computing the so-called *iterative fixpoint semantics*, the stable model semantics and the well-founded model semantics are described and applied to example programs to better illustrate their functioning.

The kind of negation which is generally used in logic programming is called negation as failure and can be described as follows: A negated literal $\neg A$ is true, if its positive counterpart A cannot be derived from the program. A possible application of negation as failure is given in Listing 1.2_14. Since `male(eduard)` is given and `married(eduard)` cannot be derived, `bachelor(eduard)` is logical consequence of the program, while `bachelor(john)` is not. Negation as failure is also known under the term *non-monotonic* negation, because the addition of new facts to a program may cause some of its entailments to be no longer true: The addition of `married(eduard)` to the program in Listing 1.2_14 invalidates the conclusion `bachelor(eduard)`.

```
married(john).
male(john).
male(eduard).
bachelor(X) ← male(X), not married(X)
```

Listing 1.2_14: An Illustration for the Use of Negation as Failure

The adoption of negation as failure brings about several interesting, not to say intricate questions. Consider the program in Listing 1.2_15. If the literal `q` is assumed to be false, then the literal `p` must be true according to the second rule. This, however, causes the literal `q` to be true. On the other hand there is no possibility of deriving the literal `q`. Hence the semantics of Program 1.2_15 is not clear. Therefore syntactical restrictions on logic programs have been proposed, to ensure that all programs satisfying these restrictions have an intuitive declarative semantics.

```
q ← p.
p ← not q.
```

Listing 1.2_15: A program with Recursion through negation

1.2.1 Computation of the Iterative Fixpoint Semantics for Stratified Logic Programs with Negation as Failure

One possibility of limiting the use of negation is by *stratification* (see Definition **??** in Subsection **??**).

It is easy to see that for some programs no stratification can be found. Since in Program 1.2_15 `q` depends on `p` by the first rule and `p` depends on `q` by the second rule, they would have to be defined in the same stratum. Since `q` depends *negatively* on `p`, this case is precluded by the third premise above. A program for which no stratification can be found is called *non-stratifiable*, programs for which a stratification exists are called *stratifiable*. The iterative fixpoint semantics does not provide a semantics for non-stratifiable programs.

[1] defines the semantics for stratified logic programs as an iterated fixpoint semantics based on the immediate consequence operator $\mathbf{T}_P$ (Definition **??**) as follows. Let $S_1, \ldots, S_n$ be a stratification for the program P. Recall that $\mathbf{T}_P^i(I)$ denotes the i-fold application of the immediate consequence operator to the database instance I. Then the semantics of the program P is the set M_n where M_0 is defined as the initial instance over the extensional predicate symbols of P, and the M_i are defined as follows:

$$M_1 := \mathbf{T}_{S_1}^{\omega}(M_0),\ M_2 := \mathbf{T}_{S_2}^{\omega}(M_1),\ \ldots,\ M_n := \mathbf{T}_{S_n}^{\omega}(M_{n-1})$$

This procedure shall be illustrated at the example program in Listing 1.2_16. The intensional predicate symbols `has_hobbies`, `has_child`, `married`, and `bachelor` can be separated into the strata $S_1 := \{\texttt{has\_hobbies}\}$, $S_2 := \{\texttt{has\_child}, \texttt{married}\}$, $S_3 := \{\texttt{bachelor}\}$. The initial instance $M_0 = \{\{john\}_{human}, \{john\}_{plays\_the\_piano}, \{john\}_{male}\}$ is directly derived from the facts of the program. Since `has_hobbies` is the only element of S_1, only the first rule is relevant for the computation of $M_1 := M_0 \cup \{\{john\}_{has\_hobbies}\}$. The second and the third rule are relevant for the computation of M_2, which is the same instance as M_1. Finally, the fourth rule allows to add the fact `bachelor(john)` yielding the final instance $M_3 := M_2 \cup \{\{john\}_{bachelor}\}$.

```
human(john).
male(john).
plays_the_piano(john).

has_hobbies(X) ← plays_the_piano(X).
has_child(X) ← human(X), not has_hobbies(X).
married(X) ← human(X), has_child(X).
bachelor(X) ← male(X), not married(X).
```

Listing 1.2_16: A stratifiable program with negation as failure

1.2.2 Magic Set Transformation for Stratified Logic Programs

While the computation of the iterative fixpoint of a stratified logic program allows to answer an arbitrary query on the program, it is inefficient in the sense that no goal-directed search is performed. One method for introducing goal-directedness into logical query answering, that is also relatively easy to implement, is the magic set rewriting as introduced in Section 1.1.2. The task of transferring the magic set approach to logic programs with negation has therefore received considerable attention [16], [34], [4], [42], [3].

The main problem emerging when applying the magic set method to programs with negative literals in rule bodies is that the resulting program may not be stratified. There are two approaches to dealing with this situation. The first one is to use a preprocessing stage for the original program in order to

obtain a stratified program under the magic set transformation and is pursued by [16]. The second one is to accept the unstratified outcome of the magic set transformation and to compute some other semantics which deals with unstratified programs. This second approach is employed by [6], which proposes to compute the well-founded model by Kerisit's *weak consequence operator* at the aid of a new concept called *soft stratification*.

Because of its simplicity and straightforwardness, only the first approach is presented in this article. In [16] three causes for the unstratification of a magic-set transformed program are identified:

- both an atom `a` and its negation `not a` occur within the body of the same rule of the program to be transformed.
- a negative literal occurs multiple times within the body of a rule of the original program
- a negative literal occurs within a recursive rule

With `b` occurring both positively and negatively in the body of the first rule, Listing 1.2_17 is an example for the first cause of unstratification. The predicate symbol `c` is an extensional one, and therefore no magic rules are created for it, `a` and `b` are intensional ones. When the magic set transformation from the last chapter is naively applied to the program and the query `a(1)`, which means that negated literals are transformed in the very same way as positive ones, the magic set transformed program in Listing 1.2_18 is not stratified, because it contains a negative dependency cycle among its predicates: `magic_b`^{b} negatively depends on `b`^{b}, which again depends on `magic_b`^{b}.

```
a(x) ← not b(x), c(x,y), b(y).
b(x) ← c(x,y), b(y).
```

Listing 1.2_17: A program leading to unstratification under the naive magic set transformation

```
magic_a^b(1).
magic_b^b(x) ← magic_a^b(x).
magic_b^b(y) ← magic_a^b(x), not b^b(x), c(x,y).
a(x) ← magic_a^b(x), not b^b(x), c(x,y), b^b(y).
magic_b^b(y) ← magic_b^b(x), c(x,y).
b(x) ← magic_b^b(x), c(x,y), b(y).
```

Listing 1.2_18: The unstratified outcome of the magic set transformation applied to Program 1.2_17

[16] proposes to differentiate the contexts in which a negative literal is evaluated by numbering the occurrences of the literal in the rule body before the magic set transformation is applied. The numbered version of Listing 1.2_17 is displayed in Listing 1.2_19 where the two occurrences of `b` have been numbered. Additionally, for each newly introduced numbered predicate symbol p_i its defining rules are copies from the definition of the unnumbered symbol p, with all occurrences of p replaced by p_i. In this way, the semantics of the program remains unchanged, but the program becomes stratified under the magic set transformation, as can be verified in Listing 1.2_20.

```
a(x) ← not b_1(x), c(x,y), b_2(y).
b_1(x) ← c(x,y), b_1(y).
b_2(x) ← c(x,y), b_2(y).
```

Listing 1.2_19: Program 1.2_17 with differentiated contexts for the literal `b`

```
magic_a^b(1).
magic_b_1^b(x) ← magic_a^b(x).
magic_b_2^b(y) ← magic_a^b(x), not b_1^b(x), c(x,y).
a^b(x) ← magic_a^b(x), not b_1^b(x), c(x,y), b_2^b(y).
magic_b_1^b(y) ← magic_b_1^b(x), c(x,y).
b_1(x) ← magic_b_1^b(x), c(x,y), b_1(y).
magic_b_2^b(y) ← magic_b_2^b(x), c(x,y).
b_2(x) ← magic_b_2^b(x), c(x,y), b_2(y).
```

Listing 1.2_20: The stratified outcome of the magic set transformation applied to program 1.2_19

Also for the second and third source of unstratification, elimination procedures can be specified that operate on the adorned rule set, but are carried out prior to the magic set transformation. For more details and a proof, that the resulting programs are indeed stratified see [16].

1.2.3 Computation of the Stable Model Semantics

While the iterative fixpoint semantics provides an intuitive and canonical semantics for a subset of logic programs with negation as failure, several attempts have been made to assign a semantics to programs which are not stratifiable. One of these attempts is the *stable model semantics* (see Section **??**).

An example for a program which is not stratifiable but is valid under the stable model semantics is given in Listing 1.2_21.

```
married(john, mary).
male(X) ← married(X,Y), not male(Y).
```

Listing 1.2_21: A non-stratifiable program with a stable model semantics

The stable model semantics for a program P is computed as follows: In a first step all rules containing variables are replaced by their ground instances. In a second step the program is transformed with respect to a given model M into a program $GL_M(P)$ by deleting all those rules which contain a negative literal $not(L)$ in their body where L is contained in M, and by deleting all those negative literals $not(L)$ from the rules for which L is not contained in M. Clearly, the semantics of the program P remains unchanged by both of these transformations with respect to the particular model M. Since this transformation has first been proposed by Gelfond and Lifschitz in [29], it is also known under the name *Gelfond-Lifschitz-Transformation* (See also Definition **??**).

An Herbrand interpretation M is a *stable set* of P if and only if it is the unique minimal Herbrand model of the resulting negation-free program $GL_M(P)$. See Definition **??** and Lemma **??**. The stable model semantics for a program P (written $S_\Pi(P)$) is defined as the stable set of P, and remains undefined if there is none or more than one of them.

The Gelfond-Lifschitz-Transformation of Listing 1.2_21 with respect to the set $M := \{$`married(john, mary)`, `male(John)`$\}$ results in the Program in Listing 1.2_22. Rule instances and negative literals of rule instances have been crossed out according to the rules mentioned above. Since the unique minimal Herbrand model of the resulting Program is also $M = \{$`married(john, mary)`, `male(John)`$\}$, M is a stable set of the original program in 1.2_21. Since there are no other stable sets of the program, M is its stable model.

```
married(john, mary).
```

```
male(john) ← married(john, mary), not male(mary).
male(mary) ← married(mary, john), not male(john).
male(mary) ← married(mary, mary), not male(mary).
male(john) ← married(john, john), not male(john).
```

Listing 1.2_22: Listing 1.2_21 transformed with respect to the set {`married(john, mary)`, `male(John)`}

An efficient implementation of the stable model semantics (and also the well-founded model semantics) for range-restricted and function-free normal logic programs is investigated in [38] and its performance is compared to that of SLG resolution in [15].

1.2.4 A More Efficient Implementation of the Stable Model Semantics

The approach of [38] recursively constructs all possible stable models by adding one after another positive and negative literals from the set of *negative antecedents* (Definition 4) to an intermediate candidate full set B (see Definition 6). The algorithm (see Listing 1.2_23) makes use of backtracking to retract elements from B and to find all stable models of a program. In addition to the program P itself and the intermediate candidate set B, the algorithm takes a third argument which is a formula ϕ that is tested for validity whenever a stable model is found. The algorithm returns true if there is a stable model of ϕ and false otherwise. Moreover it can be adapted to find all stable models of a program or to determine whether a given formula is satisfied in all stable models.

Definition 4 (Negative Antecedents) *Given a logic program P, the set of its negative antecedents $NAnt(P)$ is defined as all those atoms a such that not(a) appears within the body of a rule of P.*

Definition 5 (Deductive closure) *The deductive closure $Dcl(P,L)$ of a program P with respect to a set of literals L is the smallest set of atoms containing the negative literals L^- of L, which is closed under the following set of rules:*

$$R(P,L) := \{ \quad c \leftarrow a_1,\ldots,a_n \mid c \leftarrow a_1,\ldots,a_n, not(b_1),\ldots,not(b_m) \in P, \quad \{not(b_1),\ldots,not(b_m) \subseteq L^-\}\} \tag{1.1}$$

Definition 6 (Full Sets (from [38])) *A set Λ of not-atoms (negated atoms) is called P-*full *iff for all $\phi \in NAnt(P)$, $not(\phi) \in \Lambda$ iff $\phi \notin Dcl(P,\Lambda)$.*

Lemma 4 *Consider the logic program $P := \{q \leftarrow not(r).q \leftarrow not(p).r \leftarrow q.\}$ The negative antecedents of the program are $NAnt(P) = \{r,p\}$. $\Lambda_1 := \{not(r), not(p)\}$ is not a full set with respect to P because $not(r)$ is in Λ_1, but r is in the deductive closure $Dcl(P,\Lambda_1)$. The only full set with respect to P is $\Lambda_2 := \{not(p)\}$: $not(p) \in \Lambda_2$ and $p \notin Dcl(P,\Lambda_2)$ holds for p and $not(r) \notin \Lambda_2$ and $r \in Dcl(P,\Lambda_2)$ holds for the other element r of $NAnt(P)$.*

Theorem 1 (Relationship between full sets and stable models ([38])) *Let P be a ground program and Λ a set of not-atoms (negated atoms).*
(i) If Λ is a full set with respect to P then $Dcl(P,\Lambda)$ is a stable model of P.
(ii) If Δ is a stable model of P then $\Lambda = not(NAnt(P) - \Delta)$ is a full set with respect to P such that $Dcl(P,\Lambda) = \Delta$.

According to theorem 1 it is sufficient to search for full sets instead of for stable models, because stable models can be constructed from these full sets. This approach is pursued by the algorithm in Listing 1.2_23.

Definition 7 (L covers A) *Given a set of ground literals L and a set of ground atoms A, L is said to* cover *A, iff for every atom a in A either $a \in L$ or $not(a) \in L$ holds.*

```
function stable_model(P,B,φ)
  let B' = expand(P,B) in
    if conflict(P,B') then false
    else
      if (B' covers NAnt(P)) then test(Dcl(P,B'),φ)
      else
        take some χ ∈ NAnt(P) not covered by B'
        if stable_model(P, B' ∪ {not(χ)},φ) then true
            else stable_model(P, B' ∪ {χ},φ)
```

Listing 1.2_23: Efficient computation of the stable model semantics for a logic program

The algorithm is not fully specified, but relies on the two functions `expand` and `conflict`, which undergo further optimization. The idea behind the function expand is to derive as much further information as possible about the common part B of the stable models that are to be constructed without losing any model. One could also employ the identity function as an implementation for `expand`, but by burdening the entire construction of the full sets on the backtracking search of the function `stable_model`, this approach would be rather inefficient. A good choice for the `expand` function is to return the least fixpoint of the *Fitting operator* $F_P(B)$:

Definition 8 (Fitting operator F_P) *Let B be a set of ground literals, and P a ground logic program. The set $F_P(B)$ is defined as the smallest set including B that fulfills the following two conditions:*
(i) for a rule $h \leftarrow a_1,\ldots,a_n,not(b_1),\ldots,not(b_m)$ with $a_i \in B, 1 \leq i \leq n$ and $not(b_j), \in B, 1 \leq j \leq m$, h is in $F_P(B)$.
(ii) if for an atom a such that for all of its defining rules, a positive premise p is in its body and $not(p)$ is in B, or a negative literal $not(p)$ is in its body with p in B, then a is in $F_P(B)$.

The function `conflict` returns true whenever (i) B covers $NAnt(P)$, and (ii) if there is an atom a in the set B such that $a \notin Dcl(P,B)$ or a literal $not(a) \in B$ such that $a \in Dcl(P,B)$. In this way, `conflict` prunes the search for full sets (and therefore the search for stable models) by detecting states in which no stable model can be constructed as early as possible. For further optimizations regarding the computation of the stable model semantics with the function `stable_model` the reader is referred to [38].

1.2.5 Computation of the Well-Founded Model Semantics

Another approach to defining a semantics for logic programs that are neither stratifiable nor locally stratifiable is the *well-founded model approach* [49] (see Section **??**).

Recall that in this context the term *interpretation* refers to a set of positive or negative literals $\{p_1,\ldots,p_k,not(n_1),\ldots,not(n_i)\}$ and that the notation $\overline{S}$, with $S = \{s_1,\ldots,s_n\}$ being a set of atoms, refers to the set $\{\neg s_1,\ldots,\neg s_n\}$ in which each of the atoms is negated.

An *unfounded set* (see Section **??**) of a logic program P with respect to an interpretation I is a set of (positive) atoms U, such that for each instantiated rule in P which has head $h \in U$, at least one of the following two conditions applies.

- the body of the rule is not fulfilled, because it contains either a negative literal `not(a)` with `a`$\in I$ or a positive literal `a` with `not(a)`$\in I$.
- the body of the rule contains another (positive) atom $a \in U$ of the unfounded set.

The greatest unfounded set turns out to be the union of all unfounded sets of a program. Note that the definition above does not immediately provide an algorithm for finding the greatest unfounded set. In this subsection, however, a straight-forward algorithm is derived from the definition and in the following subsection, a more involved algorithm for computing the well-founded semantics is introduced.

The computation of the well founded semantics is an iterative process mapping interpretations to interpretations and involving the computation of immediate consequences and greatest unfounded sets. The initial interpretation is the empty set, which reflects the intuition that at the beginning of the program examination, nothing is known about the entailments of the program. The iteration uses the following three kinds of mappings:

- the immediate consequence mapping $\mathbf{T}_P(I)$ of the program with respect to an interpretation I
- the greatest unfounded set mapping $\mathbf{U}_P(I)$, which finds the greatest unfounded set of a program with respect to an interpretation I
- $\mathbf{W}_P(I) := \mathbf{T}_P(I) \cup \overline{\mathbf{U}_P(I)}$ which maps an interpretation to the union of all of its immediate consequences and the set of negated atoms of the greatest unfounded set.

The well-founded semantics (see Section **??**) of a logic program is then defined as the least fixpoint of the operator $\mathbf{W}_P(I)$. The computation of well founded sets and of the well founded semantics is best illustrated by an example (see Listing 1.2_24). The set of immediate consequences $\mathbf{T}_P(\emptyset)$ of the empty interpretation of the program 1.2_24 is obviously the set $\{c(2)\}$. The greatest unfounded set $\mathbf{U}_P(\emptyset)$ of the program with respect to the empty interpretation is the set $\{d(1), f(2), e(2), f(1)\}$. $f(1)$ is in $\mathbf{U}_P(\emptyset)$, because there are no rules with head $f(1)$, and therefore the conditions above are trivially fulfilled.

Note that the fact `a(1)` is not an unfounded fact with respect to the interpretation $\emptyset$, although one is tempted to think so when reading the program as a logic program with negation as failure semantics.

The three atoms $\{d(1), f(2), e(2)\}$ form an unfounded set, because the derivation of any of them would require one of the others to be already derived. There is no possibility to derive any of them first. Hence, according to the well-founded semantics, they are considered false, leading to $I_1 := \mathbf{W}_P(\emptyset) = \mathbf{T}_P(\emptyset) \cup \overline{\mathbf{U}_P(\emptyset)} = \{c(2)\} \cup \overline{\{d(1), f(2), e(2), f(1)\}} = \{c(2), \neg d(1), \neg f(2), \neg e(2), \neg f(1)\}$.

In the second iteration `a(1)` is an immediate consequence of the program, but still neither one of the atoms `a(2)` and `b(2)` can be added to the interpretation (also their negated literals cannot be added). After this second iteration the fixpoint $\{c(2), \neg d(1), \neg f(2), \neg e(2), \neg f(1), a(1)\}$ is reached without having assigned a truth value to the atoms `a(2)` and `b(2)`.

```
b(2) ← ¬ a(2).
a(2) ← ¬ b(2).

d(1) ← f(2), ¬ f(1).
e(2) ← d(1).
f(2) ← e(2).

a(1) ← c(2), ¬ d(1).
c(2).
```

Listing 1.2_24: Example program for the well-founded semantics

The computation of this partial well-founded model involves guessing the unfounded sets of a program P. If P is finite, all subsets of the atoms occurring in P can be tried as candidates for the unfounded sets. In practice those atoms that have already been shown to be true or false do not need to be reconsidered, and due to the fact that the union of two unfounded sets is an unfounded set itself, the greatest unfounded set can be computed in a bottom up manner, which decreases the average case complexity of the problem. Still, in the worst case $O(2^a)$ sets have to be tried for each application of the operator $\mathbf{U}_P$, with a being the number of atoms in the program. In [50] a deterministic algorithm for computing the well-founded semantics of a program is described.

1.2.6 Computing the Well-Founded Semantics by the Alternating Fixpoint Procedure

The central idea of the alternating fixpoint procedure[50] is to iteratively build up a set of negative conclusions $\tilde{A}$ of a logic program, from which the positive conclusions can be derived at the end of the process in a straightforward way. Each iteration is a two-phase process transforming an underestimate of the negative conclusions $\tilde{I}$ into a temporary overestimate $\tilde{\mathbf{S}}_P(\tilde{I})$ and back to an underestimate $\mathbf{A}_P(\tilde{I}) := \tilde{\mathbf{S}}_P(\tilde{\mathbf{S}}_P(\tilde{I}))$. Once this two-phase process does not yield further negative conclusions, a fixpoint is reached. The set of negative conclusions of the program is then defined as the least fixpoint $\tilde{A} := \mathbf{A}_P^{\omega}(\emptyset)$ of the monotonic transformation $\mathbf{A}_P$.

In each of the two phases of each iteration the fixpoint $S_P(\tilde{I}) := \mathbf{T}_{P'}^{\omega}(\emptyset)$ of an adapted version of the immediate consequence operator corresponding to the ground instantiation of the program P_H plus the set $\tilde{I}$ of facts that are already known to be false, is computed.

In the first phase the complement $\tilde{\mathbf{S}}_P(\tilde{I}) := \overline{(H - \mathbf{S}_P(\tilde{I}))}$ of this set of derivable facts constitutes an overestimate of the set of negative derivable facts, and in the second phase the complement $\tilde{\mathbf{S}}_P(\tilde{\mathbf{S}}_P(\tilde{I}))$ is an underestimate.

Let's now turn to the adapted immediate consequence operator. As in the previous sections, the algorithm does not operate on the program P itself, but on its Herbrand instantiation P_H. Based on the Herbrand instantiation P_H and a set of negative literals $\tilde{I}$ a derived program $P' := P_H \cup \tilde{I}$ and a slightly altered version of the immediate consequence operator $\mathbf{T}_{P'}$ of P' are defined.

A fact f is in the set $\mathbf{T}_{P'}(I)$ of immediate consequences of the program P' if all literals l_i in the body of a rule with head f are fulfilled. The difference to the previous definition of the immediate consequence operator is that the bodies of the rules are not required to be positive formulas, but may contain negative literals as well. A negative literal in the rule body is only fulfilled, if it is explicitly contained in the program P' (stemming from the set $\tilde{I}$ which is one component of P'). A positive literal in the body of P' is fulfilled if it is in the interpretation I.

For the proof of the equivalence of the partial models computed by the well-founded model semantics and the alternating fixpoint algorithm the reader is referred to [50].

The computation of the well-founded semantics of the program in Listing 1.2_24 with the alternating fixpoint procedure is achieved without guessing well-founded sets by the following steps.

- $\mathbf{S}_P(\emptyset) = \{c(2)\}$. The fact $a(1)$ cannot be derived because negation is not treated as negation as failure, but only if the negated literal is in $\tilde{I}$. Similarly, neither one of the facts $b(2)$ and $a(2)$ are in $\mathbf{S}_P(\emptyset)$.

- $\tilde{\mathbf{S}}_P(\emptyset) = \overline{(H - \mathbf{S}_P(\emptyset))} = \overline{\{a(2), b(2), f(2), f(1), d(1), e(2), f(2), c(2), a(1)\}\text{-}\{c(2)\}} = \{\neg a(2), \neg b(2), \neg f(2), \neg f(1), \neg d(1), \neg e$ is the first overestimate of the derivable negative facts.

- $\mathbf{S}_P(\tilde{\mathbf{S}}_P(\emptyset)) = \{c(2), a(1), b(2), a(2)\}$ and thus $\mathbf{A}_P(\emptyset) = \tilde{\mathbf{S}}_P(\tilde{\mathbf{S}}_P(\emptyset)) = \{\neg f(2), \neg f(1), \neg d(1), \neg e(2)\}$ is the second underestimate of the derivable negative literals (the first one was the emptyset).
- $\tilde{\mathbf{S}}_P(\tilde{A}_P(\emptyset)) = \{\neg a(2), \neg b(2), \neg f(2), \neg f(1), \neg d(1), \neg e(2)\}$
- $\mathbf{A}_P(\mathbf{A}_P(\emptyset)) = \{\neg f(2), \neg f(1), \neg d(1), \neg e(2)\} = \mathbf{A}_P(\emptyset)$ means that the fixpoint has been reached and $\tilde{A} = \mathbf{A}_P(\emptyset)$ is the set of of negative literals derivable from the program.
- The well founded partial model of the program is given by $\tilde{A} \cup \mathbf{S}_P(\tilde{A}) = \{\neg f(2), \neg f(1), \neg d(1), \neg e(2), c(2), a(1)\}$, which is the same result as in the previous section.

For finite Herbrand universes the partial well-founded model is computable in $O(h)$ with h being the size of the Herbrand Universe [50].

1.2.7 Other Methods for Query Answering for Logic Programs with Negation

While this chapter gives a first idea on methods for answering queries on stratified and general logic programs with negation, many approaches have not been mentioned. The previous sections have shown different ways of implementing the stable model semantics and the well-founded model semantics for general logic programs mainly in a forward chaining manner (except for the magic set transformation, which is a method of introducing goal-directed search into forward chaining).

The best known backward chaining method for evaluating logic programs with negation is an extension of SLD resolution with negation as failure, and is called SLDNF [14][2][45][24]. SLDNF is sound with respect to the completion semantics [18] of a logic program and complete for Horn logic programs [32].

Przymusinski introduced SLS resolution[41] as a backward chaining operational semantics for general logic programs under the perfect model semantics [40]. SLS resolution was extended by Ross to global SLS-resolution[43], which is a procedural implementation of the well-founded model semantics.

Chapter 2

Chaining and Memoization in Xcerpt: An Outlook

2.1 Outline

The second part of this deliverable briefly outlines some of the issues involved in applying the techniques for efficient rule chaining discussed in the previous chapter to the Web query language Xcerpt developed in the I4 working group. In the following, we assume a rough working knowledge of Xcerpt.

To illustrate the novel issues we start with a use case around ad-hoc search or crawling where neither query nor the extent of the relevant data is known a priori. Thus it is not available for preprocessing such as indexing but must be retrieved and processed during query processing. Furthermore, Xcerpt reasoning abilities can be employed to analyze not only the syntactical but also the semantical relations of the queried data as expressed, e.g., in RDF. After introducing said use case we discuss efficient evaluation of (locally stratified) reasoning programs using a novel form of subsumption that is capable to treat not only complete (as conventional subsumption in logic programming) but also incomplete Xcerpt terms. In this way, it addresses the major obstacle to adopting the efficient chaining techniques surveyed in the previous chapter to Xcerpt.

This work will continue after the end of REWERSE and will, eventually, be integrated into the I4 Xcerpt prototype that currently implements chaining in a naive or semi-naive way.

Despite of the dominance of web search giants such as Google or Yahoo, today's search engines exhibit three major shortcomings: (a) web pages of the so-called *invisible web* remain unindexed and thus useless to search engine users, (b) the expressivity of search queries is very limited when compared to sophisticated query languages such as XQuery, SPARQL , or Xcerpt, and (c) the semantics of web data is generally ignored. As of today, the first shortcoming is being tackled by the employment of focused crawlers, whereas the second problem remains unsolved due to the complexity of expressive query evaluation, and the third shortcoming is only acknowledged in the domain of query answering by providing RDF query languages such as SPARQL or RQL. We propose a new concept called *ad-hoc search* to face all three challenges and combine focused crawling with sophisticated query answering on semi-structured data using the versatile query language Xcerpt.

Ad-hoc search is the realm of complex queries and semantic query answering. General purpose search, in contrast, is of today the realm of distributed data processing, huge index structures, and simple queries. Indeed, complex queries or semantic query processing cannot, as of today, be performed on the

huge data quantities that general purpose search engines must process.

Applications of ad-hoc search could be (a) the scheduling of a soccer match in a small town by crawling the hcalender sites of people interested in sports, or (b) solving the single source shortest path problem for a person who published a FOAF document about himself, or (c) finding open-source projects for a given developer described by DOAP files. We demonstrate the suitability of the declarative rule-based language Xcerpt^{RDF} for ad-hoc search tasks, demonstrate the need for declarative and recursive languages with negation as failure for writing crawlers, and show how subsumption tests can improve efficiency in evaluating ad-hoc search programs and semantic search queries in general.

2.2 Introduction

With the growth of semi-structured data on the Web still following an exponential curve, and with the Web itself becoming the preferred way of deploying data-intensive applications in sciences such as biology, physics, genetics, computer science, etc, finding relevant information and search becomes ever more important.

The rise of the Web as a medium for information interchange, the emergence of the Semantic Web as a means to allow automatic transformation, reasoning and processing of Web content, and the usage of microformats and ontologies to represent knowledge in a coherent manner have spurred a rich diversity of formats that are present on the Web, such as XML, RDF, FOAF, DOAP, Dublin Core, SKOS, etc. Documents in these formats are generally interlinked with each other and related information is often dispersed over multiple documents, such that information relevant to a search query must be collected following inter-document links.

Since the beginning of the HTML era, search engines use crawlers for gathering huge amounts of documents to be indexed. Several crawling strategies have been developed and compared [19, 46, 36, 39] with respect to their ability of finding the most relevant documents. Once these documents are saved to disk, indexing techniques such as inverted list indexing[7, 12] are applied to allow the instantaneous answering of occurrence queries for words inside of these documents.

While the deployment of web search engines continues to be a huge success, and has both a large impact on our society and economy, the challenges to efficient, reliable and accurate search have rather increased than been solved. More importantly, with the advent of semantic information on the web, the economic potential of semantic search is much greater than that of ordinary search. In this section we give a short overview over the challenges that have to be met, and argue that they can be overcome by *ad-hoc search*, which is on-the-fly focused crawling, reasoning and complex querying within the same program.

- In order to optimally exploit information supplied in the diverse formats enumerated above, a whole lot of different indexing techniques must be applied to data on the web. While it is sufficient to generate inverse list indexes for words in pure HTML documents, the type of index which fits best RDF documents such as FOAF or DOAP, or HTML documents annotated with microformats is not so clear, and may vary with each usage scenario of the data.
- While static web pages make up only a small portion of the entire information available on the web, dynamic web pages generated from databases, also known as the *invisible*, often remain unindexed by search engines, and are thus invisible for the majority of users of the Web. In [**?**] Sherman and Price distinguish four types of invisibility of Web Content:
  - The *opaque web* consists of web pages that search engines simply do not crawl, since they are considered not important enough for the average user. Sometimes the depth of the crawl

on each site is limited, assuming that just a sample of the pages of each site is sufficient for the search engines' users. In other cases web pages are not crawled frequently enough or disconnected from the rest of a web site.

- The *private web* consists of pages that have been deliberately excluded from being indexed by search engine crawlers. This may either be due to password protection or due to `robots.txt` files or the "noindex" option in the head of html files.
- [**?**] defines the *proprietary web* as those web sites which require a registration or agreement to terms of use of the content of a web site by the user. A search engine crawler can neither register nor give such an agreement to the web site provider, leaving pages on these pages unindexed.
- The *truly invisible web* is defined as all those pages which remain unindexed, since they are in a file format that is harder to read for search engines such as compressed files or flash. Moreover it contains dynamically generated web pages, or information from databases that must be extracted by providing a query string.

The opaque web is certainly a valid application area for ad-hoc search, provided the user has an idea about where to find relevant pages. Also the private web could be a suitable application in the case that the ad-hoc search program's author has access to pages that are not accessible, or should not be accessed by ordinary search engine crawlers. Registration is a technical hindrance to crawling of web-sites, which can be overcome unpunished, if the result of a private search remains unpublished, as it is the case for ad-hoc search. Finally, dynamically generated pages of the truly invisible web are a target for ad-hoc search, if the author of the search program has an intuition about admissible query strings to be used for web page generation.

- Moreover, index generation only pays off if the number of queries issued on a collection of documents is relatively large. In the case that crawling is performed in order to just answer a particular query, index generation would be breaking a butterfly on the wheel. For use-cases in which the desired information can be restricted to a small subset of all web-pages, it is thus more efficient and may also be indispensable to use ad-hoc search instead of ordinary search engines.

In this deliverable we present several examples for ad-hoc searches **??**, give an overview about related work both for writing crawlers and for expressive query languages 2.4, establish language features which are required for an easy-to-use and declarative authoring of ad-hoc search programs, and elaborate on a use-case for solving the single-source shortest path problem at the aid of a crawler authored in the declarative language Xcerpt^{RDF} **??**.

2.3 Ad-hoc-search Use Cases

Solving the Single Source Shortest Path Problem for FOAF documents For the last years, social network services such as MySpace, Facebook and Orkut have gained widespread attention of the online community. Probably the most open way of building a social network is the publication of Friend-Of-A-Friend (FOAF) documents. A service provided by most social networks is the presentation of a shortest path $p_1, p_2, \ldots p_n$ between two given persons p_1 and p_n such that for all $1 \leq i \leq n-1$ the person p_i knows the person p_{i+1}. A slight simplification of this problem is to only display the degree of knowledge, i.e. the length of such a shortest path. In this deliverable, we refer to the simplified version as the *Single Source Shortest Path Problem (SSSPP) for Foaf Documents*.

The SSSPP for FOAF documents is a very well suited application for ad-hoc search, since the amount of pages needed to answer a query is limited. Consider for example the issue of finding all friends of degree 3 for some given person p. Usually, foaf documents only contain friendship relationships emanating from the publisher of the document, thus for answering the above query it suffices to perform a crawl of depth 3 starting out from the foaf document of p. A solution to this use-case is presented in more detail in Section 2.5.

Scheduling a Soccer Match within a Small Town Based on ICalender Data iCalendar is a standardized format for saving calendar data of personal calendars adopted by the IETF[1] in RFC 2445. It is very widely adopted by both proprietary applications such as Google Calendar, Microsoft Exchange Server, Lotus Notes, Novell Groupwise, FaceBook, and open source applications such as Mozilla Calendar, Plone, Korganizer and Open-XChange. iCalendar data can be easily transformed to RDF format by publicly available scripts such as the one provided by Elias Torres[2] or Dan Connlly and Libby Miller[3].

Provided that a reasonable amount of calendar data is publicly availabe, an ad-hoc search program could be written to crawl FOAF profiles of people within a certain geographical area and query the associated calendar data in order to find a suitable date and participants for some kind of reunion such as a soccer match, or a village fair. Also for this use-case the crawling strategy is easily determined, since only profiles and calendars pertaining to the region in question are to be crawled.

Finding Descriptions of a Project for a given Open Source Developer

2.4 Related work

Ad-hoc search involves both crawling of (Semantic) Web documents as well as sophisticated reasoning based on an expressive query language. Therefore related work can be separated into crawling algorithms employed by search engines, and query languages for Web and Semantic Web content.

- The fish search algorithm [19] is one of the oldest algorithms employed by focused crawlers. Starting out from a seed URL, it builds a priority list of URLs to be explored using a search query and a scoring function which estimates the relevancy of documents relative to the search query. At each step, the first element from the priority list is processed, and the links are extracted from the document. If the document is relevant to the search query, the depth of the links is set to some predefined positive integer. If not, the depth of the links is set to one less than that of the parent. Links with depth zero are not processed. Two parameters α and $width$ are used to determine the location where links are inserted in the priority list.

- The shark search algorithm [31] extends the fish search algorithm in several ways. While the scoring function of the fish search algorithm is binary (return values 0 or 1 only), the shark search algorithm employs a fuzzy similarity measure between 0 and 1. Additionally, the shark search algorithm uses meta-information contained in the links to documents and in their close textual context in order to assess the relevancy of the linked documents. Both the fish search and the shark search algorithm are crawling algorithms for the conventional web. With the embarkment of *semantic web* data, focused crawlers need not rely on scoring functions or similarity measures,

[1] Internet Engineering Task Force

[2] available at `torrez.us/ics2rdf/`

[3] available at `www.w3.org/2002/12/cal/ical2rdf.pl`

but may simply chose to only follow those links between documents which are identified by some chosen RDF predicate, which is the approach taken by ad-hoc searches in this deliverable. A combination of both techniques may prove to be beneficial.

- In [39] Gautam and Srinivasan point out the necessity for finding a good traid-of between exploration and exploitation of Web content. A web crawler may either chose to visit only those pages that have the highest relevancy score, thus being very exploitative of the information about the relevancy. Or it may be explorative in the sense that it prefers a page with a lower score, and thereby avoid being trapped in a local optimum of the search graph. Using a variant of Best-First crawling called *Best-N-First* crawling, where the parameter N controls the explorativity/exploitativity (the higher N is, the more explorative) of the crawler. The findings of [39] are among others that (a) explorative crawlers have an advantage over exploitative crawlers in finding the most relevant pages in the long run, but initially perform worse, and (b) blind exploration yields poor results when starting from neighbors of relevant pages, but performs alright when starting from relevant pages themselves.

  In the case of ad-hoc search on the Semantic Web, the issue of exploration versus exploitation can, however, be in many cases satisfactory resolved in favor of either of the two strategies. When compared to ordinary web data, Semantic Web Data is more precise in the sense that links between pieces of information or between resources are typed, and in that these types can be uniquely determined by their URI. As a consequence, crawlers of the Semantic Web often need not employ a scoring component, but can decide to consider only a specific fragment of all links between documents, in which case exploration is no longer possible. In a very similar way, exploitation can become the method of choice, if the depth of the crawl is bounded by a search parameter (e.g. find all friends of degree no greater than 3).

2.5 Answering the SSSPP for Foaf Profiles with Xcerpt^{RDF}

Traditionally, an ad-hoc search would be processed using a scripting language such as Perl, Python or Ruby or at the aid of a general purpose programming language. While these languages often supply libraries for the access to Web formats – most prominently XML and RDF –, none of these languages has been explicitly designed for native support of XML and RDF. Xcerpt^{RDF} in contrast, has been conceptualized to support these formats by providing the following equipment:

- structured query and construct terms for both XML and RDF
- incompleteness constructs for both XML and RDF data
- support for Web-Crawling by rule chaining
- syntactic sugar for RDF specificities such as blank nodes, RDF containers and collections, reification and concise bounded descriptions
- negation as failure
- built-in understanding of some RDFS and OWL vocabulary such as `rdfs:subClassOf`, `rdfs:subPropertyOf`, `owl:equivalentClass`, `owl:equivalentProperty`, `owl:inverseFunctionalProperty`. These properties are implemented by a translation to simple Xcerpt^{RDF} rules.

2.5.1 Crawling FOAF Documents to Answer SSSPP Problems

Listings 2.5_1 and 2.5_2 show how the single source shortest path problem for `foaf:knows` relationships distributed over multiple a priori unknown documents on the Web is implemented in Xcerpt^{RDF} .

In lines 1 to 3, namespace prefixes are declared to allow concise authoring of Xcerpt^{RDF} query, construct and data terms.

Line 5 serves to declare the predicate `foaf:mbox-sha1sum`, which relates an instance of the class `foaf:Person` to the sha1[4] hash value of its email address, as an inverse functional property. The OWL Language Reference [20] states that if an RDF property is defined to be inverse functional, the object of a statement uniquely determines its subject. Hence, if the Xcerpt^{RDF} program discovers two RDF statements with the property `foaf:mbox-sha1sum`, and the same sha1 hash value as objects, it can deduce that the subjects of the statements denote the same real-world entity – in this case the same person. Having realized this, Xcerpt^{RDF} replaces all occurrences of one subject by the other.

In line 7 goals having the outermost label `Acquaintance` are declared to be tabled terms, i.e. answers to these query terms are memoized in a hierarchical data structure, and when a new subgoal with predicate `Acquaintance` is encountered, this data structure is searched for already subsuming goals. Tabling in Xcerpt is realized in a similar way as in [47], but due to the expressivity of Xcerpt query terms, the algorithm for deciding subsumption between query terms is more complex – it is a generalized version of simulation unification[11] –, and a special hierarchical data structure is needed for recording results of the subsumption tests. The algorithm for deciding subsumption is presented in section **??**, and the data structure is introduced in section **??**.

In lines 9 to 11 the seed for the crawler is given as an Xcerpt data term, also called a fact in logic programming terminology. The url `http://eg.org/persons/anna` is used as the starting point for transitively finding other relevant foaf documents until a maximum depth of four has been reached.

The actual crawling is realized by the two rules in Listing 2.5_1, the first one discovering new relevant documents by following the `rdfs:seeAlso` links, the second one asserting the RDF graphs found in these documents to the in memory database. It is important to note that the evaluation of these two rules – no matter if a backward chaining or a forward chaining strategy is applied – is intertwined. In a forward chaining evaluation, the first rule is applied to import the RDF graph from the seed url to local memory. Then the second rule is used to find additional documents from the initial RDF graph following `rdfs:seeAlso` predicates. Afterwards the RDF graphs found at these new locations are imported to local memory by the first rule, and so on.

RDF/XML documents being both in XML format, but supplying RDF data can be queried syntactically and semantically in Xcerpt^{RDF} – in other words, such documents can be seen as XML trees or as RDF graphs. The programmer simply specifies the desired view by giving the keyword "RDF" or "XML" in the resource specification of a query term (see line 19).

RDF and XML are both graph structured, which means that similar query constructs, such as incompleteness in depth, incompleteness in breadth, term variables, optional subterms are beneficial for its treatment[10]. There are, however, inherent differences between RDF and XML, most prominently that RDF is treated as knowledge, whereas XML is treated as data. One philosophy behind RDF is that "any person can state anything about any resource", demanding that the labels in an RDF graph are global identifiers, such that statements in distinct RDF documents referencing the same URI are considered to provide information about the same real world entity. On the other hand, the same qualified name may appear in different XML documents that have nothing to do with each other. This is why

[4] `http://en.wikipedia.org/wiki/SHA1`

the local memory of an Xcerpt processor may include many independent XML data terms, but only one default RDF graph.

The crawler program in Listing 2.5_1 successively binds the entire RDF graphs from the encountered RDF documents to the term variable `Graph` (line 20) and subsequently merges these graphs with the default graph in memory (line 14). *Merging* of RDF graphs means computing the union of the sets of RDF triples in the graphs to be merged, thereby removing duplicate triples and standardizing apart blank node identifiers. (See [30] for a detailed discussion about *union semantics* versus *merge semantics* of RDF graphs). Support for named graphs[13] in Xcerpt^{RDF} is being considered.

```
1  declare ns-prefix foaf="http://xmlns.com/foaf/0.1/";
   declare ns-prefix rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#";
3  declare ns-prefix rdfs="http://www.w3.org/2000/01/rdf-schema#";

5  declare inverseFunctionalProperty foaf:mbox-sha1sum;

7  table Acquaintance;

9  CONSTRUCT
     SeeAlso [ "http://eg.org/persons/anna/foaf.xrdf", 0 ]
11 END

13 CONSTRUCT
     var Graph
15 FROM
     and (
17     SeeAlso[[ var SeeAlso ]],
       in {
19       resource { var SeeAlso, rdf },
         var Graph as /.*/{{ }}
21     }
     )
23 END

25 CONSTRUCT
     SeeAlso[ var SeeAlso, var N + 1 ]
27 FROM
     and (
29     /.*/{{
         rdf:type--> foaf:Person,
31       rdfs:seeAlso--> var SeeAlso
       }},
33     SeeAlso[ var SeeAlso1, var N <= 3 ]
     )
35 END
```

Listing 2.5_1: Finding relevant information for answering an ad-hoc search query with an Xcerpt^{RDF} crawler

2.5.2 Formulation of SSSPP as an Unstratified Logic Program

Finding the relevant information is only half a solution to answer an ad-hoc search query. The second part is to derive new knowledge from the gathered Semantic Web information by applying deductive rules, and to find all ground instances of the query which are implied by the given data and rules. In many cases this second step requires the query language of choice to provide recursion, negation as failure and rich query terms.

The second part of our ad-hoc search program implements the single source shortest path problem[53] for the subgraph of the RDF graph constructed in Listing 2.5_1 consisting only of its `foaf:knows` relationships and the nodes they connect. It turns out that implementing SSSPP in a logic programming language such as Xcerpt requires negation as failure, since one must ensure, that only the shortest paths – i.e. *not* any path that is longer – are considered.

In lines 36 to 38 the seed URI representing Anna is encoded as an Xcerpt fact, and it is asserted that Anna knows herself over zero other persons. In order to keep track of the length of a path from the seed URI to other URIs representing acquaintances, all terms with the outermost label `Acquaintance` are binary, the first subterm being the URI, and the second one an integer value representing its distance to the seed.

The rule in lines 40 to 48 derives acquaintances of degree $n+1$ from acquaintances of degree n. In the body of the rule a conjunction is used to query both the RDF graph collected before and ordinary Xcerpt data terms, which can be considered as XML data. The third conjunct of the query (line 46) is a negated literal and serves to filter out all those bindings for the variable `Aq`, for which a shorter path can be found. This third conjunct uses the syntax `< N + 1` to specify that this goal shall only unify with dataterms whose second subterm can be interpreted as an integer value smaller than `N + 1`. This extension of simulation unfication can be integrated into the framework of [44] by the addition of the following decomposition rule:

$$\frac{(< N) \preceq M}{N < M}$$

Alternatively, the same effect could be achieved by rewriting the subterm `not(Acquaintance[var Aq, < N + 1])` by the subgoal `not(closer[var Aq, N + 1])` and by the inclusion of the following rule defining the terms `closer`.

```
CONSTRUCT
  closer[ var Aq, var N ]
FROM
  Acquaintance[ var Aq, var Distance ]
WHERE
  var Distance < var N
END
```

```
36 CONSTRUCT
     Acquaintance[ http://eg.org/persons/anna, 0 ]
38 END

40 CONSTRUCT
     Acquaintance[ var Aq, var N + 1 > 0 ]
42 FROM
     and (
44     var Person{{ foaf:knows--> var Aq }},
```

```
    Acquaintance[ var Person, N ],
46  not ( Acquaintance[ var Aq, < N + 1 ] )
  )
48 END
```

Listing 2.5_2: Answering an ad-hoc search query by rule chaining and negation as failure

In many cases not only the degree of friendship is of interest, but also the shortest path from one person to another following `foaf:knows` relationships. As Prolog programmers know, it is easy to adapt the above program to also return this information.

Due the recursive rule in Listing 2.5_2, the semantics and evaluation of our program is non-trivial, since it is unstratifiable both according to the definition in [44] and according to the following definition, which is very close to the ordinary definition of stratification of logic programs:

Definition 9 (Stratification of an Xcerpt Program) *A* Stratification *of an Xcerpt program P consisting of a set of rules $R = r_1, \ldots, r_n$ is a partitioning of R into layers $L_1, \ldots, L_k$ such that the following conditions are fulfilled:*

- *Rules deriving facts that share the same outermost label are in the same layer.*
- *If a rule r_i positively depends on a rule r_j, and r_j is in layer L_m, then r_i is in a layer L_o with $o \geq m$. r_i positively depends on r_j, iff there is a* positive *subgoal in the body of r_i that simulation unifies with the head of r_j.*
- *If a rule r_i negatively depends on a rule r_j, and r_j is in layer L_m, then r_i is in a layer L_o with $o > m$. r_i negatively depends on r_j, iff there is a* negative *subgoal in the body of r_i that simulation unifies with the head of r_j.*

Definition 10 (Stratifiable) *An Xcerpt program P is stratifiable, iff there exists a valid stratification for it.*

Definitions 9 and 10 are in line with the definitions of stratification and stratifiability for ordinary logic and datalog programs. See [9] for an overview over various semantics and evaluation possibilities for unstratifiable logic programs.

While the semantics for stratified logic programs is given by an iterative fixpoint semantics, and is always two-valued, unstratified programs may not always have a unique two-valued model. The well-founded semantics is a three valued semantics in which ground atoms may either be true, false or of unknown truth value. Well-founded models can be computed for arbitrary logic programs with negation, and the models of stratified programs are always total in the sense that the truth values of all ground atoms are either true or false, never unknown. Many unstratified programs have a two-valued well-founded model, and it turns out that there is also a unique two valued model for the program in Listing 2.5_1 and 2.5_2 under the well-founded semantics. To see this, a new concept, called *local stratification* (see Definition 14) is introduced, and a local stratification for our program is given. Gelder, Ross and Schlipf have shown in [51] that all locally stratified programs have a two valued well-founded model.

Algorithms for computing the well-founded semantics have been investigated, and one of the most popular ones is SLG-resolution [15].

Definition 11 (Well-founded Relation ([54])) *A binary relation R is well-founded on a class X, if and only if every non-empty subset of X has a minimal element with respect to R.*

Definition 12 (Dependency Graph ([42])) *Let P be a ground program consisting of a set of rules $\{r_1,\ldots,r_n\}$. The set of vertices V of the dependency graph $G_P = (V,E,L)$ of P is the set of ground atoms that appear either positively or negatively in at least one of the rules of P. $E \subseteq (V \times V)$ is the set of edges of the dependency graph and $L : E \rightarrow \{1,-1\}$ is a labeling function on the set of edges, giving each edge either a positive or a negative value. For a rule $r_i : h \leftarrow l_1,\ldots l_k$ with head h and body literals $l_j, 1 \leq j \leq k$, k edges are constructed in the dependency graph G_P. If l_p is a positive body literal in r_i, then $(l_p,h) \in E$ with $L((l_p,h)) = 1$. If $l_n := not(l'_n)$ is a negative literal in the body of r_i then $(l'_n,h) \in E$ with $L((l'_n,h)) = -1$.*

Definition 13 (Negative Dependency Relation) *Let P be a general logic program, ground(P) its Herbrand instantiation, $G_{ground(P)} = (V,E,L)$ the dependency graph of ground(P) and $A,B \in V$.*

A depends negatively *on B, iff there is a path from B to A in $G_{ground(P)}$ involving at least one negative edge.*

Definition 14 (Local Stratification ([17])) *A logic program P is locally stratified if and only if the negative dependeny relation of P is well-founded.*

An equivalent definition of local stratification is given in [51]:

Definition 15 (Local Stratification[51]) *A logic program P is locally stratified, if one can assign an ordinal rank to each atom in the Herbrand base of P such that no atom depends upon an atom of greater rank or depends negatively in one of equal or greater rank in any instantiated rule.*

Theorem 2 (Well-founded models of Locally Stratified Programs [51]) *If a program P is locally stratified, then it has a well-founded model, which is identical to its perfect model.*

Definition 16 transfers the concept of local stratification to Xcerpt programs. It differs from the definition of local stratification for ordinary logic programs only in the first two conditions, which are due to the differentiation between ground query terms and ground data terms in Xcerpt. While a ground subgoal in logic programming only unifies with the term which is syntactically equivalent, there may be multiple Xcerpt data terms that a ground Xcerpt query term simulation unifies with – for example the ground query term `Acquaintance[ eg:bob, < 3 ]` simulation unifies with the ground facts `Acquaintance[ eg:bob, 0 ]`, `Acquaintance[ eg:bob, 1 ]`, and `Acquaintance[ eg:bob, 2 ]`. To ensure that a the goal $g :=$ `Acquaintance[ eg:bob, < 3 ]` to these atoms, they must be defined in a layer that is equal to or lower than the layer of g.

Definition 16 (Local Stratification of an Xcerpt Program) *A local stratification of an Xcerpt program P is a partitioning of the ground terms occurring in rule instantiations into a set of layers $\{L_1,\ldots L_k\}$, such that the following holds:*

- *if a positive ground query term q simulates into a data term d, then q is in a higher, or the same level as d.*
- *a negative ground query term $q = notq'$ with q' simulating into a data term d is in a strictly higher level thand d.*
- *if a ground query term q from layer L_i appears positively within the body of an instantiated rule r_j, then the instantiated head of the rule is in L_i or in a higher level.*
- *if a ground query term q from layer L_i appears negatively within the body of an instantiated rule r_j, then the instantiated head of the rule is in a higher lever than L_i.*

A local stratification for Program 2.5_2 would thus be the stratification $S := \{L_1, L_2, \ldots\}$ with

- $L_1 := \{Ac[\text{anna}, < 0], Ac[\text{bob}, < 0], Ac[\text{chuck}, < 0], Ac[\text{anna}, 0], Ac[\text{bob}, 0],$
  $Ac[\text{chuck}, 0]\}$
- $L_2 := \{Ac[\text{anna}, < 1], Ac[\text{bob}, < 1], Ac[\text{chuck}, < 1], Ac[\text{anna}, 1], Ac[\text{bob}, 1],$
  $Ac[\text{chuck}, 1]\}$
- $L_3 := \{Ac[\text{anna}, < 2], Ac[\text{bob}, < 2], Ac[\text{chuck}, < 2], Ac[\text{anna}, 2], Ac[\text{bob}, 2],$
  $Ac[\text{chuck}, 2]\}$
- ...

2.5.3 Efficient Evaluation of Xcerpt^{RDF} Multi-Rule Programs

The program in Listings 2.5_1 and 2.5_2 can be queried in many different ways, e.g:

- Find all acquaintances of Anna and their degrees of friendship:
  `Acquaintance{ var Person, var Degree }`
- Find all acquaintances of Anna with a friendship degree of four:
  `Acquaintance{ var Person, 4 }`
- Over how many other people does Anna know Bob?
  `Acquaintance{ http://eg.org/bob, var Degree }`
- Does Anna know Bob over three other people?
  `Acquaintance{ http://eg.org/bob, 3 }`
- Is there a path from Anna to Bob following `foaf:knows` relationships?
  `Acquaintance{{ http://eg.org/persons/bob }}`

All of the above queries could be evaluated in a forward or a backward chaining manner. The advantages of both methods have been discussed in detail, the most prominent advantage of backward chaining being its goal-directedness and the most prominent advantage of forward-chaining being its closeness to the fixpoint operator for logic programs, and in some cases better termination properties. Many efforts have been undertaken to combine the best of both worlds [15, 5, 22, 48]. The results of these methods being either extensions of backward chaining algorithms with memoization or forward chaining algorithms enhanced with goal directedness through rule-rewriting, have come astonishingly close to each other and are in many cases even equivalent[8].

In this section we describe a backward chaining algorithm with subsumption checking and memoization similar to SLG resolution for evaluating the program in Listing 2.5_2, starting out with the query `Ac[eg:chuck, var Distance]`[5] assuming that the RDF graph in Listing 2.5_3 has been collected by the crawler from Listing 2.5_1. We then argue that the algorithm works for any locally stratified Xcerpt^{RDF} program.

[5] `Ac` is an abbreviation for the predicate `Acquaintance` and the namespace prefix `eg` is assumed to be bound to `http://eg.org/persons/`

```
@prefix eg: <http://eg.org/persons/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<eg:anna> <foaf:knows> <eg:bob>.
<eg:anna> <foaf:knows> <eg:chuck>.
<eg:bob> <foaf:knows> <eg:chuck>.
<eg:chuck> <foaf:knows> <eg:anna>.
```

Listing 2.5_3: RDF Data Collected by the Crawler in Listing 2.5_1

Figure 2.1 depicts the resolution steps necessary to derive that anna knows chuck directly – i.e. over only one `foaf:knows` relationship. In the first step, the initial query `Ac[eg:chuck, var Distance]` is issued and labeled with the ordinal 1. In the second step, the initial query is resolved against the recursive rule from Listing 2.5_2, and the 3 subqueries labeled 2, 3, and 4 are generated just as with ordinary resolution. In the third step, subquery 2 is resolved against the data, yielding the binding `eg:bob` for the variable `Person`. Subsequently (step 4) the recursive rule is applied again in order to resolve subgoal 3, and the binding `eg:anna` is found for the variable `Person1` (step 5). Step 6 and 7 are analogous to step 4 and 5. In Step 8, we find that subgoal 9 is just a variant subgoal of subgoal 1, and that continuing the resolution of subgoal 9 would result in an infinite branch of the resolution tree. Thus, subgoal 9 is labeled as a consuming subgoal attached to the generating subgoal 1. Since 1 has not found any solutions yet, the evaluation continues with subgoal 10. Subgoals 10, 7 and 4 are negated literals whose positive counterparts are variants of the goals 6, 3, and 1, respectively, neither of which has found a solution yet. Therefore the subgoals 10, 7, and 4 are suspended until their variants find solutions, or are completed. Backtracking to subgoal 8, we find that it cannot be instantiated in any other way and therefore label it as completed (step 12). Backtracking to subgoal 6, we find that it can be instantiated by the fact `Ac[eg:anna, 0]` yielding the binding 2 for variable `Distance` (step 13), and the solution `Ac[eg:anna, 0]` for subgoal 6.

Subgoals 7 and 4 are found to be variants of subgoals 3 and 1, respectively, and are thus subspended until these subgoals are completed or find a solution – in which case the subgoals 7 and 4 fail. Backtracking further, we find that goal 5 cannot be instantiated in any other way and is thus completed (step 16), and the same holds for subgoal 3 (step 17).

2.6 Decidability of Xcerpt Query Term Subsumption

Theorem 3 (Subsumption is monotone) *Let q be a query term. The following transformations can be applied to q such that the transformed version $t(q)$ of q is subsumed by q.*

- *if q has incomplete subterm specification, it may be transformed to the analogous query term with complete subterm specification.*

$$\frac{a\{\{q_1,\ldots,q_n\}\}}{a\{q_1,\ldots,q_n\}} \tag{2.1}$$

$$\frac{a[[q_1,\ldots,q_n]]}{a[q_1,\ldots,q_n]} \tag{2.2}$$

- *if q has unordered subterm specification, it may be transformed to the analogous query term with ordered subterm specification.*

$$\frac{a\{\{q_1,\ldots,q_n\}\}}{a[[q_1,\ldots,q_n]]} \tag{2.3}$$

$$\frac{a\{q_1,\ldots,q_n\}}{a[q_1,\ldots,q_n]} \tag{2.4}$$

- *if q is of the form desc q' then the descendant*

construct may be eliminated.

$$\frac{desc\ q}{q} \tag{2.5}$$

- *if q is of the form 'desc q′', then it may be wrapped inside an arbitrary label l.*

$$\frac{desc\ q}{l\{\{desc\ q\}\}} \tag{2.6}$$

- *if q has incomplete-unordered subterm specification, then a fresh variable X may be appended to the end of the subterm list, provided that X does not appear in q.*

$$X \notin Vars(\{q_1,\ldots,q_n\}) \Rightarrow \frac{a\{\{q_1,\ldots,q_n\}\},}{a\{\{q_1,\ldots,q_n,varX\}\}} \tag{2.7}$$

- *if q has incomplete-ordered subterm specification then a fresh variable may be* inserted *anywhere in the subterm list, provided that X does not occur in q.*

$$X \notin Vars(q_1,\ldots,q_n) \Rightarrow \frac{a[[q_1,\ldots,q_n]],}{a[[q_1,\ldots,q_i,varX,q_{i+1},\ldots,q_n]]} \tag{2.8}$$

- *if q has unordered subterm specification, then the subterms of q may be arbitrarily permuted. The rule for incomplete-unordered subterm specification is given below, the one for complete-unordered subterm specification is analogous, but omitted for the sake of brevity.*

$$\pi \in Perms(\{1,\ldots,n\}) \Rightarrow \frac{a\{\{q_1,\ \ldots,\ q_n\}\}}{a\{\{q_{\pi(1)},\ \ldots,\ q_{\pi(n)}\}\}} \tag{2.9}$$

- *if q contains a variable* `var X`*, which occurs in q at least once in a positive context (i.e. not within the scope of a without) then all occurrences of* `var X` *may be substituted by another Xcerpt query term.*

$$X \in PV(q), t \in QTerms \Rightarrow \frac{q}{q\{X \mapsto t\}} \tag{2.10}$$

Notice that if a variable appears within q only in a negative context (i.e. within the scope of a `without`*), the variable cannot be substituted by an arbitrary term to yield a transformed term that is subsumed by q. The query terms* `a{{ without var X }}` *and* `a{{ without b{ } }}` *together with the data term* `a{ c }` *illustrate this characteristic of the subsumption relationship. For further discussion of substitution of variables in a negative context see Example 5.*

- *if q has an optional subterm q_i = optional q', then this subterm may be omitted from q, which is formalized by the following rule:* [6]

$$q_i \text{ is optional} \Rightarrow \frac{a\{\{q_1,\ldots,q_i,\ldots,q_n\}\}}{a\{\{q_1,\ldots,q_{i-1},q_{i+1},\ldots q_n\}\}} \tag{2.11}$$

- *if q has a subterm q_i, then q_i may be transformed by any of the transformations in this list except for Equation 2.10 to the term $t(q_i)$, and this transformed version may be substituted at the place of q_i in q, as the following rule shows:* [7] [8]

$$\frac{q_i}{t(q_i)} \Rightarrow \frac{a\{\{q_1,\ldots,q_n\}\}}{a\{\{q_1,\ldots,q_{i-1},t(q_i),q_{i+1},\ldots q_n\}\}} \tag{2.12}$$

[6] The respective rule for incomplete-ordered subterm specification is omitted for the sake of brevity. Negated or optional subterms within terms with complete subterm specification are not allowed.

[7] The respective rules for complete-unordered subterm specification, incomplete-ordered subterm specification and complete-ordered subterm specification are omitted for the sake of brevity.

[8] The exclusion of Equation 2.10 ensures that variable substitutions are only applied to entire query terms and not to subterms. Otherwise the same variable might be substituted by different terms in different subterms.

- *if the label of q is a regular expression e, this regular expression may be replaced by any label that matches with e, or any other regular expression e' which is subsumed by e (see Definition **??**).*[7]

$$\frac{e, e' \in RE, e \text{ subsumes } e' \Rightarrow e\{\{q_1, \ldots, q_n\}\}}{e'\{\{q_1, \ldots, q_n\}\}} \quad (2.13)$$

- *if q contains a negative subterm q_i = without r and r' is a query term such that $t(r') = r$ (i.e. r' subsumes r) for some transformation step t, then q_i can be replaced by $q_i' :=$ without r'.*

$$\frac{(q_i = \text{without } r) \wedge \frac{r'}{r} \wedge (q_i' = \text{without } r') \Rightarrow a\{\{q_1, \ldots, q_i, \ldots, q_n\}\}}{a\{\{q_1, \ldots, q_{i-1}, q_i', q_{i+1}, \ldots q_n\}\}} \quad (2.14)$$

The proof of the above theorem is straight-forward since each of the transformation steps can be shown independently of the others. For all of the transformations, inverse transformation steps t^{-1} can be defined, and obviously for any query term q it holds that $t^{-1}(q)$ subsumes q.

As shown above, the substitution of a variable X in a negative context of a query term q by a query term t, which is not a variable, results in a query term $q' := q[X \mapsto t]$ which is in fact more general than q. In other words $q[X \mapsto t]$ subsumes q for any query term q if X only appears within a negative context in q. On the other hand, if X only appears in a positive context within q, then q' is less general – i.e. q subsumes q'. But what about the case of X appearing both in a positive and a negative context within q? Consider the following example:

Lemma 5 *Let $q :=$ `a{{ var X, without b{{ var X }} }}`. One is tempted to think that substituting X by `c[]` makes the first subterm of q less general, but the second subterm of q more general. In fact, a subterm `b[ c ]` within a data term would cause the subterm `without b{{ var X }} }}` of q to fail, but the respective subterm of q' to succeed, suggesting that there is a data term that simulation unifies with q', but not with q, meaning that q does not subsume q'. However, there is no such data term, which is due to the fact that the second occurrence of `X` within q is only a consuming occurrence. When this part of the query term is evaluated, the variable `X` is already bound.*

Definition 17 (Incompatible Variable Restrictions) *Let $q_1 :=$ var X -> t_1 and $q_2 :=$ var X -> t_2 be two query terms. We say that the variable restrictions for X in q_1 and q_2 are incompatible iff*

- *the labels of t_1 and t_2 are both qualified names and distinct.*
- *the labels of t_1 and t_2 are both regular expressions and the Intersection of the languages they define is empty.*
- *one of t_1 and t_2 is a qualified name n and the other is a regular expression e, and n is not in the language defined by e.*

Theorem 4 (Subsumption of Query Terms with Incompatible Variable Restrictions) *Let q be a query term that contains two subterms with incompatible variable restrictions for a variable X. Then there is no data term t such that t simulates with q. Hence q is subsumed by any other query term.*

Theorem 5 (Subsumption by transformation) *Let q_1 and q_2 be two query terms such that q_1 subsumes q_2 and q_2 does not have any incompatible variable restrictions. Then q_1 can be transformed into q_2 by a sequence of subsumption monotone query term transformations listed in Theorem 3.*

Proof 1 *We must distinguish two cases:*

- q_1 *and* q_2 *are subsumption equivalent (i.e. they subsume each other)*
- q_1 *strictly subsumes* q_2

The first case is the easier one. If q_1 *and* q_2 *are subsumption equivalent, then there is no data term t, such that t simulates with one, but not the other. Hence* q_1 *and* q_2 *are merely syntactical variants of each other. Then* q_1 *can be transformed into* q_2 *by consistent renaming of variables (Equation 2.12), by omitting optional subterms (Equation 2.11), and by reordering sibling terms within subterms of q that have unordered subterm specification (Equation 2.9).*

The second case is somewhat trickier. Let's assume that q_1 *subsumes* q_2*, but not the other way around. Then there is a data term d, such that* q_1 *simulates into d, but* q_2 *does not. In the following list, we give all possible reasons for d simulating into* q_1 *but not into* q_2*. For each of these cases, we give a sequence of subsumption monotone transformations* $t_1, \ldots t_n$ *from theorem 3, such that* $q_1' := t_n(t_{n-1}(\ldots t_1(q_1)))$ *does not simulate into d. By theorem 3, and by the transitivity of the subsumption relationship,* q_1' *still subsumes* q_2*. Hence by considering d and by applying the transformations, we have brought* q_1 *"closer" to* q_2*. If* q_1' *is still more general than* q_2*, then one can find another dataterm* d' *that simulates with* q_1'*, but not with* q_2*, and another sequence of transformations to be applied can be deduced from this theorem. This process can be repreated until* q_1 *has been transformed to a simulation equivalent version of* q_2*.*

- *The label* l_d *of d matches with the label* l_1 *of* q_1*, but not with the label* l_2 *of* q_2*. In this case substitute* l_2 *for* l_1 *in* q_1*. Note that since* q_1 *subsumes* q_2 *and* q_2 *does not have incompatible variable restrictions,* l_1 *must subsume* l_2*, and therefore the substitution of* l_2 *for* l_1 *in* q_1 *is a subsumption monotone query term transformation.*

  Lemma 6 *Let* $q_1 := /.*/\{\{a,c\}\}$*,* $q2 := html\{\{a,c,d\}\}$ *and* $d := xml\{a,b,c,d\}$*. Then transform* q_1 *to* $q_1' := html\{\{a,c\}\}$*.*

- *The subterm specification of d is unordered, the one of* q_1 *is unordered, but the one of* q_2 *is ordered. Each of* q_1*,* q_2 *and d has more than one subterm. In this case change the subterm specification of* q_1 *to ordered.*

  Lemma 7 *Let* $q_1 := a\{\{b,c\}\}$*,* $q_2 := a[[b,c,d]]$*,* $d := a\{b,c,d\}$*. Transform* q_1 *to* $q_1' := a[[b,c]]$*.*

- *The number of direct subterms of d is greater than the number of direct subterms of* q_2*, and* q_2 *has breadth-complete subterm specification.* q_1 *is breadth-incomplete. Then add fresh variables to the subterms of* q_1 *until* q_1 *has the same number of subterms as* q_2 *and change the subterm specification of* q_1 *to complete.*

  Lemma 8 *Let* $q_1 := a\{\{b,c\}\}$*,* $q_2 := a\{b,c,d\}$*,* $d := a\{b,c,d,e\}$*. Transform* q_1 *to* $q_1' := a\{b,c,varX\}$*.*

- q_2 *has a direct subterm* q_{2_s} *for which there is no direct subterm* d_s *within d such that* q_{2_s} *simulates into* d_s*. Since* q_1 *subsumes* q_2*, either* q_1 *has incomplete subterm specification or there is a subterm* q_{1_s} *that subsumes* q_{2_s}*. If there is such a subterm* q_{2_s}*, replace it by* q_{1_s} *(this can be achieved by multiple application of Equation 2.12), otherwise insert a fresh variable into* q_1 *(Equation 2.7 or 2.8) and replace this fresh variable by* q_{1_s} *(Equation 2.10).*

- *In fact, the preceding case is a special version of a more general case, that is comprised within the following case:*

  There is no injective mapping from the children of q_2 to the children of d such that q_{2_i} is simulates into $\pi(q_{2_i})$.

  Since q_1 subsumes q_2 there is an injective mapping from the subterms of q_1 to the subterms of q_2, such that $q_{1_i} \preceq \pi(q_{1_i})$ for all subterms q_{1_i} of q_1.[9].

  According to Theorem 3 it is allowed to replace q_{1_i} in q_1 by $\pi(q_{1_i})$ (Equation 2.12). Moreover, if there is a subterm q_{2_i} in q_2 which is not in the range of π, this subterm may be added to the subtermlist of q_1 (Equations 2.7 and 2.10).

- *q_2 has incomplete subterm specification and a negative subterm q_{2_n} = without q'_2 that simulates with some subterm d_s of d. Additionally, there is an injective mapping π*[10] *from the positive subterms of q_2 to the subterms of d, such that d_s is not in the range of π, and such that q_{2_i} simulates into $\pi(q_{2_i})$ for all subterms q_{2_i} of q_2. Moreover there is a mapping μ from the positive subterms of q_1 to the positive subterms of q_2 such that for all subterms q_{1_i} of q_1 it holds that q_{1_i} subsumes $\mu(q_{1_i})$. However, there is no negative subterm q_{1_n} of q_1 that simulates with d_s.*

  We distinguish two cases:

  - *there is a negative subterm q_{1_n} = without q'_1 in q_1 such that q'_2 subsumes q'_1. Then replace q'_1 by q'_2 in q_1.*
  - *there is no such subterm in q_1. Then insert a fresh variable X into the subtermlist of q_1 by Equation 2.7 or 2.8 and replace X by q_{2_n} (Equation 2.12).*

Infinite chains of subsumption decreasing query terms can be constructed starting out from any query term that is a variable, incomplete in breadth, incomplete in depth, or a subterm which is a variable, incomplete in breadth or in depth as the following example shows.

Lemma 9 (Infinite Sequences of Subsumption Decreasing Query Terms)

- $a\{\{\ \}\}, a\{\{b\}\}, a\{\{b,b\}\}, \ldots$
- $a\{varX\}, a\{b\{\{\ \}\}, a\{b\{\{c\}\}, a\{b\{\{c,c\}\}, \ldots$
- *desc* $a\{\ \}$, $b\{$*desc* $a\{\ \}\}, b\{b\{$*desc* $a\{\ \}\}\}, \ldots$

For any query term q that contains a negative subterm with a breadth-incomplete subterm specification, with a depth-incomplete subterm, or with a variable, one can find an infinite sequence of query terms $q_1, q_2, \ldots$ such that q_i is strictly subsumed by q_{i+1} for all $i \in \mathbb{N}$ and $q_1 = q$.

Lemma 10 (Infinite Sequences of Subsumption Increasing Query Terms)

- *Let* $q := a\{\{$*without* $b\{\{\}\}\}\}$ *be a query term with a and b being arbitrary labels or regular expressions.*

  - $q_1 := q = a\{\{$ *without* $b\{\{\ \}\}\ \}\}$
  - $q_2 := a\{\{$ *without* $b\{\{\ a\{\{\ \}\}\ \}\}\ \}\}$
  - $q_3 := a\{\{$ *without* $b\{\{\ a\{\{\ a\{\{\ \}\}\ \}\}\ \}\}\ \}\}$
  - $\ldots$

[9] If there was no such mapping, it would be easy to find a data term e such that q_2 would simulate into e, but q_1 would not

[10] which must be also index monotone in the case of ordered subterm specification of q_2

The data term l_1 [l_2 [ ]] does not simulate with q_1, but simulates with q_2, the data term l_1 [l_2 [a[ ]]] does not simulate with q_2, but simulates with q_3, etc. Note that the sequence above can be constructed irrespectively of the subterm specification of q itself, and of potential other subterms of q.

- *Let q :=b {{ without desc c{ } }} be a query term with b and c being arbitrary labels or regular expressions. Then the following infinite sequence of query terms fulfills that q_i is strictly subsumed by q_{i+1} for all $i \in \mathbb{N}$:*
  - $q_1 := q = b\{\{$ *without desc* $c\{\ \}\ \}\}$
  - $q_2 := b\{\{$ *without a{ desc* $c\{\ \}\}\ \}\}$
  - $q_3 := b\{\{$ *without a{ a{ desc* $c\{\ \}\ \}\}\ \}\}$
  - ...
- *Let q be a query term with a negative subterm that contains a variable. Then this variable can be substituted by a breadth incomplete query term such as a{{ }}, yielding a query term q_1, which is more general than q. Now the same reasoning applies as for the breadth incomplete case of this proof.*

Note that also for query terms with regular expressions containing at least one $+$ or $*$ sign appearing in a negative context, one can find such an infinite chain of strictly subsumption increasing query terms.

Theorem 6 (Decidability of Xcerpt query term subsumption) *Let q_1 and q_2 be two Xcerpt query terms. Determining whether q_1 subsumes q_2 is decidable.*

Theorems 3 and 5 together imply that for any two query terms q_1 and q_2 there is a sequence of subumption monotone query term transformations $(t1,\ldots,t_n)$ with $t_n(t_{n-1}(\ldots t_1(q_1)\ldots)) = q_2)$ if and only if q_1 subsumes q_2.

However, this does not prove that finding this transformation is feasible. Starting out from q_1 one can build up a search tree by application of all possible transformations from theorem 3. Unfortunately, this search tree is infinite, since it may contain infinite branches as Examples 9 and 10 show. Theorem **??** shows, however, that we can prune the search tree to a finite size without losing the completeness of the search algorithm.

2.7 Conclusion

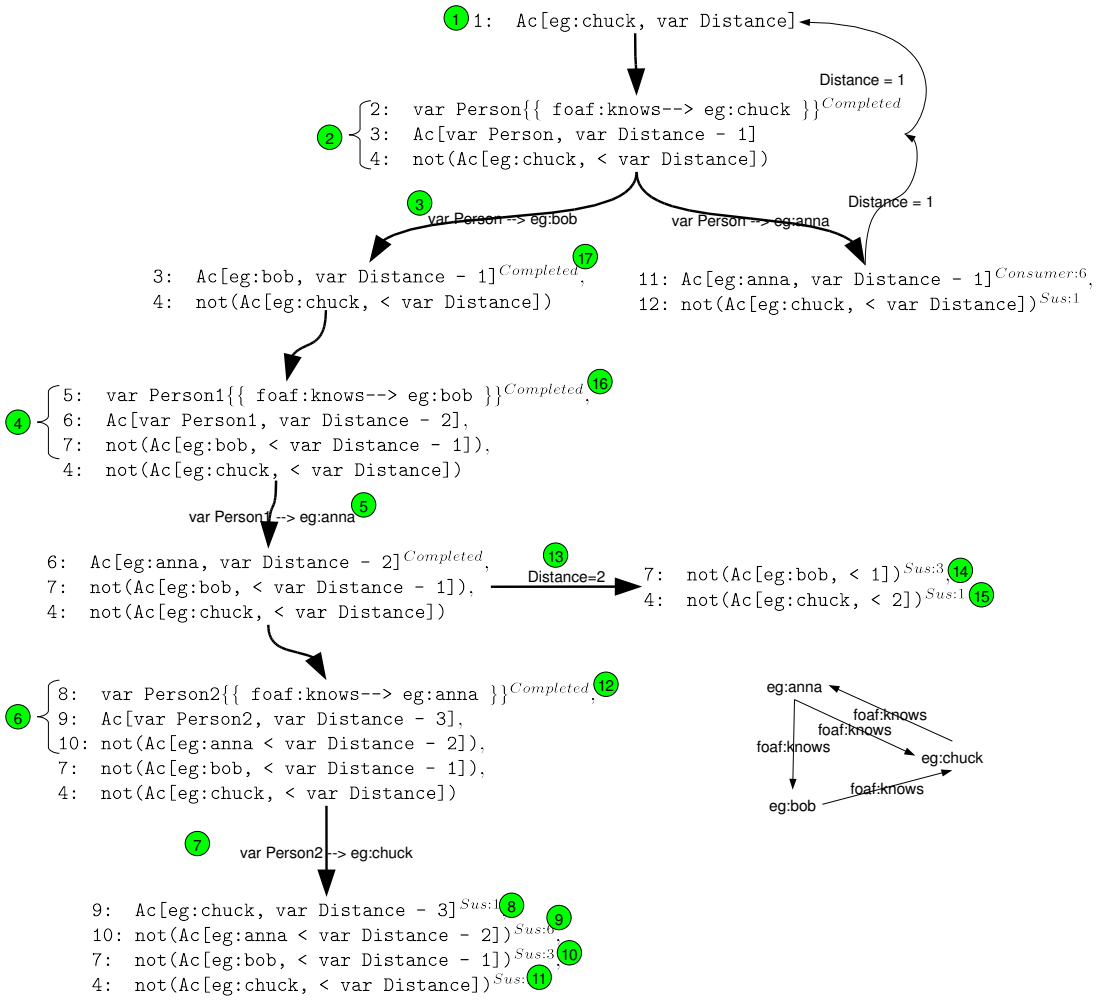
The lessons learnt in this deliverable are summarized as follows:

- With the advent of meaningful Semantic Web data, the time has come for a new class of search applying to smaller quantities of information, and answering more expressive, precise queries.
- With semantic search targeting the structure and relatedness of pieces of information, an expressive language must be supplied to authors of search queries, that is easy to understand and use.
- A language for authoring search queries must be data versatile and applicable to all kinds of graph-structured or semi-structured data.
- A pattern based approach eases the adaption of search queries to varying search needs, and allows the reuse of memoized results in case of subsuming queries.

- Negation as failure is a must for rule based query languages on the Semantic Web.
- Crawling is a query language issue.
- Rich query patterns significantly ease the authoring of ad-hoc search programs.

Finally we illustrate the inalienability of subsumption tests both for the termination of rule based programs on the Web and for its efficient evaluation and show that subsumption tests are also of relevance for efficient semantic search engines in general.

Figure 2.1 SLG Resolution for the goal `Ac{eg:chuck, var Distance}`



Acknowledgements.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Bibliography

[1] K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[2] K. Apt and K. Doets. A New Definition of SLDNF-Resolution. *Journal of Logic Programming*, 18:177–190, 1994.

[3] I. Balbin, K. Meenakshi, and K. Ramamohanarao. A Query Independent Method for Magic Set Computation on Stratified Databases. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 711–718, 1988.

[4] I. Balbin, G. Port, K. Ramamohanarao, and K. Meenakshi. Efficient Bottom-Up Computation of Queries of Stratified Databases. *Journal of Logic Programming*, 11:295–344, 1991.

[5] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 269–283. ACM Press, 1987.

[6] A. Behrend. Soft Stratification for Magic set based Query Evaluation in Deductive Databases. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 102–110. ACM Press, 2003.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[8] F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. *Data and Knowledge Engineering*, 5(4):289–312, 1990.

[9] F. Bry, N. Eisinger, T. Eiter, T. Furche, G. Gottlob, C. Ley, B. Linse, R. Pichler, and F. Wei. Foundations of rule-based query answering. In *Reasoning Web, Third International Summer School 2007*, volume 4636 of *LNCS*. Springer-Verlag, 2007.

[10] F. Bry, T. Furche, and B. Linse. Data model and query constructs for versatile web query languages: State-of-the-art and challenges for xcerpt. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning , Budva, Montenegro (10th–11th June 2006)*, volume 4187 of *LNCS*, 2006.

[11] F. Bry and S. Schaffert. Towards a declarative query and transformation language for xml and semistructured data: Simulation unification.

[12] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR '85: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110, 1985.

[13] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust.

[14] L. Cavedon and J. Lloyd. A Completeness Theorem for SLDNF-Resolution. *Journal of Logic Programming*, 7:177–191, 1989.

[15] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the Association for Computing Machinery*, 43(1):20–74, 1996.

[16] Y. Chen. A Bottom-up Query Evaluation Method for Stratified Databases. In *Proc. International Conference on Data Engineering (ICDE)*, pages 568–575. IEEE Computer Society, 1993.

[17] P. Cholak and H. A. Blair. The complexity of local stratification. *Fundam. Inform.*, 21(4):333–344, 1994.

[18] K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Base*, pages 293–322. Plenum Press, New York, NY, USA, 1978.

[19] P. M. E. De Bra and R. D. J. Post. Information retrieval in the World-Wide Web: Making client-based searching feasible. *Computer Networks and ISDN Systems*, 27(2):183–192, 1994.

[20] M. Dean and G. Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004.

[21] S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[22] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proc. Symposium on Logic Programming (SLP)*, pages 264–272, 1987.

[23] R. B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.

[24] W. Drabent. Completeness of SLDNF-Resolution for Non-Floundering Queries. In *Proc. International Symposium on Logic Programming*, page 643, 1993.

[25] C. Fan and S. W. Dietrich. Extension Table Built-ins for Prolog. *Software — Practice and Experience*, 22(7):573–597, 1992.

[26] M. Fitting. Fixpoint Semantics For Logic Programming – A Survey. *Theoretical Computer Science*, 278:25–51, 2002.

[27] C. L. Forgy. Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Expert systems: a software methodology for modern applications*, pages 324–341, 1990.

[28] J. P. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 88–98. ACM Press, 1993.

[29] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. International Conference and Symposium on Logic Programming*, pages 1070–1080. The MIT Press, 1988.

[30] P. Hayes. RDF Semantics. Technical report, W3C, 2004.

[31] M. Hersovici, M. Jacovi, Y. S. Maarek, D. Pelleg, M. Shtalhaim, and S. Ur. The shark-search algorithm. an application: tailored web site mapping. *Comput. Netw. ISDN Syst.*, 30(1-7), 1998.

[32] J. Jaffar, J.-L. Lassez, and J. Lloyd. Completeness of the Negation as Failure Rule. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 500–506, 1983.

[33] J.-M. Kerisit. A Relational Approach to Logic Programming: the Extended Alexander Method. *Theoretical Computer Science*, 69:55–68, 1989.

[34] J.-M. Kerisit and J.-M. Pugin. Efficient Query Answering on Stratified Databases. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 719–726, 1988.

[35] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[36] F. Menczer, G. Pant, P. Srinivasan, and M. E. Ruiz. Evaluating topic-driven web crawlers. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 241–249, 2001.

[37] W. Nejdl. Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 43–50. Morgan Kaufmann Publishers Inc., 1987.

[38] I. Niemelä and P. Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.

[39] G. Pant, P. Srinivasan, and F. Menczer. Exploration versus Exploitation in Topic Driven Crawlers. In *Proceedings of the Second International Workshop on Web Dynamics*, Honolulu, Hawaii, May 2002.

[40] T. Przymusinsik. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann, 1988.

[41] T. C. Przymusinski. On the Declarative and Procedural Semantics of Logic Programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

[42] K. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1990.

[43] K. Ross. A Procedural Semantics for Well-Founded Negation in Logic Programs. *Journal of Logic Programming*, 13:1–22, 1992.

[44] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, University of Munich, 2004.

[45] J. Sheperdson. Unsolvable Problems for SLDNF-Resolution. *Journal of Logic Programming*, 10:19–22, 1991.

[46] P. Srinivasan, J. Mitchell, O. Bodenreider, G. Pant, and F. Menczer. Web crawling agents for retrieving biomedical information. In *Proc. Int. Workshop on Agents in Bioinformatics (NETTAB-02)*, 2002.

[47] H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings on Third international conference on logic programming*, pages 84–98, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[48] H. Tamaki and T. Sato. OLDT Resolution with Tablulation. In *International Conference on Logic Programming*, pages 84–98, 1986.

[49] J. van den Bussche and J. Paredaens. The Expressive Power of Structured Values in Pure OODBs. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 291–299, 1991.

[50] A. van Gelder. The Alternating Fixpoint of Logic Programs With Negation. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–10, 1989.

[51] A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, 1991.

[52] L. Vieille. A Database-Complete Proof Procedure Based on SLD-Resolution. In *International Conference on Logic Programming*, pages 74–103, 1987.

[53] Wikipedia. Shortest path problem — wikipedia, the free encyclopedia, 2007. [Online; accessed 18-February-2008].

[54] Wikipedia. Well-founded relation — wikipedia, the free encyclopedia, 2008. [Online; accessed 22-February-2008].