INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Subsumption-Based Resolution for Rule Languages with Rich Unification

Olga Poppe

Aufgabensteller:

Prof. Dr. François Bry

Betreuer:

Dr. Tim Furche

Benedikt Linse

13. Juni 2009

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, 13. Juni 2009 Olga Poppe

Abstract

Finding, transforming or integrating information on the Web and Semantic Web often involves matching a structured query with semi-structured data. In logic programming languages, this notion of matching is formalized by unification. With the plethora of different data formats available on the Web, there is no single sensible notion of matching or unifying queries with data. We introduce the notion of *rich unification*, which generalizes standard unification in logic programming, matching SPARQL query patterns with RDF graphs, evaluation of XPath or XQuery expressions on XML documents, and simulation of Xcerpt terms with RDF or XML data.

On top of rich unification, we introduce a rule language $Xcerpt^s$ with recursion and negation as failure which is a variant of Xcerpt. We present an evaluation algorithm, called *Subsumption-Based Resolution for rule languages with rich unification* (SBR) resolving a query w.r.t. a locally stratified program with the bounded-term-size property. With the help of numerous $Xcerpt^s$ resolution examples we explain the main features of the algorithm and its differences to some other evaluation strategies such as SLD-, OLDT- and SLG-resolution. SBR maintains the advantages of the locally stratified SLG-resolution, such as soundness of the evaluation, completeness of the answer set, termination for programs with the bounded-term-size property. Besides, thanks to rich unification SBR is generic and because of subsumption-checks more efficient than locally stratified SLG-resolution. We also suggest some possible improvements of the algorithm.

Since SPARQL is the most important RDF query language we pay special attention to this language. Usually SPARQL rules are evaluated independently from each other and the semantics of muli-rule SPARQL programs has not been defined until now. We extend the semantics of SPARQL from single rules to possibly recursive *SPARQL programs* with negation as failure. Thanks to this extension SPARQL becomes Turing-complete. We define the notions of the *well-founded semantics* and *local stratification* for SPARQL programs. Local stratification is a syntactic feature of a program which guarantees that the well-founded model of the program is two valued, with other words the truth-value of each query w.r.t the program can be determinated.

Zusammenfassung

Suche, Transformation, und Integration von Daten im Web und Semantischen Web beinhaltet oft Auswertung einer strukturierten Anfrage gegen semi-strukturierte Daten. In logischen Programmiersprachen ist dieser Begriff der Auswertung durch Unifikation formalisiert. Wegen der vielen verschiedenen Datenformate im Web gibt es keinen eideutigen sinnvollen Begriff der Auswertung oder Unifikation von Anfragen mit Daten. Wir schlagen den Begriff der *reichen Unifikation* vor, der Standard-Unifikation aus der Logikprogrammierung, Auswertung von SPARQL Anfragen auf RDF Graphen, Auswertung von XPath oder XQuery Ausdrücke auf XML Dokumente, und Simulation von Xcerpt Termen mit RDF oder XML Daten generalisiert.

Basierend auf reicher Unifikation stellen wir die Regelsprache $Xcerpt^s$ mit Rekursion und Negation as Failure vor. $Xcerpt^s$ ist eine Variante von Xcerpt. Wir präsentieren einen Auswertungsalgorithmus, genannt *Subsumptions-Basierte Resolution für Regelsprachen mit reicher Unifikation* (SBR), der eine Anfrage bzgl. eines lokal stratifizierten Programms mit der Eigenschaft der gebundenen Termlänge auswertet. Mit Hilfe der zahlreichen $Xcerpt^s$ Auswertungsbeispiele erklären wir die wichtigsten Eigenschaften des Algorithmus und seine Unterschiede zu den anderen Auswertungsstrategien, unter anderem SLD-, OLDT-, und SLG-Resolution. SBR behält die Vorteile der lokal stratifizierten SLG-Resolution bei, wie Korrektheit der Auswertung, Vollständigkeit der Antwortmenge, Terminierung für Programme mit der Eigenschaft der gebundenen Termlänge. Außerdem, ist SBR dank reicher Unifikation generisch und wegen der Subsumptionsprüfungen effizienter als lokal stratifizierte SLG-Resolution. Wir schlagen auch mögliche Verbesserungen des Algorithmus vor.

Da SPARQL die bedeutendste RDF Anfragesprache ist, schenken wir dieser Sprache besondere Aufmerksamkeit. Normalerweise werden SPARQL Regeln unabhängig von einander ausgewertet und die Semantik von multi-Regel SPARQL Programmen wurde bis jetzt nicht definiert. Wir erweitern die Semantik von SPARQL von einzelnen Regeln zu möglicherweise rekursiven *SPARQL Programmen* mit Negation as Failure. Durch diese Erweiterung wird SPARQL Turing-vollständig. Weiterhin definieren wir die Begriffe der *wohl-fundierten Semantik* und der *lokalen Stratifikation* von SPARQL Programmen.

Lokale Stratifikation ist eine syntaktische Eigenschaft eines Programms, die gewährleistet dass das wohl-fundierte Modell des Programms zweiwertig ist, mit anderen Wörten kann der Wahrheitswert jeder Anfrage bzgl. des Programms bestimmt werden.

Acknowledgements

I would like to thank Prof. Dr. François Bry and Dr. Tim Furche for the discussions which enriched me with numerous many-sided ideas. I want to thank Benedikt Linse for the practice-oriented topic and excellent supervision. He was always there when I needed help and advice. The supervision of Prof. Dr. François Bry, Dr. Tim Furche and Benedikt Linse was both guiding and encouraging to the independent research. I appreciate the time they spent to contribute to the success of this diploma thesis. I enjoyed working with them.

I would also like to thank Dr. Norbert Eisinger for suggestions for improvement of the report on the diploma thesis.

Finally, I would like to thank my family for support during my studies.

Contents

Chapter 1

Motivation

With the amount and diversity of information on the Web rapidly and steadily growing, and with the plethora of data formats available on the Web, rule languages have been found as a suitable means for transforming and integrating information from diverse sources as well as for infering new knowledge. In the last years we have witnessed the rise of Semantic Web data formats. The Resource Description Framework (RDF) is one of the most prominent of them. SPARQL is the most important query language for RDF, that is why we pay special attention to this language. Usually SPARQL rules are evaluated independetly from each other and the semantics of muli-rule SPARQL programs has not been defined until now. But a *SPARQL program* allows execution of complex tasks (which cannot be performed by a single rule), convenient reuse and viewing of RDF graphs constructed by other rules, and even more efficient evaluation of rules. That is why we extend the declarative semantics of SPARQL from single rules to SPARQL programs, in particular the well-founded semantics for SPARQL programs is defined in this research. We have chosen the well-founded semantics because it assigns a unique model to every SPARQL program using a three valued logic and it is the minimal model of all three valued models of a program. Two valued semantics in contrast run into alternated fixpoints or multiple models in many cases. This extension of the declarative semantics makes SPARQL Turing-complete.

Spurred by the emergence of rule languages, the availability and exchange of rules on the Web is expected to rapidly increase, however, the problem of efficiently evaluating these rules has received little attention up until now. Existing approaches for evaluating multi-rule Web programs rely on a relational representation of semi-structured Web data and a translation of Web rule programs to logic programs [25, 17, 5]. These translations fail, however, in that they cannot deal with structured values for variables and in that they cannot compare queries w.r.t. containment. Our approach does not rely on a relational

representation of Web data and translation of Web rules to logic programs, but deals with the benefits and challenges of *rich unification* in multi-rule programs instead. Standard unification considers only variants and instances of goals. We call unification going beyond the scope of standard unification, rich unification. The main features of rich unification are the following:

1. Rich unification is *asymmetric*, i.e. a query entity q of a rule language unifies with its respective data entity d, the converse is not defined.

2. Rich unification returns a *substitution set* containing all non-equivalent most general unifiers of q and d.

Rich unification is a general notion abstracting from the definitions of unification in different rule languages and emphasizing their similarity.

We present an evaluation strategy called *Subsumption-Based Resolution for rule languages with rich unification* (SBR). Since rich unification is a strong means to address verying schemas of different rule laguages, the strategy is generic, i.e. it works with an arbitrary rule language satisfying some preconditions, in particular the unification and subsumption of goals must be implemented for the language. Thanks to the consideration of rich subsumption relationships between goals, SBR is more efficient than an evaluation strategy without this property because SBR performs less redundant computations.

In particular, there are two contributions of this diploma thesis, they are:

1. Definition of the well-founded semantics of SPARQL programs and

2. Development and implemetation of a sound, complete, terminating, generic and subsumption-based resolution for rule languages with rich unification.

Chapter 2

Introduction

In this chapter the fundamental notions such as *unification* and *subsumption* are defined. In Section 2.1 we explain the differences between poor (standard) unification in logic programming languages like Prolog and rich unification in rule languages like Xcerpt. Section 2.2 is devoted to the benefits of rich unification. The aims of this diploma thesis are introduced in Section 2.3.

2.1 Xcerpt Simulation vs. Prolog Unification

In this section we pay special attention to the differences between unification in Prolog and simulation in Xcerpt. The basic notions essential for this work are also defined in this section. We assume that the Xcerpt notions, such as Xcerpt term, query, rule and program are known. Otherwise see [24].

Definition 1. (Substitution).
A *substitution* is a function σ that maps variables to terms.

A substitution σ is represented by the finite set $\{ \ \mathrm{t}_1/\mathrm{x}_1, ..., \mathrm{t}_k/\mathrm{x}_k \ \}$ where $k \in \mathbb{N}$, $\{ \ \mathrm{x}_1, ..., \mathrm{x}_k \ \}$ is its domain and $\{ \ \mathrm{t}_1, ..., \mathrm{t}_k \ \}$ with $\mathrm{t}_i = \sigma(\mathrm{x}_i), \ i \in \mathbb{N}, i \leq k$, is its codomain. For the definition of the application of a substitution to an Xcerpt constant, term, query or rule see [4].

Definition 2. (Grounding substitution).
A substitution σ is called *grounding substitution* of an Xcerpt term, query or rule x, if $\sigma(x)$ is ground.

Definition 3. (Variable renaming).

A substitution ν is a *variable renaming* if it is injective and its codomain contains only variabls.

Definition 4. (Equivalent substitutions).
Two substitutions σ and ϑ are *equivalent* if there is a variable renaming ν such that $\sigma = \nu \circ \vartheta$.

The above definitions for Xcerpt are adopted from logic. The notion of *unification* in contrast cannot be adopted because Xcerpt distinguishes data and query terms. This differentiation plays an important role for the definition of Xcerpt unification, called *simulation*. Let us consider the differences between *standard unification* of Prolog terms and *simulation* of Xcerpt terms. In this research we use the notions *standard unification, Prolog unification* and *poor unification* as synonyms.

Definition 5. (**Standard unification in Prolog** [3]).
Let T be a set of terms and $t_1, t_2 \in T$. A substitution σ is a *unifier* of t_1 and t_2 iff $\sigma(t_1) = \sigma(t_2)$. Two terms are *unifiable* if there is a unifier for them. A unifier σ of t_1 and t_2 is a *most general unifier* of t_1 and t_2 iff for each unifier τ of t_1 and t_2 there is a substitution ϑ so that $\tau = \vartheta \circ \sigma$.

Unification of Prolog terms is symmetric, for instance *g(X,Y,c)* unifies with *g(Z,f(Z),c)* and vica versa *g(Z,f(Z),c)* unifies with *g(X,Y,c)*. There can be many most general unifiers of two Prolog terms but they are all equivalent, i.e. equal except for variable renaming. *g(X,Y,c)* and *g(Z,f(Z),c)* have two most general unifiers σ_1={*f(X)/Y,X/Z*} and σ_2={*Z/X,f(Z)/Y*}. But $\sigma_1 = \nu \circ \sigma_2$ where ν={*Z/X, X/Z*}.

In contrast, simulation in Xcerpt is defined between a query term and a data term. See the definition of ground query term simulation in [6]. It describes the conditions which must be satisfied in order that a ground query term q simulates into a data term d, denoted $q \preceq d$. Intuitively simulation means the following. The structure of every term can be represented as a tree. For instance the left tree in Figure 2.1 illustrates the structure of the query term *q = a{{ b{{ d, without e }}, c{{ desc f{{ /.*/ }} }} }}*. If there is a data term d such that the structure of q can be found in d, we say that q simulates into d. See Figure 2.1.

Non-ground query term simulation is defined based on ground query term simulation.

Definition 6. (Non-ground query term simulation in Xcerpt).
Let $\mathbb{D}$ be a set of data terms. A query term q containing the set of variables *Vars(q)* *simulates* into a data term d if there is a substitution $\sigma : Vars(q) \rightarrow \mathbb{D}$ such that $\sigma(q) \preceq d$ [6]. σ is the *unifier* of q and d. A unifier σ of q and d is a *most general unifier* of q and d iff for each unifier τ of q and d there is a substitution ϑ so that $\tau = \vartheta \circ \sigma$.

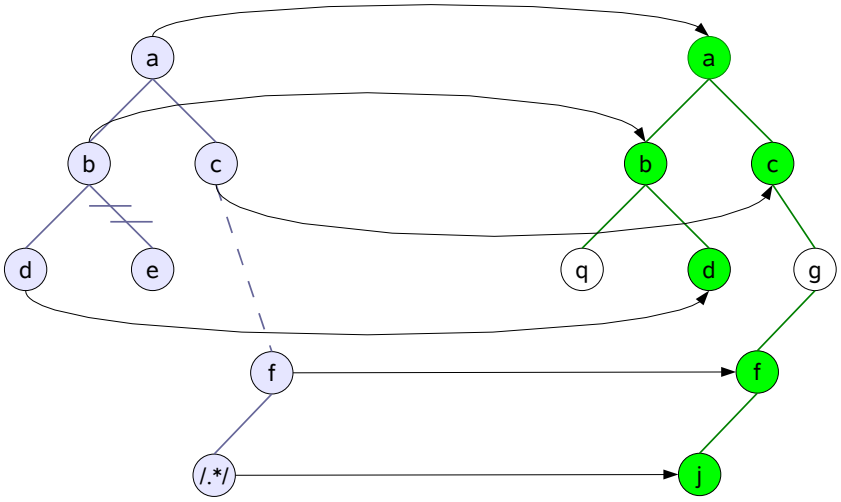

Figure 2.1: $a\{\{b\{\{d, without\ e\}\}, c\{\{desc\ f\{\{/.*/\}\}\}\}\}\} \preceq a[b[q,d], c[g[f[j]]]]$

In contrast to standard unification in Prolog, simulation in Xcerpt is asymmetric. A query term simulates into a data term. The converse is not defined. Note also that there can be many non-equivalent most general unifiers of a query term and a data term. For example *f[[var X,var Y]]⪯f[a,b,c]* where three non-equivalent most general unifiers are possible, namely σ_1={*a/X,b/Y*}, σ_2={*b/X,c/Y*} and σ_3={*a/X,c/Y*}. We call the set $\{\sigma_1, \sigma_2, \sigma_3\}$ a substitution set of *f[[var X,var Y]]* and *f[a,b,c]*.

Definition 7. (Substitution set).
The set of all non-equivalent most general unifiers of a query term q and a data term d is called *substitution set* of q and d.

Standard unification of two Prolog terms returns a substitution while simulation of an Xcerpt query term into a data term returns a substitution set.

Since we want to develop a subsumption-based evaluation strategy, *subsumption* is one of the core notions of this work. The definitions of subsumption in Prolog and subsumption in Xcerpt are similar. They are specified based on unification and simulation respectively.

Definition 8. (Subsumption in Prolog).
Let T be the set of terms and $t_1, t_2 \in T$.

$$t_1 \supseteq t_2 \text{ iff } \forall t \in T.\ t_2 \text{ unifies with } t \Rightarrow t_1 \text{ unifies with } t$$

For example, *g(X)⊇g(a)*.

Definition 9. (**Subsumption in Xcerpt** [6]).
Let D be a set of data terms, Q be a set of query terms and $q_1, q_2 \in Q$.

$$q_1 \supseteq q_2 \text{ iff } \forall d \in D.\ q_2 \preceq d \Rightarrow q_1 \preceq d.$$

In other words, a query term q_1 subsumes another query term q_2 if all data terms that q_2 simulates with are also simulated by q_1. One says q_1 is *more general* than q_2 or q_2 is *more specific* than q_1.

The superset symbol '$\supseteq$' is used to express the subsumption relationship between terms because the answer set for the subsumed term is a subset of the answer set for the subsuming term. For instance, $f\{\{desc\ a\}\} \supseteq f\{\{a\}\}$ because

$$f\{\{a\}\} \preceq f[a],\ f[a,b]...$$
$$f\{\{desc\ a\}\} \preceq f[a],\ f[g[a]],\ f[a,b],\ f[d,f[f[a]],c]...$$

2.2 Rich Unification

In this section we define the notion of *rich unification*, explain its necessity, show which features of a rule language give rise to rich unification and limit our investigation only to a part of such features.

Prolog unification considers only variants and instances of goals. Unification going beyond the scope of standard unification is called *rich unification*. In contrast to Prolog unification, rich unification returns a *substitution set* containing all non-equivalent most general unifiers of two terms. Besides, rich unifiation is *asymmetric*. (See Section 2.1 for examples.) These are the main differences between standard (poor) unification and rich unification.

Rich unification allows to resolve queries specifying only absolutely needed constraints without knowledge of data to be queried. Assume given is an XML document *bib.xml* containing information about publications. The exact contents of the document is unknown and uninteresting for us. We are looking for all authors, keywords and year of articles, so we apply the following Xcerpt rule to the document:

Listing 2.1: Xcerpt rule selecting authors, keywords and year of articles

```
1  CONSTRUCT
2    article [ all var Author,
3              var Keywords,
4              var Year       ]
5  FROM
6    in { resource [ "file:bib.xml" ],                                    (q1)
7         desc article {{ var Author   -> author {{ }},
8                         var Keywords -> keywords {{ }},
9                         var Year     -> year {{ }}     }} }
10 END
```

We do not know whether the document contains information about other publications

except for articles, where exactly the information about articles is saved in the document and what kind of information is saved for each article. And we do not care about it. The only thing we care about is the existence of any information about authors, keywords and year of articles. And the query term of the rule in Listing 2.1, let us call it q_1, specifies only these constraints and not more. Therefore q_1 could be applied to any XML document to compute solutions. Rich unification is a strong means to abstract from concrete data to be queried.

As mentioned in Section 2.1, the definition of subsumption is based on simulation, therefore one can speak of rich subsumption for a rule language with rich simulation. In this work Subsumption-Based Resolution for rule languages with rich unification (SBR) is presented. Consideration of rich subsumption between terms makes an evaluation strategy more efficient because the strategy makes no superfluous resolution steps computing already known solutions. In order to understand this point consider the following example.

Assume we are looking for all authors of recent articles about subsumption, so we use the following Xcerpt rule:

Listing 2.2: Xcerpt rule selecting all authors of recent articles about subsumption

```
1 CONSTRUCT
2   authors [ all var Author ]
3 FROM
4   in { resource [ "file:bib.xml" ],                                    (q2)
5        desc article {{ var Author -> author {{ }},
6                        keywords {{ keyword { "subsumption" } }},
7                        year { var Year }                          }}
8        where { var Year > 2007 } }
9 END
```

Note that the Xcerpt query in Listing 2.2, let us call it q_2, is subsumed by q_1. All solutions for q_2 have already been computed during the evaluation of q_1. Hence, it is more efficient to evaluate q_2 against the solutions for q_1 than against the initial XML document which can be very extensive. If the results of previous evaluations are saved they can be reused later for the resolution of subsumed queries in order to speed up their evaluation and even avoid non-termination of the evaluation of recursive queries.

Rich unification and subsumption are abstract notions, they are independent not only from conctrete data to be queried but also from a rule language. All properties and advantages of rich simulation and subsumption in Xcerpt described above are also valid for other rule or query languages such as SPARQL, XPath, XQuery and so on. Rich unification is a strong means to address varying schemas of different rule languages.

Definition 10. (Rich unification, rule language with rich unification).
Let L be a rule language, q be a query entity in L and d be the data entity to be queried

by q. If the substitution set of q and d contains more than one element the unification of q with d is called *rich unification* and L is called a *rule language with rich unification.*

The idea of unification has different names in different rule languages. For instance an Xcerpt query term *simulates* into an XML document and a SPARQL query pattern *matches* RDF data and so on. Rich unification is a general notion abstracting from the definitions of unification in different rule languages and emphasizing their similarity.

In this work we consider a variant of Xcerpt. Let us call it $Xcerpt^s$ where s means *simple*. $Xcerpt^s$ is characterized by the following features:

1. Query terms can be incomplete. This is denoted with help of double brackets. For example the incomplete query term *f[[var X, var Y]]* simulates into the data term *f[a,b,c]* where the substitution set { {*a/X,b/Y*}, {*b/X,c/Y*}, {*a/X,c/Y*} } is yield.

2. If a query term is incomplete the variables appearing as the children of the term can be optional. These variables must not be bound by simulation. For instance *f[[optional var X]]*$\preceq$*f[ ]*.

3. The range of a variable can be restricted. For example *f[*$\leq$*5]*$\preceq$*f[5]* and *f[6]*$\preceq$*f[*$>$*5]*. The $\leq$-symbol can appear only in query terms, the $>$-symbol can be used only in construct terms. This notation is not allowed in Xcerpt. But the idea can be expressed by *where*-clause of a query. We use this notation in this work to demonstrate that the SBR algorithm works for arbitrary unification.

4. Sum or Difference are possible as children of a construct term. For example *f[5]*$\preceq$*f[var X+1]* where X is bound to 4.

Combinations of the above four features of $Xcerpt^s$ are possible within a term, for example *f[* $\leq$*1, 3, p[[optional var Y, var Z]]]*$\preceq$*f[var V, var X+1*$>$*0, p[a]]* where the substitution set { {$\leq$*1/V, 2/X, a/Z*} } is returned.

Xcerpt has many other features contributing to rich simulation. They are unordered query terms, *where*-clause, modifiers such as *descendant*, *without*, *group by*, *order by* and so on. For the sake of brevity they are not considered in this work. But if simulation and subsumption were implemented for terms with these features, the SBR algorithm would be able to resolve queries w.r.t. arbitrary Xcerpt programs.

Since $Xcerpt^s$ is much less expressive than Xcerpt, the simulation in $Xcerpt^s$ is poorer than the simulation in Xcerpt. But it is richer than standard unification. As Figure 2.2 shows, the simulation in $Xcerpt^s$ considers variants and instances of complete terms and besides it takes the unification relationships going beyond the scope of standard unification into account. $Xcerpt^s$ is a rule language with rich simulation.

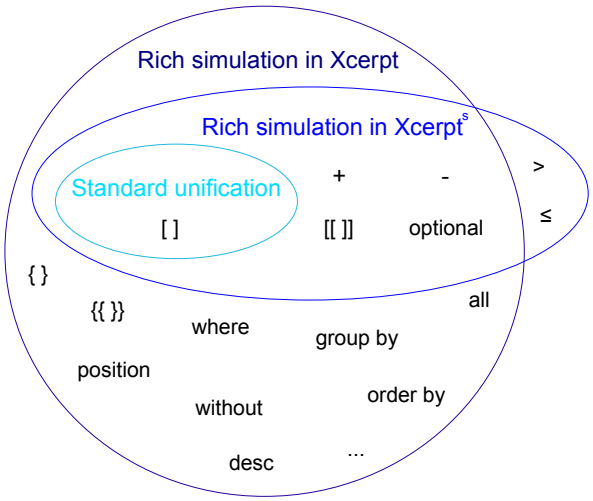

Figure 2.2: Standard unification vs. rich simulation in Xcerpt^s vs. rich simulation in Xcerpt

To define the simulation in *Xcerpt*^{s} we need the definitions of the notions of *Xcerpt*^{s} *signature* and *term*.

Definition 11. (Xcerpt^s signature).
An *Xcerpt*^{s} signature or vocabulary is a computably enumerable symbol set $\mathcal{L} = \{\text{Fun}_L^n\}$ with $n \in \mathbb{N}$, called n-ary function symbols of $\mathcal{L}$. The 0-ary function symbols are called constants of $\mathcal{L}$.

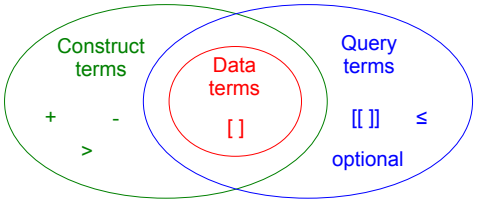

Figure 2.3: Xcerpt^s: data terms vs. construct terms vs. query terms

Let $\mathcal{L}$ be the signature of *Xcerpt*^{s}. Like Xcerpt, *Xcerpt*^{s} distinguishes data, construct and query terms. As Figure 2.3 shows, in *Xcerpt*^{s} each data term is both a construct and a query term. But there are language features typical only for construct or query terms.

Definition 12. ($\mathcal{L}$-term).
An $\mathcal{L}$-*term* is defined inductively as follows:

1. Each variable X is an $\mathcal{L}$-construct and query term.
2. Each optional variable *optional* X is an $\mathcal{L}$-query term.
3. Each constant (i.e. a string or a number) c of $\mathcal{L}$ is an $\mathcal{L}$-construct and query term.

4. Each sum $m{+}l$ where m and l are variables or numbers, is an $\mathcal{L}$-construct term.

5. Each difference $m{-}l$ where m and l are variables or numbers, is an $\mathcal{L}$-construct term.

6. Each expression of the form $\leq m$ where m is a variable, number, sum or difference, is an $\mathcal{L}$-query term.

7. Each expression of the form $m{>}c$ where m is a variable, number, sum or difference and c is a number, is an $\mathcal{L}$-construct term.

8. For each $n \in \mathbb{N}$, if f is an n-ary function symbol of $\mathcal{L}$ and $\mathrm{t}_1, ..., \mathrm{t}_n$ are $\mathcal{L}$-construct terms, then $f[\mathrm{t}_1, ..., \mathrm{t}_n]$ is an $\mathcal{L}$-construct term.

9. For each $n \in \mathbb{N}$, if f is an n-ary function symbol of $\mathcal{L}$ and $\mathrm{t}_1, ..., \mathrm{t}_n$ are $\mathcal{L}$-query terms, then $f[[\mathrm{t}_1, ..., \mathrm{t}_n]]$ is an $\mathcal{L}$-query term.

Definition 13. (Ground $\mathcal{L}$-term).
A *ground $\mathcal{L}$-term* is an $\mathcal{L}$-term without variables.

Definition 14. (Ground simulation in Xcerpt^s).
Let T_g be the set of ground $\mathcal{L}$-terms. The *ground simulation* of the members of T_g is defined as follows:
$GS = \{$
$\{\ (\ t,\ t\ )\ |\ t \in T_g\ \} \cup$
$\{\ (\ \leq e_1,\ e_2\ )\ |\ e_1, e_2, eval(e_1), eval(e_2) \in T_g \text{ and } eval(e_2) \leq eval(e_1)\ \} \cup$
$\{\ (\ e_1,\ e_1 > m\ )\ |\ e_1, eval(e_1), m \in T_g \text{ and } eval(e_1) > m\ \} \cup$
$\{\ (\ \leq e_1,\ e_1 > m\ )\ |\ e_1, eval(e_1), m \in T_g \text{ and } eval(e_1) > m\ \} \cup$
$\{\ (\ f[t_1, ..., t_n],\ f[s_1, ..., s_n]\ )\ |\ f \text{ is an arbitrary function symbol}, n \in \mathbb{N},\ f[t_1, ..., t_n],\ f[s_1, ..., s_n] \in T_g \text{ and } \forall i \in \mathbb{N}, i \leq n.\ (t_i, s_i) \in GS\ \} \cup$
$\{\ (\ f[[t_1, ..., t_n]],\ f[s_1, ..., s_k]\ )\ |\ f \text{ is an arbitrary function symbol}, n, k \in \mathbb{N},\ f[[t_1, ..., t_n]],\ f[s_1, ..., s_k] \in T_g \text{ and there is a total, injective and monotone mapping } \pi : \{t_1, ..., t_n\} \to \{s_1, ..., s_k\} \text{ such that } \forall i \in \mathbb{N}, i \leq n.\ (t_i, \pi(t_i)) \in GS\ \},$
where e_1 and e_2 are expressions of the form m, $m{+}l$ or $m{-}l$, $m, l \in \mathbb{Z}$, and $eval(e_1)$ and $eval(e_2)$ are the results of the evaluation of these expressions, $eval(e_1), eval(e_2) \in \mathbb{Z}$.

Definition 15. (Non-ground simulation in Xcerpt^s).
Let T be the set of $\mathcal{L}$-terms and $q, d \in T$. A substitution σ is the *unifier* of q and d iff for all grounding substitutions τ it holds that $(\tau \circ \sigma(q), \tau \circ \sigma(d)) \in GS$. *$q$ simulates* into d if there is a unifier of them.

The notions of a *most general unifier, substitution set* and *subsumption* for $Xcerpt^s$ are defined analogously to the definitions of these notions for Xcerpt. See Definitions 6, 7 and 9 respectively.

2.3 Aims of this Work

The main goal of this work is the development of a sound, complete, terminating, generic and subsumption-based evaluation strategy for rule languages with rich unification, in short SBR.

In particular, we consider the following points in this diploma thesis:

1. Extention of the semantics of SPARQL from single rules to SPARQL programs, especially definition of the notions of a *SPARQL program, the well-founded semantics* and *local stratification for SPARQL programs* (Chapter 3)
2. Easy-to-read informal description of the notions and actions of SBR as well as their formal definition (Chapter 4)
3. Soundness, completeness, termination and complexity of SBR w.r.t. a locally stratified program with the bounded-term-size property (Chapter 5)
4. Analysis of the related work including the comparison of SBR with other evaluation strategies such as SLD-, OLDT-, SLG-resolution. (Chapter 6)
5. Implementation of SBR in Ruby, illustrative description of the core of the algorithm and the specification of the preconditions of the utilisation of SBR by a rule language (Chapter 7)
6. Summary and suggestions for improvement of SBR (Chapter 8)

Chapter 3

Well-Founded Semantics for SPARQL Programs

In this chapter the notions of a SPARQL program and its well-founded semantics are defined. Section 3.1 describes the connections and differences of our work to other approaches on SPARQL semantics. Some basic notions are also defined in this section. Among these are *SPARQL program* (Subsection 3.1.1), *dependency graph for SPARQL programs* (Subsection 3.1.2), *matching of a SPARQL query with an RDF graph,* and *subsumption between two graphs* (Subsection 3.1.3). Section 3.2 is devoted to the well-founded semantics of SPARQL programs. In Section 3.3 local stratification for SPARQL programs is defined and its connection to the well-founded semantics is clarified.

3.1 Basic Notions

3.1.1 SPARQL Program

The Resource Description Framework (RDF) [11] is a data model for representing semantic information about World Wide Web resources. SPARQL [19] is the most important RDF query language and a W3C Recommendation since January 2008. In this work we are going to extend the semantics of SPARQL [15] from single SPARQL rules evaluated independently from each other, to possibly recursive SPARQL programs. But before we consider the semantics of SPARQL let us recall the basic RDF and SPARQL notions.

RDF information is represented in the form of triples. Each triple consists of a subject, a predicate and an object. A subject is an IRI or a blank node. A blank node is an

existentially quantified variable. It has the scope of the entire RDF graph it occurs within. A blank node does not contain any data, it serves to group data. A predicate of an RDF triple is always an IRI. An object can be an IRI, a blank node or a literal. 'RDF literals are used to identify values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals' [11].

Assume there are disjoint infinite sets *I, L, B* and *V* which denote IRIs, RDF literals, blank nodes and variables, respectively, and a set $T := I \cup L \cup B$ of RDF terms.

Definition 16. (**RDF triple** [15]).
A triple $(s, p, o) \in (I \cup B) \times I \times T$ is called an *RDF triple*. In this triple, s is a *subject*, p is a *predicate* and o is an *object*.

Definition 17. (**RDF graph** [17]).
An *RDF graph* is a finite set of RDF triples identified by an IRI.

Definition 18. (Herbrand base).
Herbrand base $\mathcal{HB}$ is the set of all RDF graphs which can be built out of the members of *T*.

Recall that in logic programming languages Herbrand base is the set of ground atoms. In SPARQL in contrast, the members of Herbrand base are not necessarily ground, they may contain blank nodes.

In this paper we restrict ourselves to CONSTRUCT rules because with their help the advantages of recursive SPARQL programs can be demonstrated best. For the sake of simplicity we ignore solution modifiers like OFFSET, LIMIT, DISTINCT and others. Simplified filter expressions and optional patterns are considered in particular to express negation as failure. Besides, FILTERs restrict solutions of a SPARQL query to those for which the filter expression evaluates to true.

Definition 19. (Filter expression).
A *filter expression* consists of the key-word 'FILTER' followed by a filter definition. Let $x, y, z \in L \cup V$ and $v \in V$. In this paper a filter definition is an expression of one of the following forms:

- $x = y$
- $x = y - z$
- $x > y$

- *!bound(v)*

Definition 20. (Graph pattern).
A *graph pattern* is recursively defined as follows:

- Each *triple pattern* $(s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (T \cup V)$[1] is a graph pattern.
- If P is a graph pattern and F is a filter expression then the *filter pattern* P *FILTER* F is a graph pattern.
- If P_1 is a graph pattern and P_2 is a basic graph pattern (i.e. a finite set of triple patterns and filter patterns) then P_1 $OPTIONAL$ P_2 is a graph pattern where P_2 is called an *optional pattern*.
- If P is a basic graph pattern and $i \in I$ then the *pattern on named graph* $GRAPH$ i P is a graph pattern. In this case the pattern P is matched against the named graph identified by the IRI i.
- A finite set of graph patterns is a graph pattern.

In this work we distinguish two types of graph patterns:

1. A *CONSTRUCT pattern* is a finite set of triple patterns.
2. A *WHERE pattern* is a graph pattern.

We assume each filter pattern of the form P $FILTER$ F to be *safe*, i.e. all variables used in a filter expression F also appear in the corresponding pattern P.

In the following the notion of a *CONSTRUCT rule* is defined. Our definition is an extension the definition of this notion in SPARQL specification [19]. Our CONSTRUCT rule may contain many CONSTRUCT patterns. Each CONSTRUCT pattern can build a new default RDF graph (like CONSTRUCT pattern defined in SPARQL specification). But besides each CONSTRUCT pattern may return one or many new named RDF graphs and it may extend one or many existing named RDF graphs. We illustrate the advantages of this idea and its differences to other approaches by an example below.

Definition 21. (SPARQL CONSTRUCT rule).
A *SPARQL CONSTRUCT rule* has the form

[1] Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable.

CONSTRUCT [[GRAPH [< IRI >]+]? < CONSTRUCT pattern >]+
[FROM < IRI >]∗
[FROM NAMED < IRI >]∗
WHERE < WHERE pattern >
where

1. < CONSTRUCT pattern > and < WHERE pattern > are graph patterns.
2. 'Each FROM clause contains an IRI that indicates a graph to be used to form the default graph. If a rule provides more than one FROM clause, providing more than one IRI to indicate the default graph, then the default graph is based on the RDF merge of the graphs obtained from representations of the resources identified by the given IRIs' [19].
3. Each FROM NAMED clause supplies an IRI for a named graph.
4. Each GRAPH $< \mathsf{IRI}_1 >, ..., < \mathsf{IRI}_n >$ provides at least one IRI to save the RDF graph returned by the respective < CONSTRUCT pattern >.

Let q be a SPARQL CONSTRUCT rule. The CONSTRUCT clause of q is called the *head* of q, denoted *head(q)*, and the WHERE clause of q is its *body*, denoted *body(q)*.

A SPARQL rule is usually evaluated independently from other rules. J. Pérez et al. formalizes the semantics of SPARQL graph pattern expressions, i.e. the evaluation of a graph pattern over an RDF graph in [15]. A. Polleres suggestes three variants of this semantics, namely bravely joining, cautiously joining and strictly joining semantics in [17]. In [18] A. Polleres proposes to consider SPARQL as a rules language allowing to query not only RDF graphs but also the result of the evaluation of CONSTRUCT statements. In this research we work out this proposal.

S. Schenk and S. Staab consider not single SPARQL patterns but a set of named RDF graphs in [25]. The authors extend named RDF graphs to so called *networked graphs* allowing:

1. 'reuse of RDF graphs enabling the dynamic copying of contents from one graph to the other,
2. viewing RDF graphs in a way that is defined by another RDF graph and
3. dynamic networking of RDF graphs. RDF graphs constitute databases and the meaning they describe comes from their dynamic networking' [25].

Definition 22. (**Networked graph**, adopted from [25]).
A *networked graph* $\mathsf{c} = (\mathsf{n}, \mathsf{G}, [\mathsf{c}_1, ..., \mathsf{c}_\mathsf{n}], \mathsf{v})$ is encoded in a named graph with name n. G is a graph, its content is explicitly included in c. $[\mathsf{c}_1, ..., \mathsf{c}_\mathsf{n}]$ is a list of networked graphs and v is a mapping from the list of networked graphs to an RDF graph called the view definition of c. The *view definition* is included in statements of the form:

$$\mathsf{n}\ \mathsf{g{:}definedBy}\ \ < \mathsf{query} >$$

where g is the namespace for the networked graph vocabulary and $< \mathsf{query} >$ is a literal containing a CONSTRUCT rule (as defined in [19]). We call the literal a *subquery* of the networked graph definition. The view definition is the union of the subqueries.

The following example from [25] illustrates Definition 22. Let ISWebGraph be a networked graph containing information about researchers working at the Information Systems and Semantic Web lab of the Institute of computer science (IFI) and IFIAdminGraph be a networked graph describing the administrative staff of IFI. Then IFIGraph in Listing 3.1 is a networked graph: IFIGraph = (u:IFIGraph, { u:ISWeb u:workingGroupOf u:IFI. u:IFI u:belongsTo u:CSDepartment. }, [ISWebGraph, IFIAdminGraph], v). v maps the RDF named graph IFIAdminGraph to itself and the RDF named graph ISWebGraph to an RDF graph where the subject of each triple is the name of a person who is not an external researcher.

Listing 3.1: Networked graph IFIGraph

```
u:IFIGraph {
 u:ISWeb u:workingGroupOf u:IFI.
 u:IFI u:belongsTo u:CSDepartment.
 u:IFIGraph g:definedBy
  "CONSTRUCT { ?s ?p ?o }
   FROM NAMED u:IFIAdminGraph
   WHERE { GRAPH u:IFIAdminGraph { ?s ?p ?o } }"
 u:IFIGraph g:definedBy
  "CONSTRUCT { ?person u:worksAt u:IFI }
   FROM NAMED u:ISWebGraph
   WHERE { GRAPH u:ISWebGraph { ?person u:worksAt u:ISWeb.
                                OPTIONAL { ?person u:status ?x.
                                   FILTER ( ?x = u:externalResearcher )
                                }.
                                FILTER ( !bound(?x) ) }
         }"
}
```

We present a more convenient possibility of reuse and viewing RDF graphs. Compare Definition 21 with the example in Listing 3.2. The first CONSTRUCT pattern builds an RDF graph equivalent to IFIGraph in Listing 3.1. Let us call the graph g. If $\mathsf{IFIGraph}_1$

already exists, g is an extension of its content, otherwise the new RDF graph g receives the identifier $\mathsf{IFIGraph}_1$. The same holds for $\mathsf{IFIGraph}_2$. Therefore the result of a CONSTRUCT pattern may specify the contents of many new named RDF graphs and extend the contents of many existing named RDF graphs.

One needs a possibility to extend an existing named RDF graph to enable recursive SPARQL programs. In each recursive call of a SPARQL rule F, the rule queries an RDF graph R containing the results of the previous evaluations of F, to compute new solutions which extend R. We shall see an example of a recursive SPARQL program below.

Besides, it is possible to construct many different RDF graphs based on the variable bindings yield during the evaluation of the same WHERE pattern of a rule. The result of the second CONSTRUCT pattern is (an extension of) the named RDF graph $\mathsf{IFIGraph}_3$. It contains a list of persons working in IFI who are not external researches.

Listing 3.2: Networked graphs $\mathsf{IFIGraph}_1$, $\mathsf{IFIGraph}_2$ and $\mathsf{IFIGraph}_3$

```
1  CONSTRUCT GRAPH IFIGraph1,IFIGraph2 { u:ISWeb u:workingGroupOf u:IFI.
2                                        u:IFI u:belongsTo u:CSDepartment.
3                                        ?s ?p ?o.
4                                        ?person u:worksAt u:ISWeb }
5            GRAPH IFIGraph3 { ?s u:worksAt u:IFI.
6                              ?person u:worksAt u:IFI }
7  FROM NAMED u:IFIAdminGraph
8  FROM NAMED u:ISWebGraph
9  WHERE { { GRAPH u:IFIAdminGraph { ?s ?p ?o } }
10           UNION
11         { GRAPH u:ISWebGraph { ?person u:worksAt u:ISWeb.
12                                OPTIONAL { ?person u:status ?x.
13                                   FILTER ( ?x = u:externalResearcher )
14                                }.
15                                FILTER ( !bound(?x) ) } }
16       }
```

Note that one needs three rather similar definitions of networked graphs to express the example in Listing 3.2. A definition of a networked graph specifies the content of a new RDF graph, it does not allow to extend an existing one. That is why we believe that Definition 21 provides a more flexible possibility of reuse and viewing RDF graphs than Definition 22.

The semantics of networked graphs is defined by their mapping to logic programs in [25]. This mapping is rather complex. We think it is more intuitive to define the notion of a *SPARQL program* and its *well-founded semantics*.

Definition 23. (SPARQL program).
A finite set of one or more SPARQL CONSTRUCT rules is called a *SPARQL program*.

The notion of SPARQL programs has not been defined before. But programs has many advantages compared to single rules evaluated independently from each other. The benefits of SPARQL programs are demonstrated by the example in Listing 3.3. In the example three SPARQL rules q_1, q_2 and q_3 which are to be resolved against the same RDF default graph in Listing 3.4, belong to the SPARQL program Q. If someone knows *Angela, Nicolas* and *Elisabeth* q_1 infers that this person is a european. If somebody knows *Edmund* too he is a bavarian (q_2). And if someone knows these four persons and has no favourite beer this person is interpreted to be a spurious bavarian by q_3.

Listing 3.3: SPARQL program Q

```
1  PREFIX foaf: <http://xmlns.com/foaf/0.1/>.
2  PREFIX :     <http://example.org/#ns>.
3  CONSTRUCT { ?x :is_a :european }                                  (q1)
4  WHERE     { ?x foaf:knows "Angela".
5              ?x foaf:knows "Nicolas".
6              ?x foaf:knows "Elisabeth" }
7  CONSTRUCT { ?x :is_a :bavarian }                                  (q2)
8  WHERE     { ?x foaf:knows "Angela".
9              ?x foaf:knows "Nicolas".
10             ?x foaf:knows "Elisabeth".
11             ?x foaf:knows "Edmund" }
12 CONSTRUCT { ?x :is_a :spurious_bavarian }                         (q3)
13 WHERE     { ?x foaf:knows "Angela".
14             ?x foaf:knows "Nicolas".
15             ?x foaf:knows "Elisabeth".
16             ?x foaf:knows "Edmund".
17             OPTIONAL { ?x :favourite_beer ?beer }.
18             FILTER ( !bound(?beer) )  }
```

Listing 3.4: RDF default graph identified by http://example.org

```
1  @prefix foaf: <http://xmlns.com/foaf/0.1/>.
2  @prefix :     <http://example.org/#ns>.
3  :tim   foaf:knows "Angela".
4  :tim   foaf:knows "Nicolas".
5  :tim   foaf:knows "Elisabeth".
6  :michi foaf:knows "Angela".
7  :michi foaf:knows "Nicolas".
8  :michi foaf:knows "Elisabeth".
9  :michi foaf:knows "Edmund".
10 :bene  foaf:knows "Angela".
11 :bene  foaf:knows "Nicolas".
12 :bene  foaf:knows "Elisabeth".
13 :bene  foaf:knows "Edmund".
14 :bene  :favourite_beer "Weissbier".
```

The first advantage of programs is a more efficient evaluation of rules because result of a query of a SPARQL program can be saved and reused to speed up the evaluation of other (more specific) queries of the program. For instance, variable bindings for the variable *?x* obtained during the evaluation of q_1 can be used to speed up the evaluation of q_2 and variable bindings yield during the evaluation of q_2 can be used for the computation of solutions for q_3.

The second advantage of SPARQL programs is rule chaining allowing viewing and reuse of RDF graphs constructed by other rules of the program. Let us rewrite the program Q so that the result of q_1 is used in q_2 and the result of q_2 is used in q_3. See Listing 3.5.

Listing 3.5: SPARQL program *Q'*

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>.
PREFIX :     <http://example.org/#ns>.
CONSTRUCT GRAPH <http://example.org/europeans> { ?x :is_a :european } (q'1)
FROM <http://example.org>
WHERE { ?x foaf:knows "Angela".
        ?x foaf:knows "Nicolas".
        ?x foaf:knows "Elisabeth" }
CONSTRUCT GRAPH <http://example.org/bavarians> { ?x :is_a :bavarian } (q'2)
FROM       <http://example.org>
FROM NAMED <http://example.org/europeans>
WHERE { GRAPH <http://example.org/europeans> { ?x :is_a :european. }
        ?x foaf:knows "Edmund" }
CONSTRUCT { ?x :is_a :spurious_bavarian }                              (q'3)
FROM       <http://example.org>
FROM NAMED <http://example.org/bavarians>
WHERE { GRAPH <http://example.org/bavarians> { ?x :is_a :bavarian. }
        OPTIONAL { ?x :favourite_beer ?beer }.
        FILTER ( !bound(?beer) ) }
```

Listing 3.6: RDF graph identified by http://example.org/europeans is the result of q'_1

```
@prefix : <http://example.org/#ns>.
:tim   :is_a :european.
:michi :is_a :european.
:bene  :is_a :european.
```

Listing 3.7: RDF graph identified by http://example.org/bavarians is the result of q'_2

```
@prefix : <http://example.org/#ns>.
:michi :is_a :bavarian.
:bene  :is_a :bavarian.
```

The result of q'_1 is the RDF graph in Listing 3.6. The first triple pattern of $body(q'_2)$ is evaluated against this named RDF graph, the second triple pattern of $body(q'_2)$ is matched

against the default graph in Listing 3.4. q_3' is built analogously to q_2'. The first triple pattern of $body(q_3')$ matches with the named graph returned by q_2' (see Listing 3.7). Other graph patterns of $body(q_3')$ are matched against the default graph. As this example shows, the advantage of rule chaining is the usage of short and clear graph patterns in a rule body which cause less errors and combined with tabling make the evaluation more efficient.

But the main advantage of rule chaining is the possibility to formulate recursive SPARQL programs like F in Listing 3.8. F computes the length of the longest path from the start node *Anna* to each other node of the finite, directed and cycle-free graph in Figure 3.1. F is evaluated against the default RDF graphs in Listings 3.9 and 3.10 and the named RDF graph in Listing 3.11 saving results of the evaluation of F. Note that the latter is constructed step by step, that is why one needs a possibility to extend an existing named RDF graph. At the beginning of the evaluation of F the named RDF graph contains only one triple, namely *:anna :has_longest_path "0".*

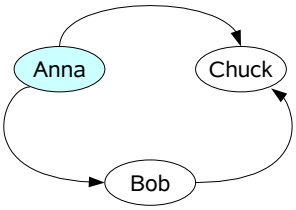

Figure 3.1: Graph depicting the contacts between the persons

Listing 3.8: SPARQL program F

```
1  PREFIX foaf: <http://xmlns.com/foaf/0.1/>.
2  PREFIX :     <http://example.org/#ns>.
3  CONSTRUCT GRAPH <http://example.org/longest_paths>
4                  { ?friend :has_longest_path ?distance }
5  FROM <http://example.org/contacts>
6  FROM <http://example.org/successors>
7  FROM NAMED <http://example.org/longest_paths>
8  WHERE { ?person foaf:knows ?friend.
9          GRAPH <http://example.org/longest_paths>
10         { ?person :has_longest_path ?x. }
11         ?y :denotes ?x.
12         ?z :successorOf ?y.
13         ?z :denotes ?distance.
14         GRAPH <http://example.org/longest_paths>
15         { OPTIONAL { ?friend :has_longest_path ?v.
16            FILTER ( ?v > ?distance )
17           }.
18           FILTER ( !bound(?v) ) }
```

```
19        }
```

Listing 3.9: RDF graph identified by http://example.org/contacts

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/>.
2 @prefix :     <http://example.org/#ns>.
3 :anna foaf:knows :bob.
4 :bob  foaf:knows :chuck.
5 :anna foaf:knows :chuck.
```

Listing 3.10: RDF graph identified by http://example.org/successors

```
1 @prefix : <http://example.org/#ns>.
2 _:I0 :denotes "0".
3 _:I1 :denotes "1". _:I1 :successorOf _:I0.
4 _:I2 :denotes "2". _:I2 :successorOf _:I1.
5 _:I3 :denotes "3". _:I3 :successorOf _:I2.
6 _:I4 :denotes "4". _:I4 :successorOf _:I3.
7 _:I5 :denotes "5". _:I5 :successorOf _:I4.
```

Listing 3.11: RDF graph identified by http://example.org/longest_paths is the result of F

```
1 @prefix : <http://example.org/#ns>.
2 :anna  :has_longest_path "0".
3 :bob   :has_longest_path "1".
4 :chuck :has_longest_path "2".
```

SPARQL programs can perform much more sophisticated tasks than single SPARQL rules because some problems can be resolved only recursively. Therefore, the definition of SPARQL programs increases the expressive power of SPARQL significantly making the language Turing-complete.

To define the semantics of SPARQL programs some technical definitions are necessary. In logic programming languages the notion of *positive* or *negative literal* underlies the definition of the well-founded semantics. We speak of *positive* or *negative graph* in SPARQL instead. Roughly speaking, the head and the body of a SPARQL rule consist of graphs. These graphs have the same features like an RDF graph but they may contain variables and constraints with FILTER- and OPTIONAL-modifiers.

Definition 24. (Head-graph).
Each CONSTRUCT pattern in a rule head is a *head-graph* of the rule. A head-graph is always positive.

Negation as failure in SPARQL can be simulated with the help of optional and filter patterns. Consider the following graph pattern selecting all bavarians who do not like beer:

```
{ ?x :is_a :bavarian.
  OPTIONAL { ?x :favourite_beer ?beer }.
  FILTER ( !bound(?beer) ) }
```

We define a shorthand notation for negation in SPARQL:

```
{ ?x :is_a :bavarian.
  ¬ ( ?x :favourite_beer ?beer. ) }
```

Definition 25. (Negated pattern).
A graph pattern of the form $\neg P$ where P is a basic graph pattern, is called a negated pattern.

Definition 26. (Positive and negative body-graph).
Let R be a SPARQL rule. There are three kinds of body-graphs:

1. If a negated pattern $\neg P$ where P is a basic graph pattern, appears in *body(R)* then $\neg P$ is a (negative) *body-graph* of R.
2. If a pattern on named graph *GRAPH* i P where $i \in I$ and P is a basic graph pattern, appears in *body(R)* then P is a (positive) *body-graph* of R.
3. If a pattern on named graph *GRAPH* i $\neg P$ where $i \in I$ and P is a basic graph pattern, appears in *body(R)* then $\neg P$ is a (negative) *body-graph* of R.
4. If a graph pattern P consisting of basic graph patterns and optional patterns, appears in *body(R)* then P is a (positive) *body-graph* of R. P may not appear within a negated pattern P' because in this case P' would be a body-graph of R.

Each body-graph is either positive or negative and all triples of a body-graph are evaluated against the same set of RDF graphs. For example, the only rule of the program F in Listing 3.8 contains the following body-graphs:
{ ?person foaf:knows ?friend. },
{ ?person :has_longest_path ?x. } and
{ ?y :denotes ?x. ?z :successorOf ?y. ?z :denotes ?distance. } are positive and
{ ¬(?friend :has_longest_path ?v. FILTER (?v > ?distance)). } is negative.

Definition 27. (Ground graph).
A *ground graph* is a graph without variables.

Note that a ground graph may contain blank nodes. An RDF graph is always ground.

Now having described the syntactic features of SPARQL we can start to consider the semantic aspects of the language.

Definition 28. (Complement of a graph).
Let G be a positive graph. The *complement* of G, written $\overline{G}$, is $\neg G$. The *complement* of a negative graph $\neg G$, written $\overline{\neg G}$, is G.

Definition 29. Let $\mathcal{HB}$ be the Herbrand base and $I \subseteq \mathcal{HB}$ be a set of ground graphs. We define:

- $\overline{I} = \{\overline{G} \mid G \in I\}$, i.e. $\overline{I}$ is the set of the complements of all graphs of I.
- $pos(I) := I \cap \mathcal{HB}$, i.e. *pos(I)* is the set of all positive graphs of I.
- $neg(I) := \overline{I} \cap \mathcal{HB}$, i.e. *neg(I)* is the set of the complements of all negative graphs of I.

Definition 30. (Consistent set of ground graphs).
A set I of ground graphs is *consistent*, iff $pos(I) \cap neg(I) = \varnothing$ holds, that is, there is no ground graph G with $G \in I$ and $\neg G \in I$.

Definition 31. (Partial and total SPARQL interpretation).
Let $\mathcal{HB}$ be the Herbrand base. A *partial SPARQL interpretation* I is a consistent set of ground graphs. I is a *total SPARQL interpretation*, iff $pos(I) \cup neg(I) = \mathcal{HB}$, that is, for each graph $G \in \mathcal{HB}$ either $G \in I$ or $\neg G \in I$ holds.

Definition 32. (Model relationship between an interpretation and a graph).
Let $\mathcal{HB}$ be the Herbrand base. The *model relationship between an interpretation I and a graph G* is defined as follows:

- If $G \in \mathcal{HB}$:
  - I satisfisies G or I is a model of G, denoted $I \models G$, iff $G \in pos(I)$
  - I falsifies G or I is not a model of G, denoted $I \not\models G$, iff $G \in neg(I)$
  - G is undefined in I iff $G \notin pos(I) \cup neg(I)$
- If G is a positive graph containing variables:
  - $I \models G$ iff $\exists D \in \mathcal{HB}$ so that G matches D (Definition 47) and $D \in pos(I)$
  - $I \not\models G$ iff $\forall D \in \mathcal{HB}$. G matches $D \Rightarrow D \in neg(I)$
  - G is undefined in I iff $\forall D \in \mathcal{HB}$. G matches $D \Rightarrow D \in neg(I)$ or D is undefined in I and $\exists D \in \mathcal{HB}$ so that G matches D and D is undefined in I
- If G is a negative graph:

- $I \models G$ iff $\forall D \in \mathcal{HB}.\ \overline{G}$ matches $D \Rightarrow D \in neg(I)$
- $I \not\models G$ iff $\exists D \in \mathcal{HB}$ so that $\overline{G}$ matches D and $D \in pos(I)$
- G is undefined in I iff $\forall D \in \mathcal{HB}.\ \overline{G}$ matches $D \Rightarrow D \in neg(I)$ or D is undefined in I and $\exists D \in \mathcal{HB}$ so that $\overline{G}$ matches D and D is undefined in I

Since SPARQL interpretation is three valued, $\not\models$ is not the same as $\neg \models$. For an interpretation I and a graph G, $I \neg \models G$ means either $I \not\models G$ or G is undefined in I.

Definition 33. (Model relationship between an interpretation and a rule body). The *model relationship between an interpretation I and a rule body B* consisting of positive or negative body-graphs $G_1, ..., G_n$ with $n \in \mathbb{N}$ is defined as follows:

- $I \models B$, iff $\forall G_i \in B.\ I \models G_i$
- $I \not\models B$, iff $\exists G_i \in B$ so that $I \not\models G_i$
- B is undefined in I iff $\forall G_i \in B.\ I \models G_i$ or G_i is undefined in I and $\exists G_i \in B$ so that G_i is undefined in I

SPARQL interpretation and RDF interpretation are quite different. Compare Definitions 31 and 34.

Definition 34. (**RDF interpretation** [5]).
An *RDF vocabulary* V consists of two disjoint sets called URIs U and literals L.
An *RDF interpretation* I of V is a tuple $(\mathsf{IR}, \mathsf{LV}, \mathsf{IP}, \mathsf{IEXT}, \mathsf{IS})$ where IR is a non-empty set of *resources* such that $\mathsf{L} \subseteq \mathsf{LV} \subseteq \mathsf{IR}$, IP is a set of *properties* and $\mathsf{IEXT} : \mathsf{IP} \to 2^{\mathsf{IR} \times \mathsf{IR}}$, and $\mathsf{IS} : \mathsf{U} \to \mathsf{IR} \cup \mathsf{IP}$ are mappings.

RDF interpretation is defined through sets and mappings whereas SPARQL interpretation is a set of graphs. Besides, RDF interpretation is two valued, in contrast to SPARQL interpretation which is three valued. Compare Definitions 33 and 35.

Definition 35. (**Model relationship between an RDF interpretation and an RDF graph** [5]).
Let I be the RDF interpretation $(\mathsf{IR}, \mathsf{LV}, \mathsf{IP}, \mathsf{IEXT}, \mathsf{IS})$ and $\mathsf{A} : \mathsf{B} \to \mathsf{IR}$ a mapping. Then $[\mathsf{I} + \mathsf{A}](\mathsf{e}) = \mathsf{a}$ if e is the literal a, $[\mathsf{I} + \mathsf{A}](\mathsf{e}) = \mathsf{IS}(\mathsf{e})$ if e is a URI, $[\mathsf{I} + \mathsf{A}](\mathsf{e}) = \mathsf{A}(\mathsf{e})$ if e is a blank node, and $[\mathsf{I} + \mathsf{A}](\mathsf{e}) = \text{true}$ if $\mathsf{e} = (\mathsf{s}, \mathsf{p}, \mathsf{o})$ is an RDF triple over V, $\mathsf{I}(\mathsf{p}) \in \mathsf{IP}$ and $(\mathsf{I}(\mathsf{s}), \mathsf{I}(\mathsf{o})) \in \mathsf{IEXT}(\mathsf{I}(\mathsf{p}))$. Finally for an RDF graph g holds: $\mathsf{I}(\mathsf{g}) = \text{true}$ if there is a mapping $\mathsf{A} : \mathsf{B} \to \mathsf{IR}$ such that $[\mathsf{I} + \mathsf{A}](t) = \text{true}$ for all RDF triples $\mathsf{t} \in \mathsf{g}$.

3.1.2 Dependency Graph for SPARQL Programs

To better understand the local stratification of SPARQL programs defined in Section 3.3 we introduce the notion of a dependency graph for SPARQL programs in this subsection.

Definition 36. (Substitution).
A *substitution* is a function ϑ mapping variables to the members of $T \cup V$.

Like in logic programming languages and in other rule languages, a substitution ϑ in SPARQL is represented by the finite set $\{ \mathrm{t}_1/?\mathrm{x}_1, ..., \mathrm{t}_k/?\mathrm{x}_k \}$ where $k \in \mathbb{N}$, $\{ ?\mathrm{x}_1, ..., ?\mathrm{x}_k \}$ is its domain and $\{ \mathrm{t}_1, ..., \mathrm{t}_k \}$ with $\mathrm{t}_i = \vartheta(?\mathrm{x}_i),\ i \in \mathbb{N}, i \leq k$, is its codomain.

Definition 37. (Application of substitution).
Let $n \in \mathbb{N}$. Application of substitution ϑ is defined as follows:

- $\vartheta(c)$=c where $c \in T$
- $\vartheta(t) = \vartheta(s)\vartheta(p)\vartheta(o)$ where t is triple pattern consisting of subject s, predicate p and object o
- ϑ*(P FILTER F)* $=$ ϑ*(P) FILTER* ϑ*(F)* where P is graph pattern and F is filter expression
- $\vartheta(P) = \{\vartheta(p_1), ..., \vartheta(p_n)\}$ where P is basic graph pattern consisting of triple or filter patterns $p_1, ..., p_n$
- ϑ*(OPTIONAL P)* $=$ *OPTIONAL* $\vartheta(P)$ where P is basic graph pattern
- ϑ*(GRAPH i P)* $=$ *GRAPH i* $\vartheta(P)$ where $i \in I$ and P is basic graph pattern
- $\vartheta(\neg P) = \neg(\vartheta(P))$ where P is basic graph pattern
- $\vartheta(G) = \{\vartheta(p_1), ..., \vartheta(p_n)\}$ where G is graph consisting of negated patterns, patterns on named graph, basic graph patterns or optional patterns $p_1, ..., p_n$
- $\vartheta(B) = \{\vartheta(G_1), ..., \vartheta(G_n)\}$ where B is rule body consisting of body-graphs $G_1, ..., G_n$
- ϑ(CONSTRUCT [[GRAPH [< IRI >]+]? h]+
  [FROM < IRI >] $*$ [FROM NAMED < IRI >]$*$
  WHERE B) =
  CONSTRUCT [[GRAPH [< IRI >]+]? ϑ(h)]+
  [FROM < IRI >] $*$ [FROM NAMED < IRI >]$*$
  WHERE ϑ(B)
  where h is a head-graph and B is the set of body-graphs of rule

Definition 38. (Instance of a graph or a rule).
Let X be a graph or a rule. X' is an *instance* of X iff there exists a substitution ϑ so that $\vartheta(X) = X'$.

Definition 39. (Grounding substitution).
Let X be a graph or a rule. A substitution ϑ is called *grounding substitution* of X, if $\vartheta(X)$ is ground.

Definition 40. (Ground instance of a graph or a rule).
An instance of a graph or a rule is *ground*, if it does not contain variables.

Definition 41. (Herbrand instantiation of a SPARQL program).
A *Herbrand instantiation of a SPARQL program* S, denoted S_H, consists of all ground instances of all rules of S.

Definition 42. (Dependency graph).
Let $\mathcal{HB}$ be the Herbrand base, S be a SPARQL program and S_H its Herbrand instantiation. *Dependency graph* $\mathrm{G}_{\mathrm{S}_H} = (V, E, L)$ of S_H is defined as follows:

- the set of vertices V is the set of the respective RDF graph(s) of S and graphs appearing in at least one rule instance $Q \in \mathrm{S}_H$,
- $E \subseteq (V \times V)$ is the set of edges of the dependency graph and
- $L : E \to \{+, -\}$ is a labeling function on the set of edges.

For each rule instance $Q \in \mathrm{S}_H$ of the form $h \leftarrow \mathrm{G}_1, ..., \mathrm{G}_k, k \in \mathbb{N}$, k edges are constructed in the dependency graph. Let G_i be body-graph of Q.

- If G_i is positive then $(\mathrm{G}_i, h) \in E$ with $L(\mathrm{G}_i, h) = +$.
- If G_i is negative then $(\overline{\mathrm{G}_i}, h) \in E$ with $L(\overline{\mathrm{G}_i}, h) = -$.

For each body-graph G_i of Q and each RDF graph $D \in \mathcal{HB}$ the following edges are constructed in the dependency graph.

- If G_i is positive and G_i matches D then $(D, \mathrm{G}_i) \in E$ with $L(D, \mathrm{G}_i) = +$.
- If G_i is negative and $\overline{\mathrm{G}_i}$ matches D then $(D, \overline{\mathrm{G}_i}) \in E$ with $L(D, \overline{\mathrm{G}_i}) = +$.

Definition 43. (Dependency relation between two graphs).
Let $\mathcal{HB}$ be the Herbrand base, S be a SPARQL program, $\mathrm{G}_{\mathrm{S}_H}$ be the dependency graph of its Herbrand instantiation S_H and $\mathrm{G}_1, \mathrm{G}_2 \in \mathcal{HB}$. G_1 *depends negatively* on G_2 iff there is a path from G_2 to G_1 in $\mathrm{G}_{\mathrm{S}_H}$ involving at least one negative edge. G_1 *depends positively* on G_2 iff there is a path from G_2 to G_1 in $\mathrm{G}_{\mathrm{S}_H}$ without negative edges.

Definition 44. (Positive and negative cycle in a dependency graph).
A *positive cycle in a dependency graph* is a cycle in the dependency graph without negative edges. A *negative cycle in a dependency graph* is a cycle in the dependency graph involving at least one negative edge.

We shall see some examples of dependency graphs in Section 3.3.

3.1.3 Matching and Subsumption of Graphs

In this subsection matching and subsumption between graphs are defined. We show that SPARQL is a rule language with rich unification.

Definition 45. (Variable renaming).
A substitution ν is a *variable renaming* if it is injective and its codomain contains only variables.

Definition 46. (Equivalent substitutions).
Two substitutions σ and ϑ are *equivalent* if there is a variable renaming ν such that $\sigma = \nu \circ \vartheta$.

For example the substitutions *{:tim/?x}* and *{:tim/?y}* are equivalent, i.e. equal except for variable renaming.

Definition 47. (Matching of a body-graph with an RDF graph, unifier).
Let G be a positive body-graph and *Vars(G)* be the set of variables appearing in G. Let D be an RDF graph and *Terms(D)* be the set of RDF terms of a subgraph of D. G *matches* D if there is a substitution $\sigma : \ Vars(G) \rightarrow Terms(D)$ such that $\sigma(G)$ is equivalent[10] to the subgraph of D. σ is a *unifier* of G and D. A unifier σ of G and D is a *most general unifier* of G and D iff for each other unifier τ of G and D there is a substitution ϑ so that $\tau = \vartheta \circ \sigma$.

SPARQL is a rule language with rich unification because matching of a body-graph G with an RDF graph D is asymmetric and many non-equivalent most general unifiers of G and D are possible. The set of all these most general unifiers is called a substitution set of G and D.

Definition 48. (Substitution set).
The set of all non-equivalent most general unifiers of a body-graph G and its associated RDF graph D is called *substitution set* of G and D.

For instance, the body-graph G_1 = *{?x foaf:knows "Angela". ?x foaf:knows "Nicolas". ?x foaf:knows "Elisabeth".}* matches the RDF graph in Listing 3.4 where the substitution set *{ {:tim/?x}, {:michi/?x}, {:bene/?x} }* is obtained. This substitution set can be saved as a set of solutions for G_1 and reused during the evaluation of another body-graph G_2 which is more specific than G_1 i.e. which is subsumed by G_1.

Definition 49. (Subsumption).
A graph G_1 *subsumes* another graph G_2, denoted $G_1 \supseteq G_2$, if all RDF graphs of the Herbrand base that G_2 matches are also matched by G_1. If $G_1 \supseteq G_2$ then G_2 is *more specific* than G_1.

As mentioned above, for the body-graphs of the program Q in Listing 3.3 the following holds: *{?x foaf:knows "Angela". ?x foaf:knows "Nicolas". ?x foaf:knows "Elisabeth".}* $\supseteq$ *{?x foaf:knows "Angela". ?x foaf:knows "Nicolas". ?x foaf:knows "Elisabeth". ?x foaf:knows "Edmund".}*

3.2 Well-Founded Semantics

This section defines the well-founded semantics for SPARQL programs. The well-founded semantics assigns a unique model to every SPARQL program using a three valued logic, i.e. each graph of the Herbrand base is either *true, false* or *undefined*. The well-founded model is the minimal model of all three valued models of the program. Two valued semantics, in contrast, run into alternated fixpoints or multiple models in many cases.

Let $\mathcal{HB}$ be the Herbrand base, S be a SPARQL program and I be a partial interpretation.

Definition 50. (Unfounded set of RDF graphs).
A set of graphs $U \subseteq \mathcal{HB}$ is *unfounded* w.r.t. S and I, if for each $h \in U$ and for each ground instance $h \leftarrow G_1, ..., G_n$ of a member of S at least one of the following holds:

1. $I \not\models G_i$ for some positive or negative body-graph G_i with $1 \leq i \leq n$ (G_i is falsified in I)
2. $G_i \in U$ for some positive body-graph G_i with $1 \leq i \leq n$ (G_i is unfounded)

Definition 51. (Greatest unfounded set).
The *greatest unfounded set* of S w.r.t. I, denoted $\mathbf{U}_S(I)$, is the union of all sets that are unfounded w.r.t. I.

Definition 52. (Well-founded semantics for SPARQL programs).
The *well-founded semantics* of S is the 'meaning' represented by the least fixed point of the operator $\mathbf{W}_S$, denoted $lfp(\mathbf{W}_S)$. We define the operator as follows:

- $h \in \mathbf{T}_S(I)$ iff there is a ground instance $h \leftarrow B$ of a member of S such that $I \models B$
- $\mathbf{U}_S(I)$ is the greatest unfounded set of S w.r.t. I
- $\mathbf{W}_S(I) = \mathbf{T}_S(I) \cup \neg\mathbf{U}_S(I)$

Definition 52 is similar to the definition of the well-founded semantics in logic. But the notions are different because the definition of the model relationship between an interpretation and a body of a SPARQL rule differs from the definition of the model relationship between an interpretation and a body of a logic rule. In SPARQL we have to take matching of a rule body with RDF graphs into account. Compare Definitions 32 and 33.

Let $\mathcal{PI} = \{I \subseteq \mathcal{HB} \cup \overline{\mathcal{HB}} \mid I \textit{ is consistent}\}$ where $\overline{\mathcal{HB}} = \{\neg A \mid A \in \mathcal{HB}\}$. Since each SPARQL program S can be easily transformed into a set of normal clauses, the operators $\mathbf{T}_S$, $\mathbf{U}_S$ and $\mathbf{W}_S$ are monotonic and the operator $\mathbf{W}_S$ has a least fixpoint $lfp(\mathbf{W}_S) = \cap\{I \in \mathcal{PI} \mid \mathbf{W}_S(I) = I\} = \cap\{I \in \mathcal{PI} \mid \mathbf{W}_S(I) \subseteq I\}$ which is a partial interpretation and a partial model of S.

Definition 53. (Well-founded model for SPARQL programs).
The *well-founded model for a SPARQL program* S is its partial model $lfp(\mathbf{W}_S)$.

Let us compute the well-founded model for the recursive SPARQL program F in Listing 3.8.

$\mathbf{W}_F \uparrow 0 = \varnothing$

$\mathbf{T}_F \uparrow 1 =$ { *{:anna foaf:knows :bob. :bob foaf:knows :chuck. :anna foaf:knows :chuck. _:I0 :denotes "0". _:I1 :denotes "1". _:I1 :successorOf _:I0. _:I2 :denotes "2". _:I2 :successorOf _:I1. _:I3 :denotes "3". _:I3 :successorOf _:I2. _:I4 :denotes "4". _:I4 :successorOf _:I3. _:I5 :denotes "5". _:I5 :successorOf _:I4. :anna :has_longest_path "0". }* } is the merge of default RDF graphs of F and its named RDF graph before the evaluation of F.
$\mathbf{U}_F \uparrow 1 =$ { *{:bob foaf:knows :anna. :chuck foaf:knows :anna. ...}* } is infinite and contains an RDF graph consisting of triples which will be never constructed by F.
$\mathbf{W}_F \uparrow 1 =$ { *{:anna foaf:knows :bob. :bob foaf:knows :chuck. :anna foaf:knows :chuck. _:I0 :denotes "0". _:I1 :denotes "1". _:I1 :successorOf _:I0. _:I2 :denotes "2". _:I2 :successorOf _:I1. _:I3 :denotes "3". _:I3 :successorOf _:I2. _:I4 :denotes "4". _:I4 :successorOf _:I3. _:I5 :denotes "5". _:I5 :successorOf _:I4. :anna :has_longest_path "0".}, ¬(:bob foaf:knows :anna. :chuck foaf:knows :anna. ...)* }

$\mathbf{T}_F \uparrow 2 = \mathbf{T}_F \uparrow 1$

$\mathbf{U}_F \uparrow 2 = \mathbf{U}_F \uparrow 1 \cup \{$ *{:anna :has_longest_path "1".}, {:anna :has_longest_path "2".}, ...* $\}$
$\mathbf{W}_F \uparrow 2 = \mathbf{W}_F \uparrow 1 \cup \{$ *¬(:anna :has_longest_path "1".), ¬(:anna :has_longest_path "2".), ...* $\}$

$\mathbf{T}_F \uparrow 3 = \mathbf{T}_F \uparrow 2$
$\mathbf{U}_F \uparrow 3 = \mathbf{U}_F \uparrow 2 \cup \{$ *{:bob :has_longest_path "2".}, {:bob :has_longest_path "3".}, ...* $\}$
$\mathbf{W}_F \uparrow 3 = \mathbf{W}_F \uparrow 2 \cup \{$ *¬(:bob :has_longest_path "2".), ¬(:bob :has_longest_path "3".), ...* $\}$

$\mathbf{T}_F \uparrow 4 = \mathbf{T}_F \uparrow 3 \cup \{$ *{:bob :has_longest_path "1".}* $\}$
$\mathbf{U}_F \uparrow 4 = \mathbf{U}_F \uparrow 3 \cup \{$ *{:chuck :has_longest_path "3".}, {:chuck :has_longest_path "4".}, ...* $\}$
$\mathbf{W}_F \uparrow 4 = \mathbf{W}_F \uparrow 3 \cup \{$ *{:bob :has_longest_path "1".}, ¬(:chuck :has_longest_path "3".), ¬(:chuck :has_longest_path "4".), ...* $\}$

$\mathbf{T}_F \uparrow 5 = \mathbf{T}_F \uparrow 4 \cup \{$ *{:chuck :has_longest_path "2".}* $\}$
$\mathbf{U}_F \uparrow 5 = \mathbf{U}_F \uparrow 4 \cup \{$ *{:bob :has_longest_path "0".}* $\}$
$\mathbf{W}_F \uparrow 5 = \mathbf{W}_F \uparrow 4 \cup \{$ *{:chuck :has_longest_path "2".}, ¬(:bob :has_longest_path "0".)* $\}$

$\mathbf{T}_F \uparrow 6 = \mathbf{T}_F \uparrow 5$
$\mathbf{U}_F \uparrow 6 = \mathbf{U}_F \uparrow 5 \cup \{$ *{:chuck :has_longest_path "0".}, {:chuck :has_longest_path "1".}* $\}$
$\mathbf{W}_F \uparrow 6 = \mathbf{W}_F \uparrow 5 \cup \{$ *¬(:chuck :has_longest_path "0".), ¬(:chuck :has_longest_path "1".)* $\} =$
$\mathbf{W}_F \uparrow 7 \Rightarrow$ A fixpoint is reached.

$lfp(\mathbf{W}_F)$ is the total well-founded model of F. It is infinite but $pos(lfp(\mathbf{W}_F))$ is finite and involves the merge of the RDF default graphs of F and its named RDF graph constructed during the evaluation of F.

3.3 Local Stratification

This section explains the connections between the well-founded semantics, dependency graph and local stratification of SPARQL programs. Local stratification is a syntactic feature of a program which guarantees that the well-founded model of the program is two valued[1]. The practical relevance of a total well-founded model is obvious. Each query w.r.t. a SPARQL program with a total well-founded model and arbitrary RDF graphs can be answered with *yes* or *no*. The answer *it is not defined* will be never returned.

Definition 54. (Local stratification for SPARQL programs).
Let $\mathcal{HB}$ be the Herbrand base and S be SPARQL program. *Local stratification* of S is a partitioning of the graphs of $\mathcal{HB}$ into a set of strata $\{\mathrm{S}_1, ..., \mathrm{S}_k\}$ with $k \in \mathbb{N}$ such that for each ground instance R of a rule in S the following holds:

1. if a graph $G \in \mathrm{S}_i$ with $i \in \mathbb{N}, i \leq k$ appears positively within the body of R then

[1]Every locally stratified SPARQL program has a total well-founded model. The proof of this statement is analogue to the proof of the same statement for Xcerpt programs, see [4].

the head of R is in S_i or in a higher stratum

2. if a graph $G \in \mathrm{S}_i$ with $i \in \mathbb{N}, i \leq k$ appears negatively within the body of R then the head of R is in a higher stratum than S_i

3. if a body-graph G matches an RDF graph D then G is in a higher or the same stratum than D

Definition 55. (Locally stratified SPARQL program).
A SPARQL program is *locally stratified* iff there exists a valid local stratification for it.

The well-founded model of a SPARQL program is total if there is no negative cycle in the dependency graph of the program. Local stratification should enforce that this condition is satisfied. Hence, the third point of Definition 54 is indispensable. Otherwise negative cycles in the dependency graph of a locally stratified program are possible. Consider the recursive SPARQL program F in Listing 3.8 and the ground rule instances f_1 and f_2 of the only rule of F in Listing 3.12. We use shorthand notation for negation and filter expressions in the listing. If there were no third point in Definition 54 F would be locally stratified even though there is a negative cycle in the dependency graph of f_1 and f_2 in Figure 3.2.

Listing 3.12: Two ground rule instances of the only rule of F

```
1  PREFIX foaf: <http://xmlns.com/foaf/0.1/>.
2  PREFIX :     <http://example.org/#ns>.
3  CONSTRUCT GRAPH <http://example.org/longest_paths>                    (f1)
4                  { :anna :has_longest_path "1" }
5  FROM <http://example.org/contacts>
6  FROM <http://example.org/successors>
7  FROM NAMED <http://example.org/longest_paths>
8  WHERE { :anna foaf:knows :anna.
9          GRAPH <http://example.org/longest_paths>
10         { :anna :has_longest_path "0". }
11         _:I0 :denotes "0".
12         _:I1 :successorOf _:I0.
13         _:I1 :denotes "1".
14         GRAPH <http://example.org/longest_paths>
15         { ¬( :anna :has_longest_path >"1" ) }
16       }
17 CONSTRUCT GRAPH <http://example.org/longest_paths>                    (f2)
18                 { :anna :has_longest_path "2" }
19 FROM <http://example.org/contacts>
20 FROM <http://example.org/successors>
21 FROM NAMED <http://example.org/longest_paths>
```

```
22 WHERE { :anna foaf:knows :anna.
23         GRAPH <http://example.org/longest_paths>
24         { :anna :has_longest_path "1". }
25         _:I1 :denotes "1".
26         _:I2 :successorOf _:I1.
27         _:I2 :denotes "2".
28         GRAPH <http://example.org/longest_paths>
29         { ¬( :anna :has_longest_path >"2" ) }
30       }
```

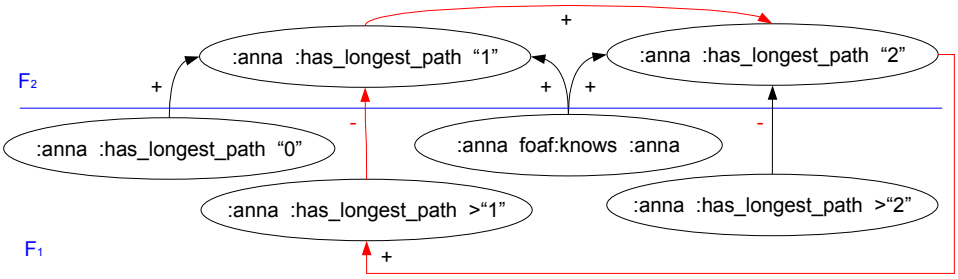

Figure 3.2: A part of the dependency graph and local stratification of the rule instances in Listing 3.12

The SPARQL program Q' in Listing 3.5 is locally stratified. Figure 3.3 shows the local stratification for some ground rule instances of the program. Yellow graphs belong to the RDF default graph of Q' in Listing 3.4. Two new RDF graphs are constructed by the program. The lilac graph is the result of the evaluation of q'_1, the green RDF graph is returned by q'_2. The white RDF graph is not constructed by the program since the body of q'_3 is *false*. The local stratification of all other ground rule instances of Q' is analogue to that shown in Figure 3.3.

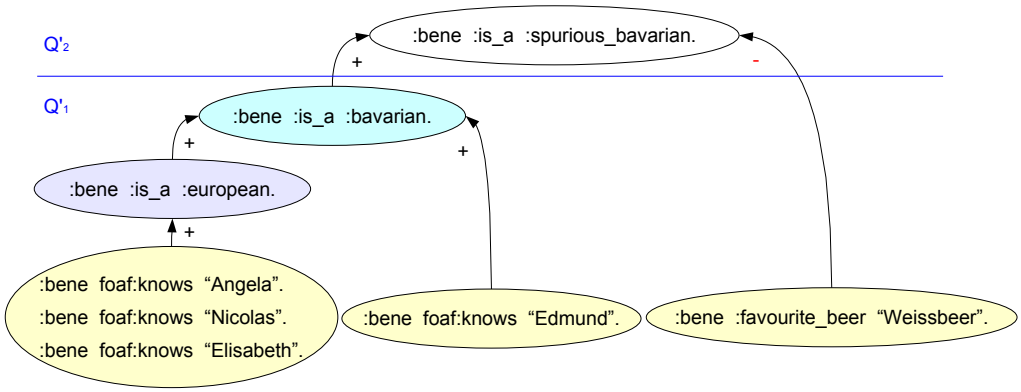

Figure 3.3: Dependency graph and local stratification for some ground rule instances of Q' in Listing 3.5

Chapter 4

Description of SBR by Examples

This is the main chapter of this work in which the Subsumption-Based Resolution algorithm for rule languages with rich unification (SBR) is presented. We start with an informal easy-to-read description of the subsumption-based resolution of a query against an $Xcerpt^s$ program (Section 4.1). The main notions of the algorithm are explained informally first and then defined formally. In Section 4.2 the resolution of a conjunctive query is described and some other notions such as *solution for a query* are defined. Since rich unification and negation are very complicated and important notions, they deserve special attention. In Sections 4.3, 4.4 and 4.5 the treatment of these two phenomena by the algorithm is explained with the help of more complex examples.

4.1 Simple Example of the Evaluation of SBR and the Main Notions of the Algorithm

In this section the main features of SBR are described with the help of an example of the resolution of a query q w.r.t. a locally stratified $Xcerpt^s$ program P. In Subsection 4.1.1 P and q are presented. Subsection 4.1.2 explains the basic ideas of SBR, for instance the difference between *solution* and *look-up goals*. Subsection 4.1.3 is devoted to the construction of the resolution tree w.r.t. P and q. In Subsection 4.1.4 we define the notions of the *state of a node* and the *state of a goal* which are very important for the resolution. The evaluation order of suspended nodes of a resolution tree is defined in Subsection 4.1.5. Subsection 4.1.6 summarizes this section.

4.1.1 Locally Stratified Xcerpt^s Program

Consider the *Xcerpt*^{s} program[1] P in Listing 4.1 and the finite directed graph in Figure 4.1 illustrating the program facts.

Listing 4.1: *Xcerpt*^{s} program P

```
1  CONSTRUCT Acquaintance[ anna, 0 ] END                                    (1)
2  CONSTRUCT                                                                (2)
3    Acquaintance[ var Friend, var Distance + 1 > 0 ]
4  FROM
5    and (
6       knows[ var Person, var Friend ],
7       Acquaintance[ var Person, var Distance ],
8       not ( Acquaintance[ var Friend, ≤ var Distance ] )
9    )
10 END
11 CONSTRUCT knows[ anna,   bob ] END                                       (3)
12 CONSTRUCT knows[ anna, chuck ] END                                       (4)
13 CONSTRUCT knows[ bob,  chuck ] END                                       (5)
14 CONSTRUCT knows[ chuck, anna ] END                                       (6)
```

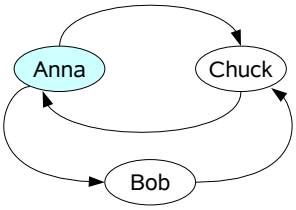

Figure 4.1: Graph depicting the contacts between the persons

The nodes of the graph are persons *Anna*, *Bob* and *Chuck*. The edges of the graph are contacts between these persons. *Anna* knows *Bob*, *Bob* knows *Chuck* and so on. These contacts are represented by the last four rules of the program P.

Anna is the start node. This is represented by the first rule of the program. *Acquaintance[anna, 0]* means that there is a path from *Anna* to herself involving 0 edges.

If one wants to know how long the shortest path from *Anna* to a friend is, one uses the second rule. The second rule says that if there is a person knowing *Anna*'s friend so that *Anna* knows this person through *Distance* edges and if there is no shorter way from *Anna* to her friend, then the shortest way from *Anna* to her friend involves *Distance+1* edges. Assume we want to know the length of the shortest way from *Anna* to *Chuck*. We

[1] The well-founded semantics for Xcerpt programs is defined in [4].

put the query q = *Acquaintance[chuck, var D].*

The reader can easily convince himself that the program is locally stratified (according to the definition of local stratification of Xcerpt programs given in [4]) and its well-founded model $lfp(\mathbf{W}_P)$ is total. See [4].

4.1.2 Basic Ideas and Advantages of SBR

SBR distinguishes solution goals and look-up goals.

Definition 56. (Solution and look-up goal).
A goal which is subsumed by no previous solution goal, is a *solution goal*. It is a *look-up goal* otherwise.

The algorithm saves all solutions for each solution goal in a table in order to reuse them during the resolution of the respective look-up goals.

Definition 57. (Table).
All solution goals, their state and solutions are saved in a *table*. The table is a list of triples. Each triple consists of the following fields:

1. solution goal
2. the state of the solution goal
3. the list of solutions for this solution goal (Definition 62)

The state of a solution goal is very much essential for the resolution of the respective look-up goals. Therefore it is also saved in the table. SBR distinguishes three states of a solution goal, namely *not completed* (initial state), *completed* and *terminated*.

Definition 58. (Completed goal).
A solution or a look-up goal is *completed* if no (other) resolution step is possible for it. It is not completed otherwise.

Roughly speaking, a goal terminates if there is no hope to find new solutions for it (compare Definitions 67, 68 and 69). Therefore, only a completed goal can terminate. Figure 4.2 describes the transitions between the states.

The idea of saving the intermediate results of the evaluation and reusing them for the resolution of similar goals is called *tabling*. The reader will see later that without tabling the resolution of the query q against the program P will not terminate. We shall also see

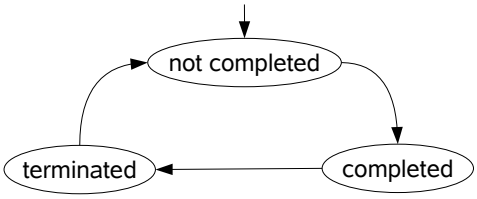

Figure 4.2: Transitions between the states of a solution goal

later that if the intermediate results of the resolution of q w.r.t. P were reused only for the variants and instances of the solution goals (like in SLG-resolution), the algorithm would make many unnecessary steps.

Therefore our algorithm using subsumption-based tabling has much better performance than an algorithm without this feature. SBR always terminates for a program with the bounded-term-size property (Definition 79) and it makes no superfluous steps.

4.1.3 Resolution Tree

At the beginning of the evaluation the table is empty. Hence q is subsumed by no previous solution goal and is therefore itself a solution goal. It is saved in the table. g simulates with the head of the second rule of P, namely *Acquaintance[var Friend, var Distance+1>0]* where the variable *Friend* is bound to *chuck* and the variable D is bound to the sum *Distance+1*, denoted { {*chuck/Friend, Distance+1/D*} }. The only substitution is propagated in the body of the second rule. The modified body of the second rule is the new query, called *resolvent*. One calls this procedure a *resolution step*.

In the definition below we use the Prolog-notation which is shorter and more familiar for the reader than the Xcerpt-notation. In all the cases the query above the line is given. All the conditions above the line must be fulfilled in order to derive the resolvent under the line.

Definition 59. (Resolution step, resolvent, dependencies of a solution goal). Let $Q_1, ..., Q_n$, H, $L_1, ..., L_m$ and C be atoms. Let q be a query and P be the program against which q is resolved.

Case 1. If q begins with a solution goal Q_1, Q_1 is resolved against the rules of P in the following way:

Query q	$Q_1 \wedge ... \wedge Q_n$
Rule r	$H \Leftarrow L_1 \wedge ... \wedge L_m$
Variable renaming ν	so that no variable of q appears in $\nu(r)$
Most general unifier σ	of $\nu(H)$ and Q_1
Resolvent	$\sigma(\nu(L_1)) \wedge ... \wedge \sigma(\nu(L_m)) \wedge \sigma(Q_2) \wedge ... \wedge \sigma(Q_n)$

Let $dep(Q_1)$ denote the *dependencies* of the solution goal Q_1. $dep(Q_1) = \sigma(\nu(L_1)), ..., \sigma(\nu(L_m))$ where $\sigma(\nu(L_1)), ..., \sigma(\nu(L_m))$ are the modified terms of the body of r. If all of them are resolved, a solution for Q_1 is found. If r is a (ground) fact H, $dep(Q_1) = \varnothing$. That means H is a solution for Q_1. For the computation of a solution for a solution goal see Definition 62.

Case 2. If q begins with a positive look-up goal Q_1, Q_1 is resolved against the solutions of the respective solution goal S, denoted *Sol(S)*, in the following way:

Query q	$Q_1 \wedge ... \wedge Q_n$
Solution	$C \in Sol(S)$
Variable renaming ν	so that no variable of q appears in $\nu(C)$
Most general unifier σ	of $\nu(C)$ and Q_1
Resolvent	$\sigma(Q_2) \wedge ... \wedge \sigma(Q_n)$

Case 3. If q begins with a negative look-up goal $\neg Q_1$, $\neg Q_1$ is *true* if the respective solution goal S has no solutions and there is no possibility to find a solution for it later. The conditions above the line guarantee that S satisfies these two requests. The following resolution steps are possible:

(a)

Query q	$\neg Q_1 \wedge ... \wedge Q_n$
$Sol(S) = \varnothing$	
S is terminated	
Resolvent	$Q_2 \wedge ... \wedge Q_n$

or

(b)

Query q	$\neg Q_1 \wedge ... \wedge Q_n$
$Sol(S) = \varnothing$	
$\neg Q_1$ is the first goal of the selected suspended node[1]	
Resolvent	$Q_2 \wedge ... \wedge Q_n$

Resolution defined here is one of the possibilities to define this notion. The peculiarities of our definition are the following:

1. For the sake of simplicity it always resolves the literals of a query beginning with the first one.

2. It distinguishes queries beginning with a solution goal and those beginning with a look-up goal. It resolves the former against the rules and facts of the program and the latter against the solutions for respective solution goals. (Note that the algorithm always treats negative literals as look-up goals.)

3. In Cases 1 and 2 of Definition 59 there can be many most general unifiers for the given query term and the head of a rule or a solution for the respective solution goal respectively because query terms can be incomplete (see [24]). Therefore the

[1] This condition is explained in Subsection 4.1.5.

number of resolvents of a query beginning with a query term Q_1 depends not only on the number of rules R such that Q_1 simulates into the head of R (if Q_1 is a solution goal) or the number of solutions C for the solution goal of Q_1 (if Q_1 is a look-up goal) but also on the number of most general unifiers of Q_1 and the head of R or Q_1 and C. See Section 4.5 for examples.

Recall that in our case the query q is a solution goal and the resolution step is as follows:

Query q	*Acquaintance(chuck, D)*
Rule 2 of P	*Acquaintance(Friend, Distance+1>0)* $\Leftarrow$ *knows(Person, Friend)* $\wedge$ *Acquaintance(Person, Distance)* $\wedge \neg$*Acquaintance(Friend,* $\leq$*Distance)*
Most general unifier σ	$\{$*chuck*$/$*Friend*, *Distance+1*$/$*D*$\}$
Resolvent	*knows(Person, chuck)* $\wedge$ *Acquaintance(Person, Distance)* $\wedge \neg$ *Acquaintance(chuck,* $\leq$*Distance)*

Variable renaming is not required in this case because the goal and the rule have no variables in common. But if there are common variables, variable renaming is indispensable. See [3] for details.

There can be many resolvents of the same query. If a query begins with a solution goal, this solution goal can simulate into the head of many rules of the program. Or if a query begins with a look-up goal, there can be many solutions for the respective solution goal. In each of these cases there can be many most general unifiers of the query and the head of a rule or the query and a solution. The algorithm makes a resolution step creating a new resolvent in each of the possible cases. Some other resolution steps are possible for all these resolvents. That is why a tree of queries is created. The root of the tree is the initial query. One calls this tree a *resolution tree*.

Definition 60. (Resolution tree).
A *resolution tree* T w.r.t. a program P and a query Q is a labeled tree where the root is Q and every node is a query. If there is a node N' in T and $N' \neq Q$ then N' has a father node N.
If N begins with a solution goal simulating into the head of a rule R in P then N' is connected to N by an edge labeled σ iff N' is a resolvent of N and R using most general unifier σ.
If N begins with a positive look-up goal then N' is connected to N by an edge labeled σ iff N' is a resolvent of N and a solution for the respective solution goal using most general unifier σ.
If N begins with a negative look-up goal then N' is connected to N by an unlabeled edge iff N' is a resolvent of N.

Now we are going to describe the growth of the tree illustrating the resolution of the

goal q = *Acquaintance[chuck, var D]* w.r.t. the program P in Listing 4.1 step by step.

$Node_1$ is the root of the tree. It contains q. See Figure 4.3. If a node N of the resolution tree begins with a solution goal S, S is coloured. For example the solution goal *Acquaintance[chuck, var D]* is blue. If a node N' is a child of N, N' has the same structure as N but with the list of the same colour as S instead of S. In the list the dependencies of S are saved. Common variables of S and the respective rule R are renamed in R and the set of variable bindings yield by the simulation of S into the head of R are propagated in all goals of N' (compare Definition 59). For instance $Node_2$ of the tree in Figure 4.3 has a blue list instead of the blue solution goal of the father $Node_1$. Note that a solution goal can appear within a list. That is why the lists within a node can be nested.

$Node_2$ is the current node now. The node begins with the solution goal *knows[var Person, chuck]*. Let us colour it in orange. The goal simulates into the head of the fourth rule of P where the variable *Person* is bound to *anna*. The rule is a fact. That is why the orange list in $Node_3$ is empty and the solution *knows[anna, chuck]* for the orange goal is saved in the table. See Table 4.1. The algorithm propagates the variable binding $\{anna/Person\}$ in all goals of $Node_3$. Compare Figure 4.3.

Definition 61. (Derivation, resolution, refutation of a query).
A *derivation* of a query Q w.r.t. a program P consists of

- a sequence $Q_1, ..., Q_n$ of nodes containing nested lists of dependencies of Q,
- a sequence $X_1, ..., X_n$ of rules of P and solutions for the solution goals,
- a sequence $\nu_1, ..., \nu_n$ of variable renamings and
- a sequence $\sigma_1, ..., \sigma_n$ of most general unifiers

such that Q_{i+1} is a resolvent of Q_i and X_i using ν_i and σ_i.

A *resolution* of Q is a finite derivation of Q which has an empty node as its last node. Otherwise the derivation is a *refutation*.

Definition 62. (Solution for a solution goal).
Let S be a solution goal. A term C is a *solution* for S w.r.t. a program P iff there is a resolution of S with

- a rule X_1 of P with the head H such that S simulates into H using the most general unifier σ_1 and the variable renaming ν_1,
- a sequence $\nu_1, ..., \nu_n$ of variable renamings and

Solution goal	State	Solutions
Acquaintance[chuck, var D]	not completed	
knows[var Person, chuck]	not completed	knows[anna, chuck]
Acquaintance[anna, var Distance]	not completed	Acquaintance[anna, 0]

Table 4.1: Table for the query *Acquaintance[chuck, var D]* after the third resolution step

- a sequence $\sigma_1, ..., \sigma_n$ of most general unifiers

such that $\sigma_n(\nu_n(...(\sigma_1(\nu_1(\ H\ )))...)) = C$.

The substitutions are applied not to a solution goal S but to a rule head H, because H is a construct term, i.e. it is always complete and therefore contains all variables which must be bound to compute a concrete solution for S. S in contrast is a query term which can be incomplete and contain no variables which are to bind. Assume the orange solution goal were *knows[[chuck]]*. The application of the substitution {*anna/Person*} to the goal would not change it, the algorithm would save *knows[[chuck]]* as a solution for the solution goal *knows[[chuck]]* and would not know whether the look-up goal *knows[anna, chuck]* is *true* or *false*. Hence only construct terms can be solutions.

Definition 63. (Solution set for a solution goal).
Let S be a solution goal. A *solution set* of S, denoted *Sol(S)*, is the set of all solutions for S.

$Node_3$ is the current node now. The empty orange list is deleted. The node begins with the green solution goal *Acquaintance[anna, var Distance]*. The goal simulates into the head of the first rule of P which is also a ground fact. That is why this resolution step is analogue to the previous one. See Table 4.1 and Figure 4.3.

4.1.4 State of a Node and State of a Goal

$Node_4$ is the current node now. It begins with the goal *not(Acquaintance[chuck, ≤0])*. As mentioned above a negative goal is always a look-up goal. It is *true* if its complement has no solutions and it is not possible to find a solution for it later. In our case the complement of the goal is a look-up goal which is subsumed by the blue solution goal.

If a look-up goal L is subsumed by a solution goal S and if there are some solutions for S in the table, L uses them. If S has not terminated yet and it is still possible to find some (other) solutions for it, the node beginning with L is suspended. It waits for the (new) solutions for S. In our case the blue solution goal has no solutions and is not

completed. That is why the algorithm suspends $Node_4$. The numbers of suspended nodes are red in the resolution tree. If the first goal of a node is a look-up goal the goal is not coloured. See Figure 4.3.

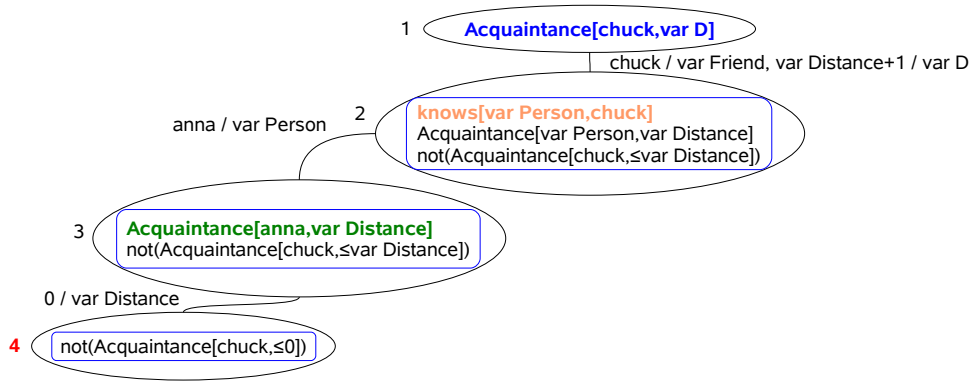

Figure 4.3: Resolution tree for the query *Acquaintance[chuck, var D]* after the third resolution step

Definition 64. (Suspended node).[1]
A node N beginning with a positive non-terminated look-up goal is *suspended* iff for the respective solution goal S at least one of the following holds:

1. S is not completed
2. There are suspended nodes in the subtree of S and S does not depend on the first goal of N[2]
3. S does not depend on itself
4. N is not the selected suspended node of the subtree of S

A node N beginning with a negative non-terminated look-up goal is *suspended* iff the respective solution goal S has no solutions and at least one of the above conditions is fulfilled.

If the algorithm considered only variants and instances of a solution goal as its look-up goals, the goal *Acquaintance[chuck, ≤0]* would be a solution goal. A new resolution tree would be created. The root of the tree would be this solution goal which would be resolved using the rules of the program. (See Section 4.3 for details.) And the algorithm

[1]The last two conditions of the definition are very complicated. They will be explained later.

[2]Roughly speaking S does not depend on a look-up goal L if L does not appear within a list of the same colour as S, that is the resolution of L is not relevant for S. Compare Definition 59.

would make some unsuccessful and superfluous resolution steps.

Every time the algorithm can make no (other) resolution step for a node *N'*, it goes back to the father node *N* of *N'* and tries to make some other resolution step possible for *N*. One calls this procedure *backtracking*. Because of tabling and backtracking SBR is *fair* and *complete*, i.e. it resolves each query of the resolution tree(s) (if it is essential for the initial query, see Definition 73 for explanation) and returns all answers for the initial query.

$Node_3$ begins with the green solution goal simulating into the head of the second rule of *P*. { {*anna/Friend1, Distance1+1/Distance*} } is the substitution set yield by this simulation. Note that the common variable *Distance* of the green goal and the second rule of *P* is renamed in the rule. The variables *Friend* and *Person* are also renamed in the rule because these variables have already been bound during the first and the second resolution steps respectively. Every time the algorithm uses repeatedly a non-ground rule, it renames the variables which have already been bound and remembers the variable names which have already been used. Thanks to this procedure every variable has only one binding within a branch of the resolution tree. It is important because if a branch of the resolution tree containing two bindings for the same variable *X* finds a solution and the head of the respective rule contains *X*, the algorithm must decide which of the two bindings for *X* to apply to the head of the rule in order to compute a solution. Compare Definition 62.

$Node_5$ is the new child of $Node_3$. It is created as described in Definition 59. The node begins with the brown solution goal *knows⌈ var Person1, anna ⌋*. It simulates into the head of the last rule of *P* which is a ground fact. And the algorithm makes a resolution step similar to the second and the third ones. See Table 4.2 and Figure 4.4.

$Node_6$ is the current node now. It begins with the look-up goal *Acquaintance⌈ chuck, var Distance1 ⌋* which is subsumed by the blue solution goal which has no solutions and is not completed yet. The algorithm suspends $Node_6$ (see Figure 4.4) and goes back to the father node.

If the algorithm worked without tabling the goal *Acquaintance⌈ chuck, var Distance1 ⌋* would be resolved against the rules of the program. Since it is only a variant of the blue goal there would be a vary similar subtree unter this goal as under the blue goal plus the two other goals of $Node_6$ in each node. This subtree would also contain a variant of the blue goal which would be also resolved with the help of the rules of *P* and so on. Therefore this branch of the tree would be infinite and the algorithm would not terminate.

The brown solution goal is completed because no other resolution steps are possible for $Node_5$. Since there are no non-empty brown lists in the subtree of $Node_5$, the subtree of the brown goal is empty and the goal terminates.

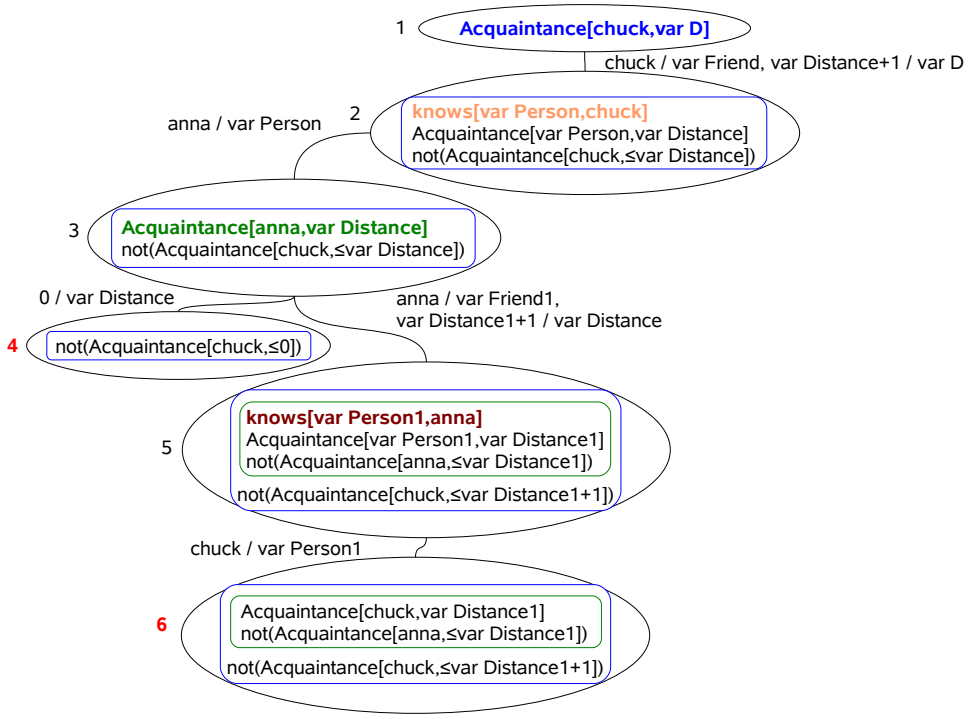

Figure 4.4: Resolution tree for the query *Acquaintance[chuck, var D]* after the seventh resolution step

The green solution goal of $Node_3$ is completed. But in contrast to the brown solution goal the green goal is does not terminate because the subtree of the goal (consisting of the green lists of $Node_5$ and $Node_6$) contains the node beginning with the non-terminated goal *Acquaintance[chuck, var Distance1]*.

These changes of the state of the solution goals are saved in the table.

Definition 65. (Subtree of a node).
Let N be a node of a resolution tree. The *subtree* of N includes the resolution tree with the root N and all auxiliary resolution trees relevant for the resolution of N[1].

Definition 66. (Subtree of a solution goal).
Let S be a solution goal. The *subtree* of S includes the resolution tree with the root containing only S and all auxiliary resolution trees relevant for the resolution of S[1].

The root of the above resolution tree contains the blue solution goal only. That is why the subtree of the first solution goal coincides with the subtree of the first node of the resolution tree. But it is not always so. If the initial query is a conjunction of query

[1]Since there is only one resolution tree in the example, this case is not relevant here and will be explained in Section 4.3.

terms the subtree of the first node of the resolution tree contains the subtree of the first solution goal and it has some other goals. See Section 4.2 for details.

Compare the subtree of the blue solution goal in Figure 4.4 with the subtree of the green solution goal in Figure 4.5. Both trees are nested. The 'blue' subtree contains three subtrees of the other solution goals. The 'orange' and 'brown' subtrees are empty, i.e. they consist of empty nodes. The branch of the subtree in Figure 4.5 ending with the empty $Node_4$ is a resolution of the green solution goal.

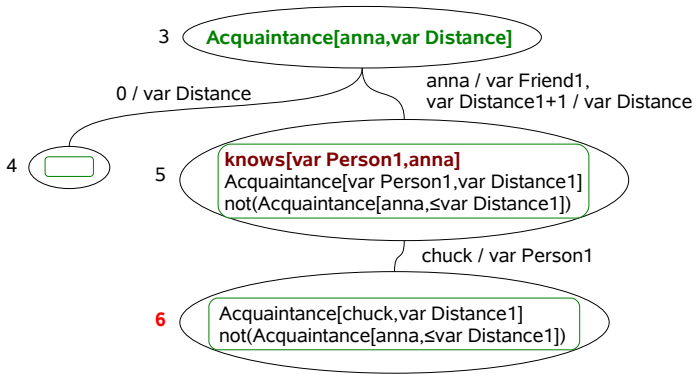

Figure 4.5: Subtree of the solution goal *Acquaintance[anna, var Distance]*

Definition 67. (Terminated positive look-up goal).
Let L be a positive look-up goal. Let S be the respective solution goal. L *terminates* iff it is completed and at least one of the following holds:

1. S is terminated
2. The node of L is the selected suspended node[2]

If S has solutions L is *true*, it is *false* otherwise.

Definition 68. (Terminated negative look-up goal).
Let $\neg L$ be a negative look-up goal. Let S be the respective solution goal. $\neg L$ *terminates* iff at least one of the following holds:

1. $Sol(S) \neq \varnothing$
2. $Sol(S) = \varnothing$ and is terminated
3. $Sol(S) = \varnothing$ and the node of $\neg L$ is the selected suspended node[2]

In the first case of the above definition $\neg L$ is *false*, in cases 2 and 3 $\neg L$ is interpreted to be *true*.

The positive look-up goal *Acquaintance[chuck, var Distance1]* of $Node_6$ does not terminate because the respective solution goal is not completed yet. The conditions of Definition 67 can be fulfilled only if the respective solution goal is completed. Because of the same reason $Node_6$ is suspended. Compare the first condition of Definition 64.

The negative look-up goal *not(Acquaintnce[chuck, ≤0])* of $Node_4$ does not terminate because the respective solution goal has no solutions and is not completed yet. Therefore $Node_4$ is suspended.

Compare the above two Definitions 67 and 68 with Definotion 59 of resolution step and Definition 64 of suspended node. Every time a look-up goal is completed and interpreted to be *true* or *false* by the algorithm, the look-up goal terminates. Otherwise the node beginning with this look-up goal is suspended.

Definition 69. (Terminated solution goal).
A solution goal *terminates* iff it is completed and its subtree contains no nodes beginning with a non-terminated goal.

As mentioned above the brown solution goal terminates because its subtree is empty. The green solution goal does not terminate because its subtree contains a non-terminated look-up goal. Every time a solution goal is completed but not terminated the algorithm tests whether this solution goal depends on itself.

Definition 70. (Solution goal depending on itself).
Let S be a completed solution goal. Let $\mathbb{S}$ be the set of all solution goals of the subtree of S including S and $\mathbb{L}$ be the set of all non-terminated look-up goals of the subtree of S.

$$S \textit{ depends on itself} \text{ iff } \mathbb{L} \neq \varnothing \text{ and } \forall L \in \mathbb{L}.\ L \subseteq S' \Rightarrow S' \in \mathbb{S}$$

A solution goal S depends on itself if a situation similar to a deadlock is reached for the subtree of the goal. It means the look-up goals of the subtree are non-terminated because they are waiting for the new solutions for the respective solution goals and the solution goals of the subtree cannot terminate because there are non-terminated look-up goals in their subtrees. We shall see an example for this situation later.

The green solution goal does not depend on itself because of the look-up goal *Acquaintance[chuck, var Distance1]* waiting for the blue solution goal which is not in the subtree of the green solution goal. The algorithm goes back to the father node.

$Node_2$ is the current node now. It begins with the orange solution goal. The goal simulates into one more fact of P. That is why there is one more branch in the subtree of

[2]This condition is explained in Subsection 4.1.5.

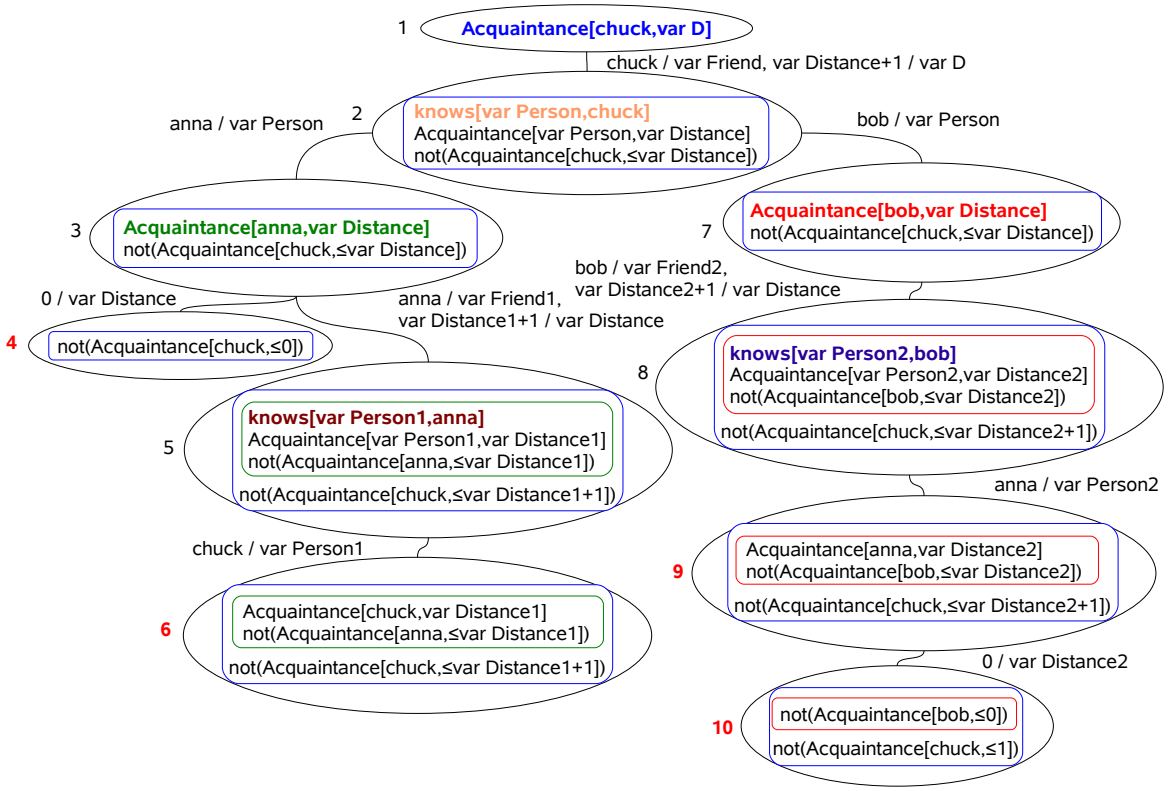

Figure 4.6: Resolution tree for the query *Acquaintance[chuck, var D]* after the sixteenth resolution step

Solution goal	State	Solutions
Acquaintance[chuck, var D]	completed	
knows[var Person, chuck]	terminated	knows[anna, chuck], knows[bob, chuck]
Acquaintance[anna, var Distance]	completed	Acquaintance[anna, 0]
knows[var Person1, anna]	terminated	knows[chuck, anna]
Acquaintance[bob, var Distance]	completed	
knows[var Person2, bob]	terminated	knows[anna, bob]

Table 4.2: Table for the query *Acquaintance[chuck, var D]* after the sixteenth resolution step

$Node_2$. It is built analogously to the previous one. See Figure 4.6. $Node_{10}$ is suspended since the red solution goal has no solutions and is not completed. $Node_9$ is suspended because there are suspended nodes in the subtree of the green solution goal and the green goal does not depend on the first goal of $Node_9$. Compare Definition 64. The lilac solution goal terminates. The red solution goal is completed but it does not terminate because of the two non-terminated look-up goals in its subtree. The look-up goal *Acquaintance[anna, var Distance2]* of suspended $Node_9$ waits for the green goal which is not in the subtree of the red goal. That is why the red solution goal does not depend on itself. The orange solution goal terminates. The blue solution goal is completed but it does not terminate because there are two non-terminated solution goals and four non-terminated look-up goals in its subtree. All new solutions and changes of the state of the solution goals are saved in the table. See Table 4.2.

4.1.5 Evaluation of Suspended Nodes

The blue solution goal depends on itself. (The first solution goal of a resolution always depends on itself.) Every time the algorithm finds such a solution goal, it processes the suspended nodes of its subtree in the order defined by Definition 71.

Definition 71. (Selected suspended node).
Let S be a solution goal depending on itself. The algorithm selects one suspended node of the subtree of S to be processed first. This node is called the *selected suspended node*. It is chosen as described by Flow diagram 4.7.

There are five conditions effecting the evaluation order of suspended nodes (compare Flow diagram 4.7). A suspended node N can be evaluated immediately if N begins with a look-up goal such that its respective solution goal has terminated or N begins with a negative look-up goal and its respective solution goal has solutions for it. In both cases the information needed for the complete evaluation of the first goal of N is already available

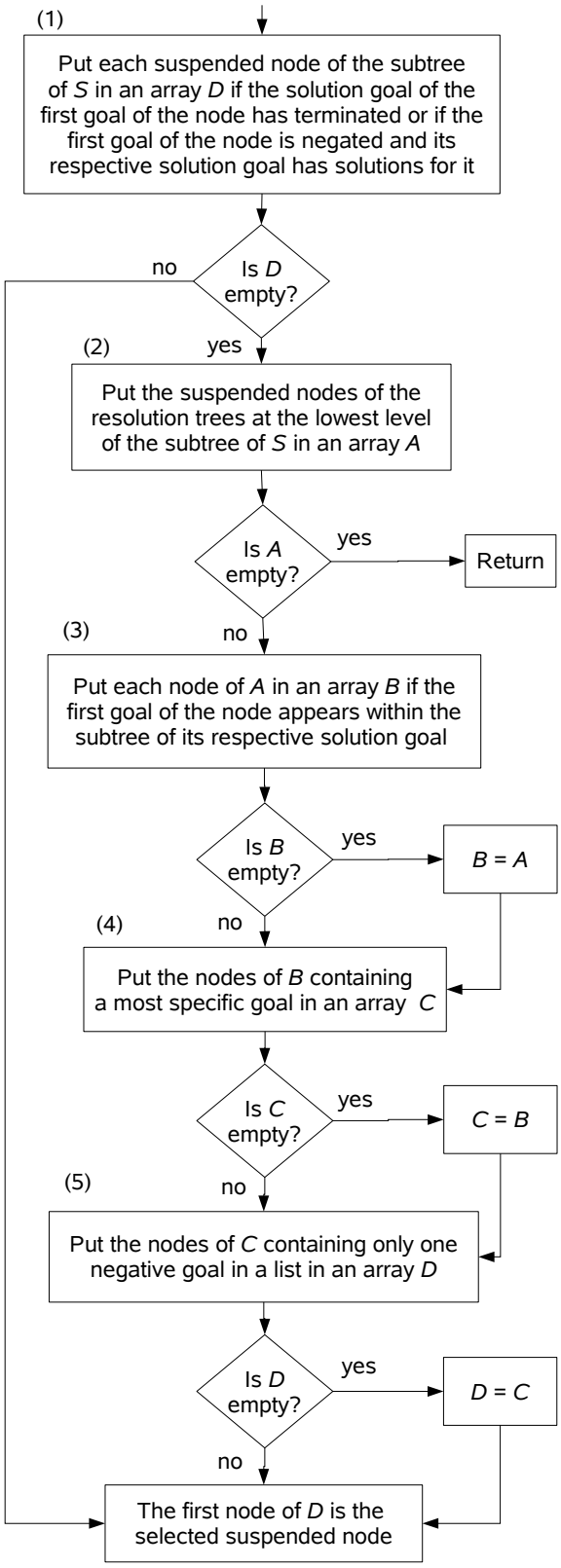

Figure 4.7: Selected suspended node

in the table and the first goal of N terminates after its derivation. In our case there is no such node.

The second condition of Flow diagram 4.7 is also irrelevant for us because the subtree of the blue goal contains only one resolution tree. The second condition enforces that the suspended nodes of auxiliary resolution trees are evaluated first. In Section 4.3 we shall see an example demonstrating the role of the first and the second conditions of the flow diagram. In our case array A contains all suspended nodes of the resolution tree in Figure 4.6.

Array B does not contain $Node_9$ any more because the first goal of the node does not appear in the subtree of its respective solution goal (compare the third condition of the flow diagram). If a solution goal S contains suspended nodes in its subtree it is still possible to find new solutions for S. Hence all nodes beginning with look-up goals waiting for S must remain suspended till all suspended nodes of the subtree of S are resolved. That is the idea of the third condition.

Now SBR has to select one node out of the array B containing nodes 4, 6 and 10. In this case the node containing the most specific goal of the suspended nodes of B is selected to be evaluated first (compare the fourth condition of the flow chart).

Definition 72. (Most specific goal).
Let $\mathbb{G}$ be a set of goals.

$$g_{ms} \in \mathbb{G} \text{ is the } \textit{most specific goal} \text{ of } \mathbb{G} \text{ iff}$$
$$\exists \mathbb{G}' \subseteq \mathbb{G}, \mathbb{G}' \neq \varnothing, \text{ such that } \forall g \in \mathbb{G}'.\ g \supseteq g_{ms} \text{ and } g_{ms} \not\supseteq g$$

In our case the set of goals appearing within the nodes of B is initialized as follows: $\mathbb{G}$ = { *not(Acquaintance[chuck, ≤0]), Acquaintance[chuck, var Distance1], not(Acquaintance[anna, ≤var Distance1]), not(Acquaintance[chuck, ≤var Distance1+1]), not(Acquaintance[bob, ≤0]), not(Acquaintance[chuck, ≤1])* }

The goal *not(Acquaintance[chuck, ≤0])* is the most specific goal of the group of goals $\mathbb{G}'$ = { *Acquaintance[chuck, var Distance1], not(Acquaintance[chuck, ≤var Distance1+1]), not(Acquaintance[chuck, ≤1])* }. Therefore $Node_4$ is the only node of array C. Compare Flow diagram 4.7.

The fourth condition is indispensable. If the algorithm processed $Node_{10}$ first and then $Node_4$, the branch of the tree containing $Node_{10}$ would return the variable binding {*2/D*} and the term *Acquaintance[chuck, 2]* would be saved in the table as a solution for the blue solution goal. This solution would not be subsumed by the look-up goal *not(Acquaintance[chuck, ≤0])* of $Node_4$. That is why the look-up goal would be *true* and the branch containing $Node_4$ would return the variable binding {*1/D*}. Therefore the second solution *Acquaintance[chuck, 1]* for the blue goal would be found. It is actually the only true solution for the user's query since the program computes the length of the

shortest path from *Anna* to *Bob* and *Chuck*.

If the algorithm evaluates $Node_4$ first and then $Node_{10}$, there will be only one solution *Acquaintance[chuck, 1]* for the blue solution goal because *not(Acquaintance[chuck, ≤1])* of $Node_{10}$ will be *false*. And the branch containing $Node_{10}$ will return no binding for the variable *D*.

If the algorithm still has a choice after the fourth condition, i.e. there are many suspended nodes in array *C*, the last condition of the flow chart selects the nodes containing only one negative goal in a list. This condition makes SBR more efficient. Section 4.4 explains why.

$Node_4$ is the selected suspended node. It begins with the look-up goal *not(Acquaintance[chuck, ≤0])*. Its complement is subsumed by the completed blue solution goal which has no solutions. That is why the negative look-up goal is *true* (compare Case 3(b) of Definition 59), it terminates (see the last condition of Definition 68) and is deleted in the child $Node_{11}$ of $Node_4$. The blue list of the new node becomes empty. The set of variable bindings $\sigma_3(\sigma_2(\sigma_1)) = \{$*0/Distance, anna/Person, chuck/Friend, 1/D*$\}$ is applied to the head of the respective rule: $\sigma_3(\sigma_2(\sigma_1($*Acquaintance[var Friend, var Distance* $+ 1 > 0$ $]))) =$ *Acquaintance[chuck, 1]* and the result is saved in the table as a solution for the blue goal (compare Definition 62 and Table 4.3). The empty blue list in $Node_{11}$ is deleted. The node becomes empty that means that a solution for the user's query is found by the branch of the tree ending with $Node_{11}$. See Figure 4.8. The order in which the suspended nodes are processed is marked in brackets after the node numbers.

The algorithm traverses the tree upwards searching for terminating solution goals or solution goals depending on itself. In our case no solution goal terminates and the only solution goal depending on itself is the first solution goal. The selected suspended node of its subtree is computed anew. It is $Node_{10}$ because it contains a more specific goal than $Node_6$. *not(Acquaintance[bob, ≤0])* is processed analogously to the negative look-up goal of $Node_4$ and the solution *Acquaintance[bob, 1]* for the red solution goal is saved in the table. See Table 4.3.

$Node_{12}$ is the next node because it contains a more specific goal than $Node_6$ (otherwise $Node_{12}$ were suspended, compare the last condition of Definition 64). The complement of *not(Acquaintance[chuck, ≤1])* is subsumed by the blue solution goal which has a solution *Acquaintance[chuck, 1]*. That is why the negative look-up goal is *false* and it terminates (compare the first condition of Definition 68). The blue list of $Node_{12}$ is not empty. That means the branch of the resolution tree which ends with $Node_{12}$ is a refutation of the blue solution goal. See Figure 4.8.

$Node_6$ is the only suspended node fulfilling the second and the third conditions of

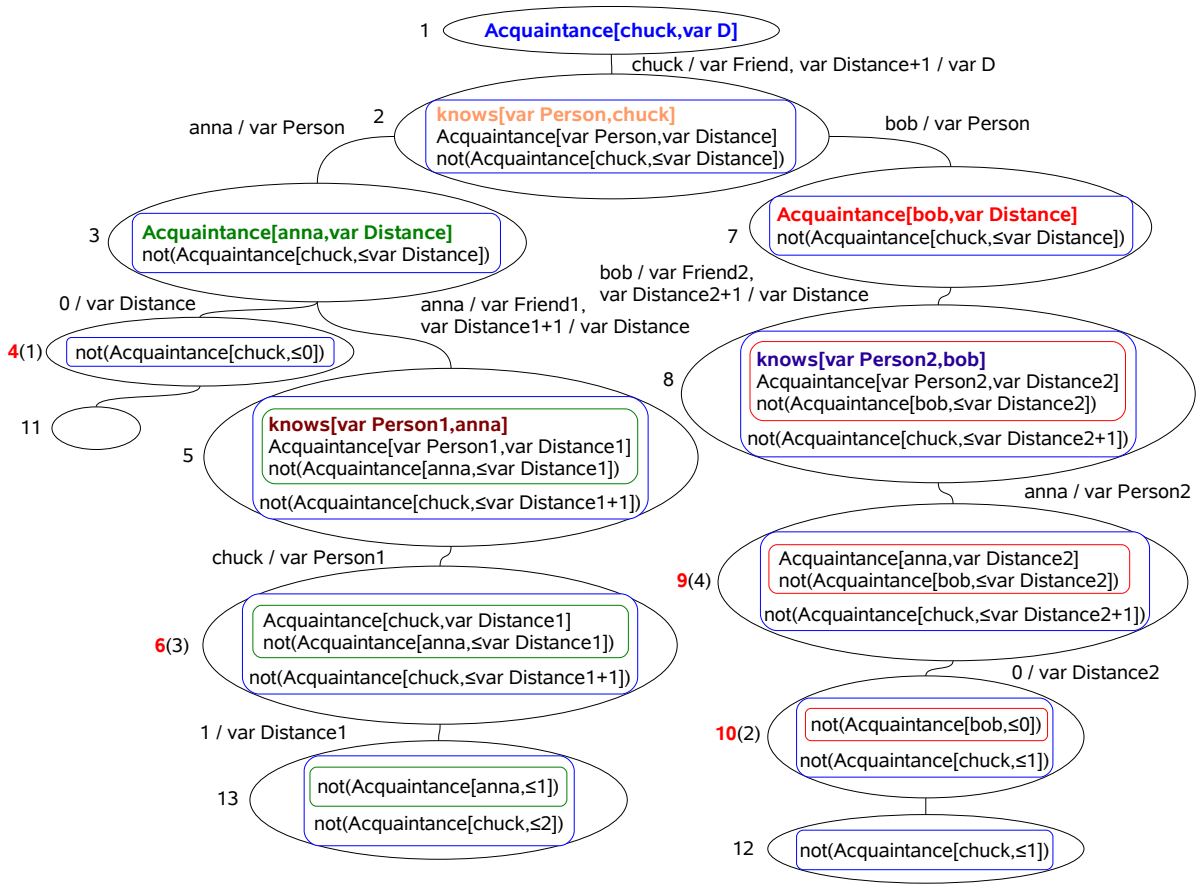

Figure 4.8: Complete resolution tree for the query *Acquaintance[chuck, var D]*

Solution goal	State	Solutions
Acquaintance[chuck, var D]	terminated	Acquaintance[chuck, 1]
knows[var Person, chuck]	terminated	knows[anna, chuck], knows[bob, chuck]
Acquaintance[anna, var Distance]	terminated	Acquaintance[anna, 0]
knows[var Person1, anna]	terminated	knows[chuck, anna]
Acquaintance[bob, var Distance]	terminated	Acquaintance[bob, 1]
knows[var Person2, bob]	terminated	knows[anna, bob]

Table 4.3: Complete table for the query *Acquaintance[chuck, var D]*

Flow chart 4.7. It begins with the positive look-up goal *Acquaintance[chuck, var Distance1]* which is subsumed by the blue solution goal. The look-up goal uses the only solution of the blue goal. $Node_{13}$ is created. See Figure 4.8. *not(Acquaintance[anna, ≤1])* is the first goal of $Node_{13}$. It is processed analogously to the negative look-up goal of $Node_{12}$. Neither the green nor the blue list of $Node_{13}$ is empty. That means the branch of the tree which ends with $Node_{13}$ is a refutation for both the green and the blue solution goals.

The algorithm traverses up the tree and finds two goals which terminate now. The look-up goal *Acquaintance[chuck, var Distance1]* of $Node_6$ terminates because of the last condition of Definition 67. Therefore the green solution goal can also terminate now (compare Definition 69). $Node_9$ can be evaluated now. Its first goal is completed and it terminates. Therefore the red solution goal and the blue solution goal also terminate. See Table 4.3.

In our case the evaluation order of suspended nodes is deterministic, i.e. array D of Flow chart 4.7 always contains only one suspended node. But it is not always so. If there are many nodes in array D it does not matter which of them is evaluated first. The reader will see some examples in the following sections.

There are no suspended nodes in the resolution tree now. The algorithm tests whether there is a look-up goal in the tree which has not used all the solutions of the respective solution goal yet. If there is such a look-up goal, the goal uses all new solutions for the respective solution goal. Some new resolution steps become possible for the nodes containing such look-up goals and propably new solutions will be found. That is why no node is deleted before this test for each look-up is completed. Infinite loops are possible (for programs with an infinite number of positive atoms in the well-founded model, such programs are not considered in this work). But without this part of the algorithm the set of solutions for the user's query can be incomplete.

In our case there is no such a look-up goal in the tree. No other resolution steps are possible. The algorithm returns the answer *Acquaintance[chuck, 1]* for the user's query *Acquaintance[chuck, var D]*. It means the shortest path from *Anna* to *Chuck* in the directed graph in Figure 4.1 involves one edge.

The resolution tree can be deleted now but it makes sense to save the table longer and reuse it for the other queries. But if the program P in Listing 4.1 changes, the table must be deleted.

4.1.6 Summary of this Section

Summarising the above description of the algorithm we would like to emphasize the following features and advantages of SBR:

1. SBR is *sound* w.r.t. the well-founded semantics.
2. The algorithm resolves queries w.r.t. *locally stratified recursive* programs.
3. Because of backward-chaining the algorithm is *goal-directed*, i.e. SBR resolves no queries which are irrelevant for the initial goal.
4. Thanks to the idea of tabling the algorithm *terminates* for a program with the bounded-term-size property.
5. Because of tabling and backtracking SBR is *fair* and *partially complete*, i.e. it resolves each query of the resolution tree(s) (if a query is essential for the initial query) and returns all answers for the user's query (if the answer set is finite) or 'false'.
6. Since our evaluation strategy bases on rich subsumption between goals, it makes no superfluous resolution steps computing already known solutions. SBR is therefore *more efficient* than an algorithm working without consideration of rich subsumption of goals.

There are some other advantages of SBR which are considered below. We shall also present some ideas for improvement of the algorithm.

4.2 Solution for a Query

In this section the difference between the *subtree of a node* and the *subtree of a solution goal* is clarified and the notins of an *auxiliary resolution tree* and a *solution for a query* are defined. All the notions are illustrated by the resolution of a conjunctive query.

Recall the $Xcerpt^s$ program P in Listing 4.1 computing the length of the shortest paths from the start node *Anna* to any other node of the finite directed graph in Figure 4.1 depicting the contacts between persons. Assume we want to know the length of the

shortest path from *Anna* to a person who knows *Chuck* such that *Chuck* is not acquainted with this person. In this case a cojunction of three query terms is needed, namely Q = *Acquaintance[var P, var D], knows [var P, chuck], not(knows [chuck, var P])*. Since in all the above conditions we are speaking about the same person, we must use the same variable P in all query terms of Q.

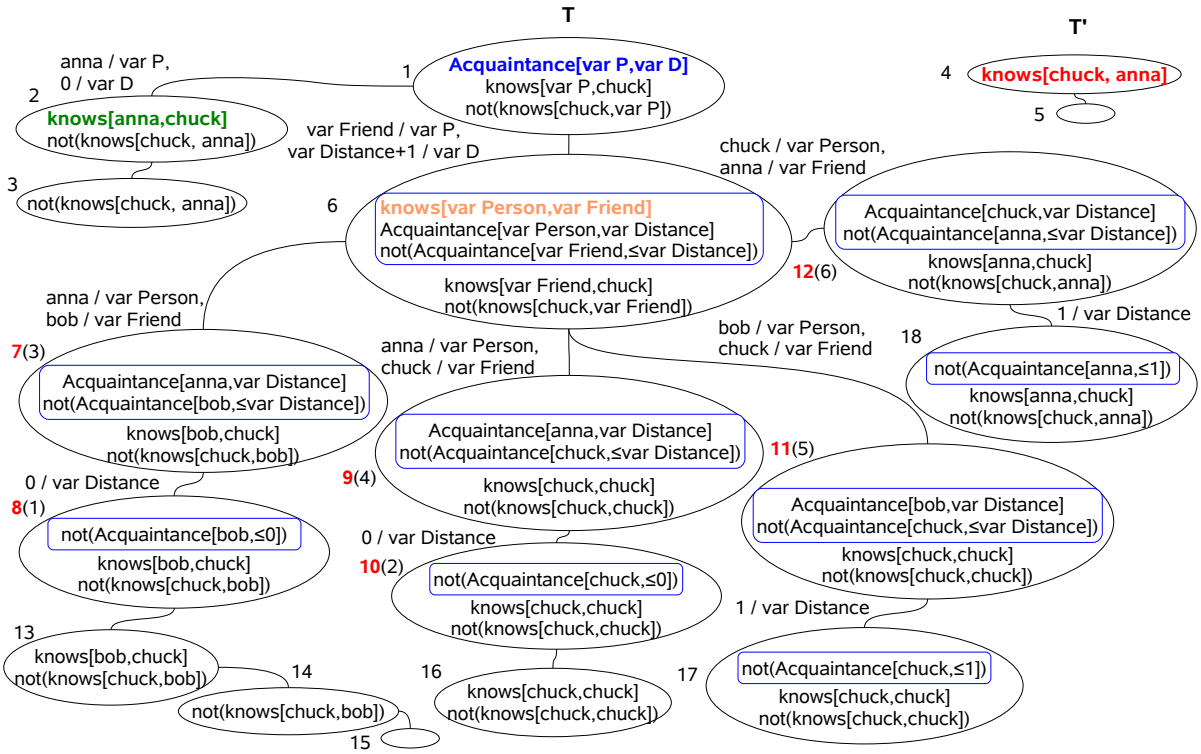

Figure 4.9: Resolution trees for the query *Acquaintance[var P, var D], knows [var P, chuck], not(knows [chuck, var P])*

Consider the resolution trees for the query Q in Figure 4.9. After the second resolution step $Node_3$ is created. It begins with the negative goal *not(knows [chuck, anna])*. Recall that the algorithm always treats a negative goal as a look-up goal. But the complement of the first goal of $Node_3$, namely *knows [chuck, anna]*, is subsumed by no previous solution goal. That is why a new resolution tree *T'* is created. The root of *T'* is the solution goal *knows [chuck, anna]*, let us colour it in red. There is a solution for the red goal, therefore *not(knows [chuck, anna])* in the old tree (let us call it T) is *false* and terminated. Since there is no non-terminated look-up goal in T waiting for the red goal, *T'* is not relevant for the resolution of Q any more. In Section 4.3 we shall see some more interesting examples of auxiliary resolution trees.

Definition 73. (Main and auxiliary resolution tree, caller resolution tree, relevant resolution tree, dependent resolution tree).

The resolution tree beginning with the initial query is the *main resolution tree*. If there is a goal $\neg q$ in a resolution tree T so that its complement q is subsumed by no previous solution goal, an *auxiliary resolution tree* T' with the root q is created. T is the *caller resolution tree* of T'. The main resolution tree is always relevant for the resolution of the initial query, in short *relevant resolution tree*. Let T be a relevant resolution tree. If there is a non-ternimated look-up goal in T waiting for a solution goal in another resolution tree T', T' is also a *relevant resolution tree*. If T' is an auxiliary resolution tree one says T *depends on* T'.

Dependency relationship between trees enforces that relevant auxiliary trees are evaluated before the main resolution tree. Because otherwise the truth-value of a look-up goal in the main tree could be determinated though an auxiliary tree can still derive some solutions for the respective solution goal. See Section 4.3 for details. That is why the dependency of an auxiliary tree from the main resolution tree is not considered.

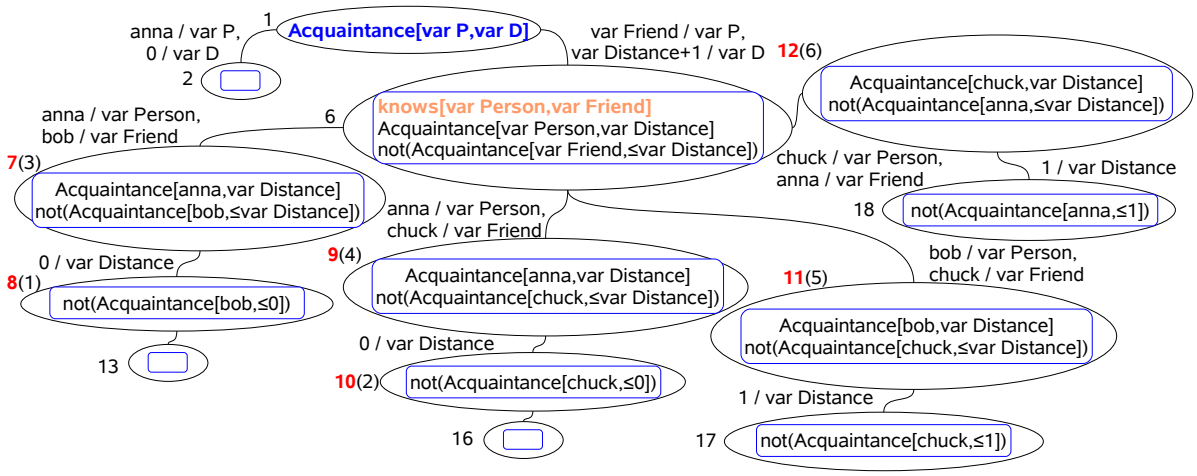

Figure 4.10: Subtree of the solution goal *Acquaintance[var P, var D]*

Compare the subtree of the first node in Figure 4.9 with the subtree of the first solution goal in Figure 4.10. Note that the auxiliary resolution tree T' is relevant neither for the resolution of Q nor for the resolution of the blue goal, that is why T' is not a part of both subtrees. Compare Definitions 65 and 66.

Compare also the evaluation order of the suspended nodes in the subtree of the blue solution goal with Definition 71. The first selected suspended node is either $Node_8$ or $Node_{10}$ because they contain the most specific goals of the suspended nodes. After these two nodes are resolved, the evaluation order of nodes 7, 9, 11 and 12 does not matter. This is a good example showing that the evaluation order of suspended nodes is in general not deterministic.

There are three resolutions of the blue solution goal but there is only one resolution of Q.

Definition 74. (Solution for a query).
Let $Q = q_1 \wedge ... \wedge q_n \wedge \neg q_{n+1} \wedge ... \wedge \neg q_{n+m}$ be a query. $C = c_1 \wedge ... \wedge c_n \wedge \neg c_{n+1} \wedge ... \wedge \neg c_{n+m}$ is a *solution* for Q w.r.t. a program iff there is a resolution R of Q where

1. $\forall i \in \{1, ..., n\}$. $\sigma_1(\nu_1(...(\sigma_l(\nu_l(q_i)))...)) = g$ is a solution goal $\Rightarrow$ c_i is a solution for g returned by R

2. $\forall i \in \{1, ..., n\}$. $\sigma_1(\nu_1(...(\sigma_l(\nu_l(q_i)))...)) = g$ is a look-up goal which is subsumed by the solution goal $S \Rightarrow c_i$ is a solution for S used in R to prove g

   We assume that k is the number of resolution steps in R and $i \leq l \leq k$ so that g is the first goal of the query Q_{l+1} which is the resolvent of the previous query Q_l using the most general unifier σ_l and the variable renaming ν_l.

3. $\forall j \in \{n+1, ..., n+m\}$. $\sigma_1(\nu_1(...(\sigma_p(\nu_p(\neg q_j)))...)) = \neg c_j$ is *true*

   $j \leq p \leq k$ so that $\neg c_j$ is the first goal of the query Q_{p+1} which is the resolvent of the previous query Q_p using the most general unifier σ_p and the variable renaming ν_p.

Definition 75. (Sound solution for a query).
A solution C for a query w.r.t. a program P is *sound* iff $lfp(\mathbf{W}_P) \models C$.

Definition 76. (Solution set for a query).
A *solution set* of a query Q, denoted *Sol(Q)*, is the set of all solutions of Q.

In Chepter 5 we prove the soundness, completeness and termination of SBR for locally stratified programs with the bounded-term-size property.

In our case *C = Acquaintance[bob, 1], knows [bob, chuck], not(knows [chuck, bob])* is a solution for *Q = Acquaintance[var P, var D], knows [var P, chuck], not(knows [chuck, var P])* because *Acquaintance[bob, 1]* is the solution for the solution goal *Acquaintance[var P, var D]* returned by the only resolution of *Q*. (Other solutions for the blue goal are irrelevant for *Q*.) *knows [bob, chuck]* and *not(knows [chuck, bob])* are look-up goals derived from the initial goals *knows [var P, chuck]* and *not(knows [chuck, var P])* respectively. They are resolved depending on the state and the solutions of the orange solution goal and they are both *true*.

4.3 Resolution Forest

In Section 4.2 the notions of the *main* and *auxiliary resolution trees, caller resolution tree, relevant resolution tree, dependent resolution tree* have been defined. In this section we give an example explaining these notions more precisely and specify the notions of a *resolution forest* and a *resolution tree at the lowest level*. The latter is essential for the evaluation order of suspended nodes (compare the second condition of Definition 71).

Consider the following locally stratified $Xcerpt^s$ program G in Listing 4.2. It has similar semantics as the program P in Listing 4.1. It also computes the length of the shortest paths from the start node *Anna* to any other node of the finite directed graph in Figure 4.11. But it uses nested function symbols n instead of the sum and the $>$-symbol in construct terms, and two auxiliary rules instead of the $\leq$-symbol in query terms. Note that because of the auxiliary rules, G has no bounded-term-size property (Definition 79). That is way SBR does not terminate for the queries which are variables or queries with the outer function symbol *AuxiliaryRule* where the second child is a variable or contains a variable. We avoid such queries.

Listing 4.2: $Xcerpt^s$ program G

```
1  CONSTRUCT Acqaintance [ anna, n[ ] ] END                          (1)
2  CONSTRUCT                                                         (2)
3     Acqaintance[ var Friend, n[ var Distance ] ]
4  FROM
5     and (
6        knows[ var Person, var Friend ],
7        Acqaintance[ var Person, var Distance ],
8        not ( AuxiliaryRule[ var Freind, var Distance ] )
9     )
10 END
11 CONSTRUCT                                                         (3)
12    AuxiliaryRule[ var Friend, n[ var Distance ] ]
13 FROM
14    AuxiliaryRule[ var Friend, var Distance ]
15 END
16 CONSTRUCT                                                         (4)
17    AuxiliaryRule[ var Friend, var Distance ]
18 FROM
19    Acqaintance[ var Friend, var Distance ]
20 END
21 CONSTRUCT knows[ anna,   bob ] END                               (5)
22 CONSTRUCT knows[ anna, chuck ] END                               (6)
23 CONSTRUCT knows[ bob,  chuck ] END                               (7)
24 CONSTRUCT knows[ chuck, anna ] END                               (8)
```

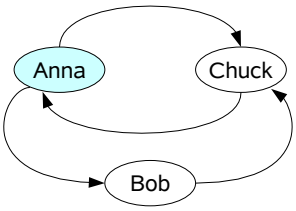

Figure 4.11: Graph depicting the contacts between the persons

Recall that a negative goal $\neg g$ is always a look-up goal. Let S be its solution goal. $\neg g$ is resolved depending on the state of S and the existence of solutions for S. If S has already been saved in the table before $\neg g$ comes, no auxiliary resolution tree is needed. Otherwise a new resolution tree T' is initialized. The root of T' is the solution goal g. The old resolution tree, let us call it T, is the caller tree of T'. The algorithm resolves g first and then goes back to the node in T beginning with $\neg g$ and processes it. Compare Definition 73.

If there is no solution for g and at least one of the nodes in T' is suspended, the node in T beginning with $\neg g$ is also suspended (compare the second condition of Definition 64).

If there is at least one solution for g, $\neg g$ in T is *false* and terminated (compare the first condition of Definition 68). If there are no solutions for g and no hope to find a solution for it later, $\neg g$ is interpreted to be *true* (see Case 3 of Definition 59) and terminates (compare the second and the third conditions of Definition 68). If there is no other look-up goal in T waiting for any solution goal in T', the complete resolution of all queries in T' is irrelevant for the initial query and is omitted. The following example illustrates this abstract description.

Let us resolve the query *Acquaintance[bob, n[var D]]* w.r.t. the program G. The main resolution tree, we call it T_0, is depicted in Figure 4.12. There are three auxiliary resolution trees. They are T_1, T_2 and T_3. See Figure 4.13. The main and auxiliary resolution trees build a resolution forest.

Definition 77. (Resolution forest).
The set of the main and auxiliary resolution trees of a query q w.r.t. a program P is called a resolution tree w.r.t. q and P.

T_1 is not a relevant resolution tree any more because the negative look-up goals of $Node_4$ and $Node_{18}$ in T_0 have terminated. That is why T_0 does not depend on T_1 any more. But the algorithm does not delete T_1 because it can again become relevant for the initial query. T_2 and T_3 are still relevant trees. T_0 depends on them.

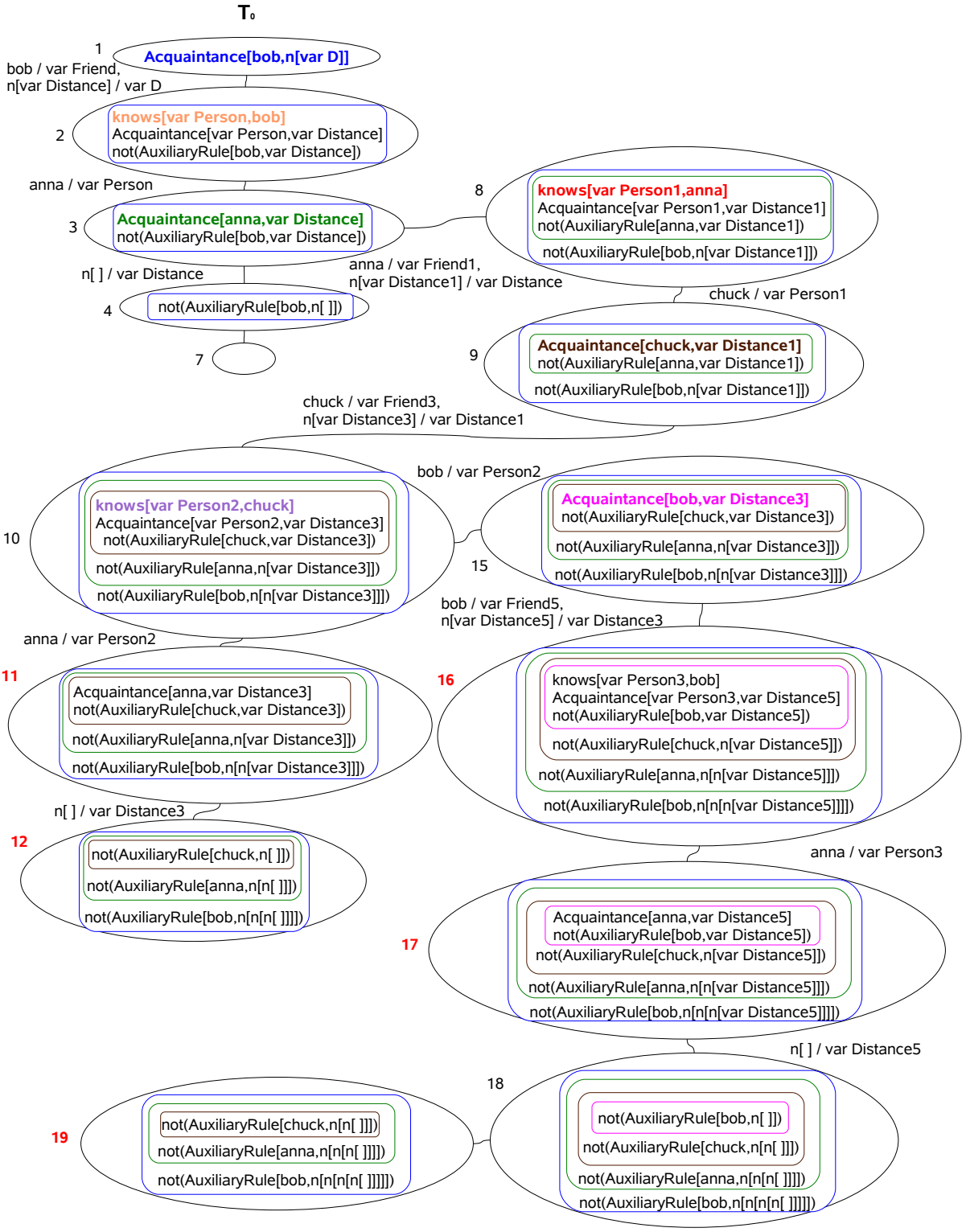

Figure 4.12: The main resolution tree for the query *Acquaintance[bob, n[var D]]* before the evaluation of suspended nodes

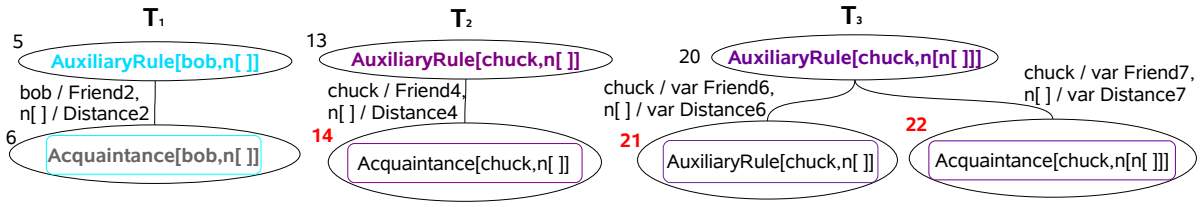

Figure 4.13: Auxiliary resolution trees for the query *Acquaintance[bob, n[var D]]* before the evaluation of suspended nodes

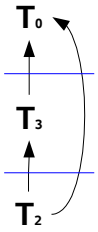

Figure 4.14: Dependencies between the resolution trees before the evaluation of suspended nodes

The only solution goal depending on itself is the first solution goal. Its subtree consists of T_0, T_2 and T_3. Note that T_3 depends on T_2. T_2 does not depend on any other auxiliary resolution tree. One calles T_2 the resolution tree at the lowest level.

Definition 78. (Resolution tree at the lowest level).
A relevant resolution tree which depends on no auxiliary tree containing suspended nodes is the *resolution tree at the lowest level*. If there is no such a resolution tree, there is a cyclic dependency between some auxiliary trees $T_1, ..., T_n$. In such a case the trees $T_1, ..., T_n$ belong to the lowest level.

The suspended nodes within the subtree of the blue solution goal are evaluated in the order specified by Definition 71. $Node_{16}$ is evaluated first because the solution goal of the first goal of the node is terminated. Afterwards the suspended nodes of the resolution trees at the lowest level are resolved. Figure 4.14 illustrates the dependencies between the trees T_0, T_2 and T_3 where the black arrows depict the dependency direction and the blue lines mark the levels. If the algorithm did not distinguish the dependency levels of the resolution trees, it could evaluate $Node_{21}$ in T_3 before $Node_{14}$ in T_2. In this case the first goal of $Node_{21}$ would be *false* though it is still possible to find a solution for its respective solution goal in T_2.

The algorithm resolves $Node_{14}$ and traverses up the tree T_2 searching for terminating solution goals or solution goals depending on itself. The solution goal of $Node_{13}$ terminates. Since $Node_{13}$ is not the root of the main resolution tree, SBR goes back to the caller tree

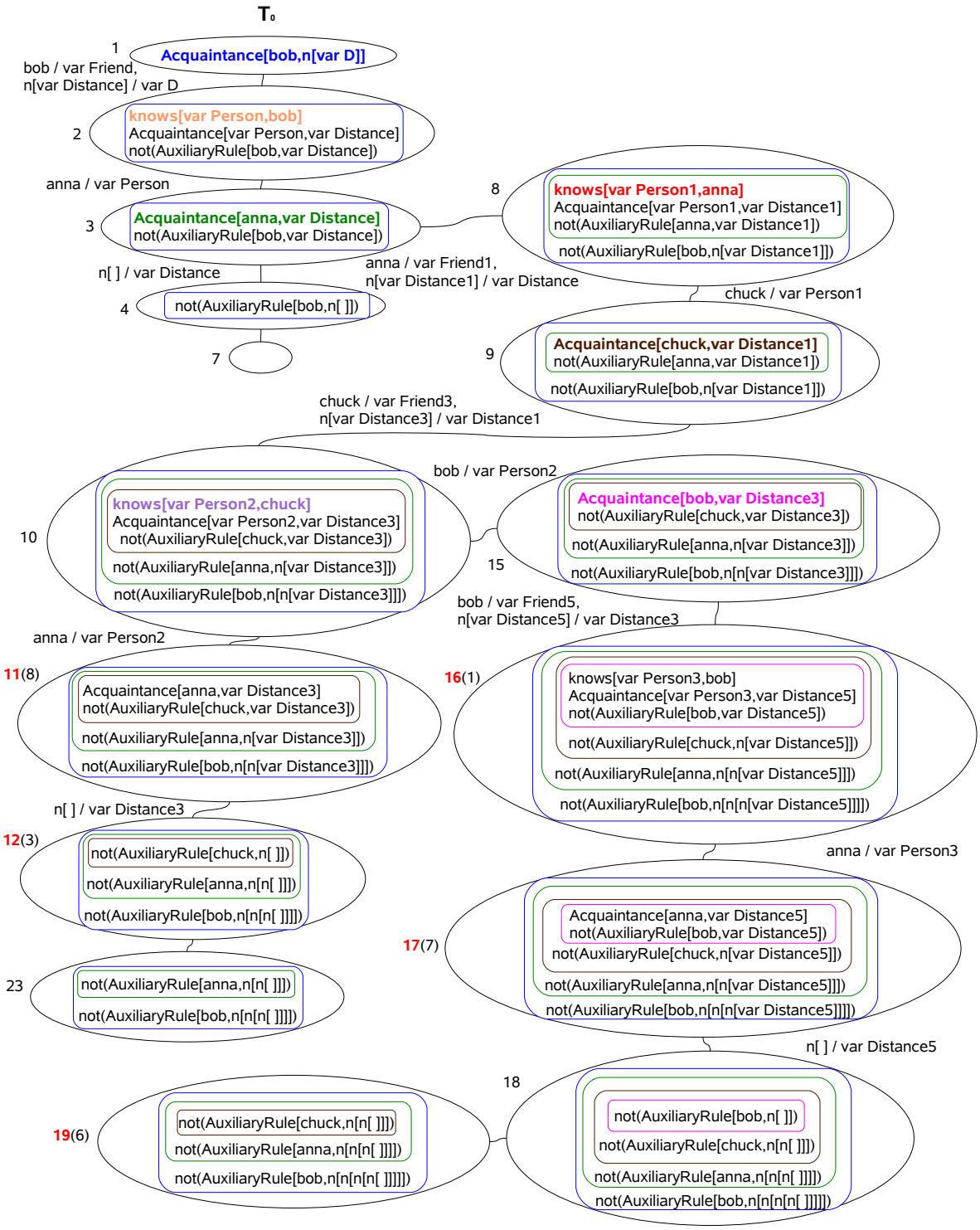

Figure 4.15: The main resolution tree for the query *Acquaintance[bob, n[var D]]*

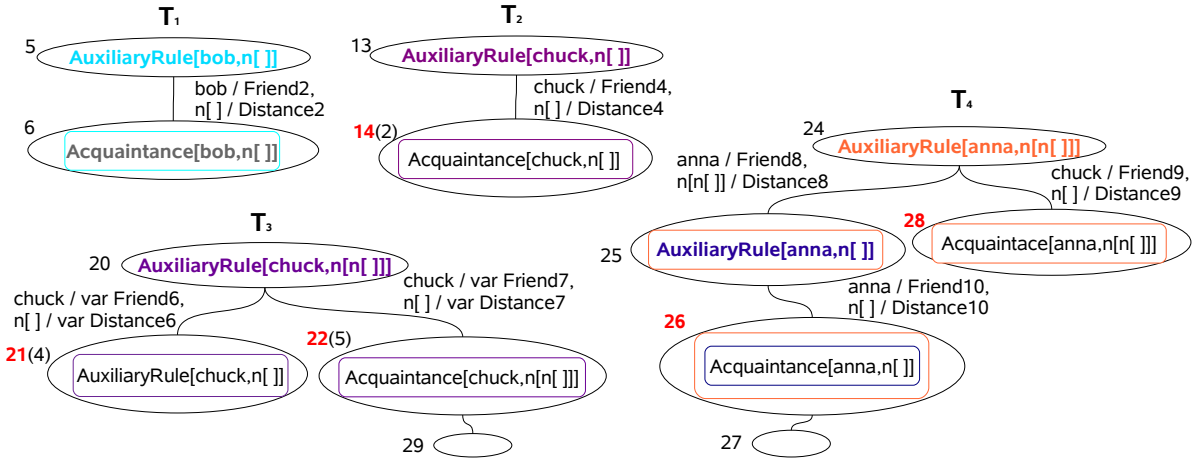

Figure 4.16: Auxiliary resolution trees for the query *Acquaintance[bob, n[var D]]*

of T_2, searches for a non-terminated look-up goal waiting for any solution goal of T_2 and resolves the node beginning with this look-up goal. It is $Node_{12}$.

$Node_{23}$ is the resolvent of $Node_{12}$. Its resolution also requires an auxiliary tree, let us call it T_4. See Figures 4.15 and 4.16. Please notice that T_4 is not completely evaluated since the first goal of $Node_{23}$ terminates and there is no other non-terminated look-up goal in the relevant trees (they are T_0, T_2 and T_3) waiting for any solution goal of T_4, therefore T_4 is an irrelevant resolution tree. If there were such a look-up goal the respective solution goal in T_4 would be completely evaluated.

SBR traverses up the tree T_0 searching for solution goals which can terminate now or a solution goal depending on itself. After the evaluation of every selected suspended node backtracking is indispensable because of the following two reasons:

1. A solution goal must terminate as soon as possible because the evaluation of its look-up goals depends on the state of the solution goal.

2. A solution goal depending on itself must be computed anew because it is possible that the previous one does not depend on itself any more.

In our case no solution goal terminates, the solution goal depending on itself is the blue goal and $Node_{21}$ is the selected suspended node of its subtree since the solution goal of the first goal of the node is terminated. The only look-up goal of the node is *false* and terminates. Backtracking does not change anything in the resolution trees T_3 and T_0.

Since T_2 has no suspended nodes any more, T_3 is the resolution tree at the lowest level of the subtree of the blue goal now and $Node_{22}$ is the next selected suspended node. After its resolution the lilac solution goal of $Node_{20}$ terminates. The first look-up goal of $Node_{19}$ in T_0 is *false* and terminates. Backtracking does not change anything in T_0.

The selected suspended node of the subtree of the blue goal is computed now. Since there are no relevant auxiliary trees any more, the last two suspended nodes of the main tree are evaluated. They are $Node_{11}$ and $Node_{17}$. The order of their resolution is not important. After the evaluation of $Node_{17}$ the pink solution goal of $Node_{15}$ terminates. After $Node_{11}$ is also evaluated, the brown, green and blue solution goals terminate. SBR returns the only solution *Acquaintance[bob, n[n[ ]]]* for the initial query.

In general if there are no (more) suspended nodes satisfying the first condition of Definition 71, SBR must start the evaluation of suspended nodes with a node beginning with a look-up goal g which belongs to a lower stratum of local stratification than the other look-up goals of suspended nodes. Because if g belongs to the stratum S_i, the result of the evaluation of g will be used by the derivation of goals in higher strata than S_i. That is why the nodes containing a most specific goal are evaluated first. In particular if for the derivation of a goal $\neg q$ an auxiliary resolution tree *T'* is needed, the ground instances goals of *T'* are always in lower strata than the ground instances of q. Therefore SBR starts the evaluation of suspended nodes with a node containing a most specific goal of an auxiliary tree at the lowest level.

4.4 Negation and the Evaluation Order of Suspended Nodes

Determination of the evaluation order of suspended nodes is one of the most complex actions of SBR. There are five conditions effecting on this order (compare Definition 71). The necessity of the first and the second conditions is illustrated by an example in Section 4.3. Subsection 4.1.5 is devoted to the third and the fourth conditions. In this section we explain the last condition which selects a suspended node containing the only negative goal in a list, and makes SBR more efficient.

Assume that after the application of the first four conditions of Definition 71 to suspended nodes SBR has a choice between two suspended nodes $Node_m$ and $Node_k$. See Figure 4.17. $Node_m$ contains the only negative goal $\neg X$ in the blue list. $Node_m$ must be processed before $Node_k$ because $\neg X$ can be interpreted to be *true*, in this case the blue list becomes empty and a solution for the blue solution goal is found. This solution can be used to prove some other look-up goals, for instance Y of $Node_k$.

If the SBR algorithm evaluated $Node_k$ before $Node_m$ the positive look-up goal Y of $Node_k$ could be interpreted to be *false* and terminate though there is a solution for it. Recall that after all suspended nodes are evaluated the algorithm tests whether there are some look-up goals in the resolution tree(s) which have not used all solutions for the

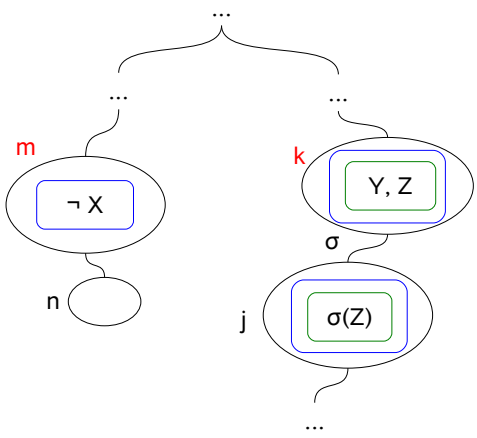

Figure 4.17: Suspended node containing the only negative goal in a list

respective solution goals. If the algorithm finds such a look-up goal it makes all resolution steps possible for the node beginning with this look-up goal. Afterwards the algorithm searches for non-completed look-up goals once again. The algorithm stops only if it does not find such a goal. Therefore $Node_j$ is derived anyway. But it is still better to save a solution for a solution goal and to use it for the resolution of look-up goals as soon as possible because the search for non-completed look-up goals is very expensive and should be processed at best only once at the end of the evaluation. Because of this reason there is the fifth condition in Definition 71.

4.5 Substitution Set

Recall that the main difference between rich unification and standard (poor) unification is the existance of many non-equivalent most general unifiers of a query term and a data term. In this section we demonstrate with an example what does it mean in practice.

Let us evaluate the query q = *AuxiliaryRule[[n[n[n[ ]]]]]* w.r.t. the *Xcerpt*^{s} program G in Listing 4.2, i.e. we are looking for all persons reachable from the start node *Anna* through at least three edges. There are two rules in G such that q simulates into the head of the rule. Each of these simulations returns a substitution set consisting of two substitutions, namely:
AuxiliaryRule[[n[n[n[ ]]]]$\preceq$*AuxiliaryRule[var Friend, n[var Distance]]* with
{ {n[n[ ]]/Distance}, {n[n[n[ ]]]/Friend} } and
AuxiliaryRule[[n[n[n[ ]]]]$\preceq$*AuxiliaryRule[var Friend, var Distance]* with
{ {n[n[n[ ]]]/Friend}, {n[n[n[ ]]]/Distance} }.

Therefore four resolution steps are possible for the first node of the resolution tree in Figure 4.18 depicting the evaluation of q. That is why for rule languages with rich unification the number of resolvents of a query beginning with a goal g depends not only on the number of rules R such that g simulates into the head of R (if g is a solution goal) or the number of solutions C for the solution goal of g (if g is a look-up goal) but also on the number of substitutions yield by the simulation of g into the head of R or g into C.

Actually we see at the very beginning that the resolution steps using the substitution *{n[n[n[ ]]]/Friend}* can never lead to a resolution of any solution goal because the variable *Friend* should be bound to a person name. But thanks to the consideration of rich subsumption between goals the respective branches of the tree are cut very soon.

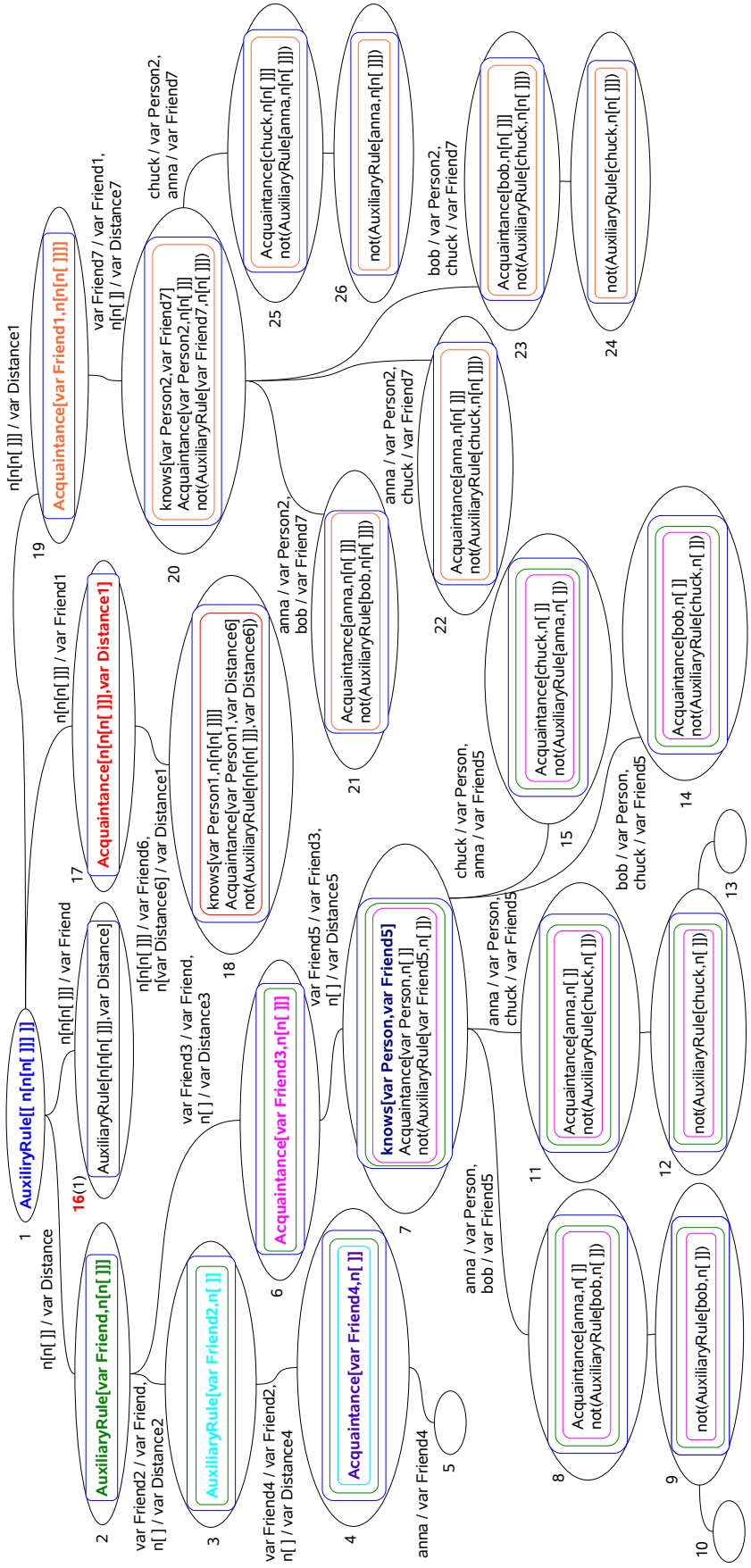

Figure 4.18: Resolution tree for the query *AuxiliaryRule[[n[n[n[ ]]]]]*

Chapter 5

Properties of SBR

In this chapter we show that subsumption-based resolution of a query w.r.t. a locally stratified program with the bounded-term-size property, terminates and returns a complete and sound answer set for the query or 'false' if there are no solutions for it. In Subsection 5.1 we explain the conditions which must be fulfilled by a program, and their necessity. We assume that the notions of *well-founded model* and *local stratification* of a program are known. Otherwise see [4].

5.1 Program Features

The respective program P must fulfil the following indispensable conditions for the termination, soundness and completeness of SBR:

1. P must be locally stratified because SBR is not able to detect and deal with cycles through negation.

2. P must have the bounded-term-size property. The property implies that the size of atoms derivable w.r.t. P and an arbitrary query, is bound, i.e. the set of derivable atoms is finite. Without this property SBR does not always terminate.

Since rule languages are rather different, the bounded-term-size property must be defined for each of them extra.

Definition 79. (**Bounded-term-size property in Prolog**, adopted from [8]).
The *size* of a term is defined recursively as follows:

- The size of a variable or a constant is 1.

- The size of a compound term $f(t_1, ..., t_k)$ is one plus the sum of the sizes of its arguments.

A finite program P has the *bounded-term-size property* if there is a function $f : \mathbb{N} \to \mathbb{N}$ such that whenever a finite set Q of atoms has no atom whose argument size exceeds n, no atom derivable from Q w.r.t. P, has an argument whose size exceeds *f(n)*, $n \in \mathbb{N}$.

Definition 80. (Bounded-term-size property in Xcerpt^s).
The *size* of a term is defined recursively as follows:

- The size of a variable, constant, sum, difference, expression with $\leq$- or $>$-symbol is 1.
- The size of compound terms $f[t_1, ..., t_k]$ and $f[[t_1, ..., t_k]]$ is one plus the sum of the sizes of its arguments.

A finite program P has the *bounded-term-size property* if there is a function $f : \mathbb{N} \to \mathbb{N}$ such that whenever a finite set Q of atoms has no atom whose argument size exceeds n, no atom derivable from Q w.r.t. P, has an argument whose size exceeds *f(n)*, $n \in \mathbb{N}$.

As there are no function symbols in SPARQL, each SPARQL program fulfils the bounded-term-size property, i.e. SBR always terminates w.r.t. a SPARQL program.

5.2 Preliminary Ideas

Let Q be an arbitrary query in a resolution tree built by SBR. The following cases are possible:

1. If Q is a look-up goal it is resolved using the solutions for its respective solution goal S. Therefore it is enough to prove the soundness, completeness and finiteness of the answer set of S to show that the resolution of Q is also sound, complete and finite.
2. If Q is a conjunction of query terms $q_1 \wedge ... \wedge q_n \wedge \neg q_{n+1} \wedge ... \wedge \neg q_{n+m}$ an answer for Q is a conjunction of answers for each $q_i,\ 1 \leq i \leq n + m,\ i, n, m \in \mathbb{N}$, returned or used by a resolution of Q (compare Definition 74). If each of the answers for q_i is sound, complete and finite the resolution of Q are also sound, complete and finite.

Hence, we only prove that each derivation of a solution goal Q computed by SBR, is sound, complete and terminating.

5.3 Soundness, Completeness, Termination and Complexity of SBR

Let Q be a solution goal. Recall that a solution goal is always positive. Let P be a locally stratified program with the bounded-term-size property. A solution C for Q w.r.t. P is sound iff $lfp(\mathbf{W}_P) \models C \Leftrightarrow C \in pos(lfp(\mathbf{W}_P))$ *or* $\exists d \in \mathcal{D}$ *so that* C *unifies with* d *and* $lfp(\mathbf{W}_P) \models d$, where $\mathcal{D} \subseteq \mathcal{HB}$ is a set of all data terms of the respective Herbrand Base $\mathcal{HB}$.

Lemma 1. ***(Soundness of SBR).***
Every solution computed by the subsumption-based resolution of Q w.r.t. P, is sound.

Proof. (By induction on length n of resolution.)

Assume the length of a resolution of Q is 1, i.e. there is a fact F in P such that $Q \preceq F$. Then F is an answer for Q returned by SBR and it holds that $lfp(\mathbf{W}_P) \models F$.

Induction hypothesis: Each subsumption-based resolution of length n of Q w.r.t. P computes a sound solution for Q.

Assume, the length of a resolution of Q is n+1. There is a rule of the form $H \leftarrow L_1 \wedge ... \wedge L_k$ in P such that $Q \preceq H$ using a most general unifier σ_1 and a variable renaming ν_1. $Q' = \sigma_1(\nu_1(L_1)) \wedge ... \wedge \sigma_1(\nu_1(L_k))$ is the resolvent of Q. Since the resolution of Q is n+1 steps long, the resolution of Q' is n steps long and by induction hypothesis, SBR computes a sound solution for each solution goal $\sigma_1(\nu_1(L_i)), 1 \leq i \leq k$, with other words $lfp(\mathbf{W}_P) \models \sigma_n(\nu_n(...(\sigma_1(\nu_1(L_1)))...)) \wedge ... \wedge \sigma_n(\nu_n(...(\sigma_1(\nu_1(L_k)))...))$. Therefore $lfp(\mathbf{W}_P) \models \sigma_n(\nu_n(...(\sigma_1(\nu_1(H)))...))$ which is a solution for Q returned by SBR. □

Lemma 2. ***(Completeness of SBR).***
If there is a sound solution C for Q w.r.t. P then there is a subsumption-based resolution of Q w.r.t. P computing C.

Proof. (By induction on step number n of the operator $\boldsymbol{W}_P$.)

Assume $\mathbf{W}_P^1(\varnothing) \models C$, i.e. C is a fact of P and Q is resolvable in one step.

Induction hypothesis: If $\mathbf{W}_P^n(\varnothing) \models C$ there is a resolution of Q w.r.t. P of length n computing C.

Assume $\mathbf{W}_P^{n+1}(\varnothing) \models C$. Then there is a ground rule instance of the form $C \leftarrow L_1 \wedge ... \wedge L_k$ in the Herbrand instantiation of P such that $\mathbf{W}_P^n(\varnothing) \models L_1 \wedge ... \wedge L_k$. By induction hypothesis, there is a resolution of $L_1 \wedge ... \wedge L_k$ of length n. Therefore the resolution of C is of length n+1. □

Let $|P|$ be the number of rules in P, Π_P the maximum number of literals in a rule body in P, $\mathcal{N}(s),\ s \in \mathbb{N}$, the number of solution goals with argument size $\leq s$ and $|mgu|$ is the maximum number of non-equivalent most general unifiers yield by the unification of a solution goal with the head of a rule in P.

Lemma 3. ***(Termination and complexity of SBR).***
SBR constructs the complete resolution forest w.r.t. an arbitrary query and P in $O(\ \mathcal{N}(s) \times |P|^{|mgu|} \times \mathcal{N}(s)^{\Pi_P}\ )$ *resolution steps.*

Proof. Let n be the maximum size of arguments in an initial query. Let T be a resolution forest for the initial query constructed by SBR. By Definition 79, no goal of T has an argument whose size exceeds *f(n)* in size for some function *f*. Let $s = f(n)$.
The number of solution goals in T is bounded by $\mathcal{N}(s)$. For each node of T beginning with a solution goal Q, the number of resolution steps is bounded by $|P|^{|mgu|}$. The number of solution goals $g \in dep(Q)$ is bounded by Π_P. The number of solution goals in the subtree of each g is less than $\mathcal{N}(s)$. Therefore the number of nodes in T is bounded by $O(\ \mathcal{N}(s) \times |P|^{|mgu|} \times \mathcal{N}(s)^{\Pi_P}\ )$. As each resolution step increases the size of T, the complete resolution forest can be constructed in $O(\ \mathcal{N}(s) \times |P|^{|mgu|} \times \mathcal{N}(s)^{\Pi_P}\ )$ resolution steps. □

The complexity of SBR is compared with the complexity of some resolution algorithms for logic programming languages, such as SLG- and SLT-resolution, in Chapter 6.

Theorem 4. ***(Result of SBR).***
Subsumption-based resolution of Q w.r.t. P terminates and returns all sound solutions for Q or 'false' if there are no solutions for Q.

Proof. By Lemma 3, SBR always terminates w.r.t. P. Since P is locally stratified its well-founded model $lfp(\mathbf{W}_P)$ is total (see [4] for the proof), i.e. each ground atom of the Herbrand base is either *true* or *false* in the well-founded model of P. There is no ground atom which is not defined in $lfp(\mathbf{W}_P)$. Since SBR ascribes the evaluation of Q to the evaluation of its ground instances, SBR always determinates the truth value of Q. If there are some sound solutions for Q, SBR returns all of them since SBR is sound (Lemma 1) and complete (Lemma 2). If there are no solutions for Q, SBR returns *false*. □

Chapter 6

Related Work

In this chapter we analyse the differences between SBR and some of the existing evaluation strategies for logic programming languages depicted in Figure 6.1, and explain why we have chosen the locally stratified SLG-resolution as basis for SBR.

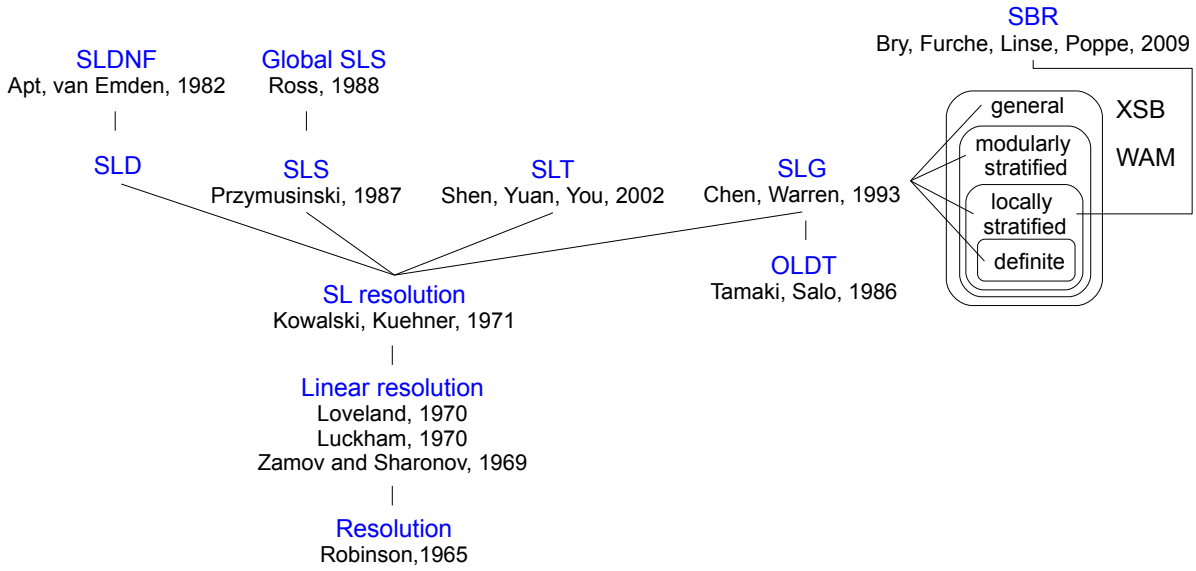

Figure 6.1: SBR and some evaluation strategies for logic programming languages

Resolution was introduced by John Alan Robinson in [22] in 1965. Today it underlies most automatic theorem proving and logic programming systems. In propositional logic the resolution rule is an inference rule that produces a new clause implied by two clauses containing a pair of complementary literals. The resulting clause contains all the literals of the initial clauses except for the pair of complementary literals. Consider the simple example in Figure 6.2. In words: if a is true then c must be true, if a is false then b must be true, with other words b or c must be true. That is why one can reduce the proof of

$(a \vee b) \wedge (\neg a \vee c)$ to the proof of $b \vee c$.

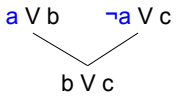

Figure 6.2: Inference rule

Resolution systems are based on refutation: to prove that a propositional formula P follows from a set of formulae S, we assume $\neg P$ and try to derive a contradiction of each derivation of $\neg P$ w.r.t. S. In other words, we try to show that the set $S \cup \{\neg P\}$ is unsatisfiable. Let $P = \neg p \wedge \neg q$ and $S = \{\ \neg p,\ \neg q \vee r,\ \neg r \vee s,\ \neg r \vee \neg s \vee t,\ p \vee \neg t\ \}$. Figure 6.3 depicts a refutation of the set $S \cup \{\neg P\}$. All resolution steps relevant for the refutation are coloured in blue. As this example shows, each initial (i.e. a clause in $S \cup \{\neg P\}$) or derived clause is resolved upon each literal using all applicable initial or derived clauses. Many redundant and irrelevant clauses are derived in this way. They make resolution extremely inefficient. That is why a restriction of the search space of the algorithm is needed.

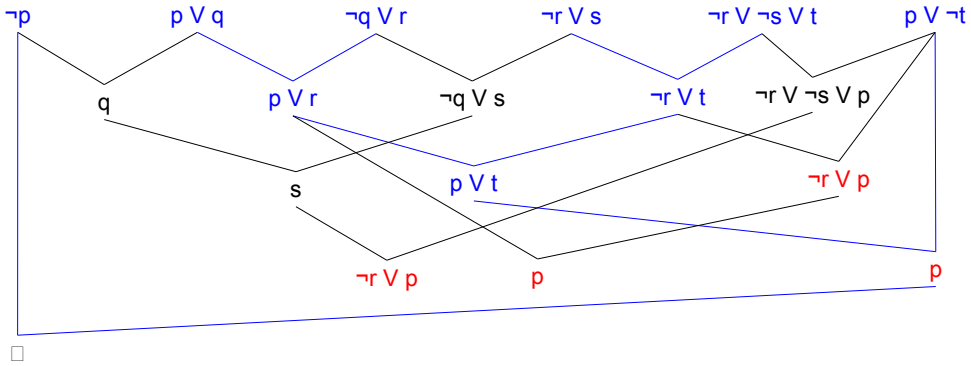

Figure 6.3: Refutation of the set $S \cup \{\neg P\}$

'The inference system of a proof procedure consists of axioms and rules of inference which determinate for a given theorem the *search space* of all derivations obtainable from the axioms by means of the inference rules. One inference system is a *refinement* of another, if the search space of the first is contained in the search space of the second' [12]. *Linear resolution* which was independently discovered by Loveland [13], Luckham [14], and Zamov and Sharonov [29] about 1970, is a refinement of resolution. It restricts a resolution proof to a linear sequence of clauses without decreasing its power.

Let us refute $\neg P$ w.r.t. S by means of linear resolution. See Figure 6.4. As this example shows, during the first resolution step $\neg P$ is resolved upon an arbitrary literal using any applicable clause in S. In each following resolution step the result of the previous

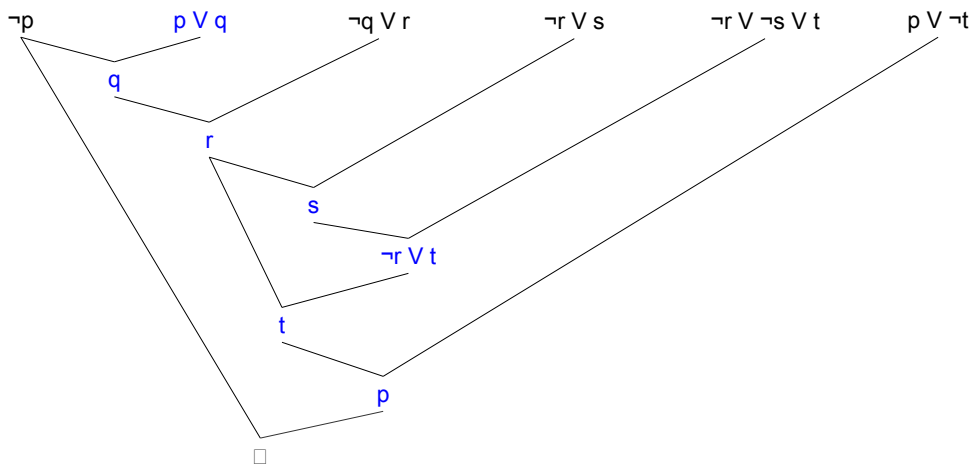

Figure 6.4: Linear refutation of $\neg P$ w.r.t. S

resolution is resolved upon an arbitrary literal using any applicable initial or derived clause. We colour the clauses which can not be arbitrary chosen, in blue. Thanks to this determination of one of the clauses of each resolution step, linear resolution significatly reduces the number of redundancies derivable and is therefore a more efficient form of resolution. The complete linear resolution tree of $\neg P$ w.r.t. S is depicted in [12]. It contains 513 nodes.

Kowalski and Kuehner suggested a restricted form of linear resolution, called *Linear resolution with Selection function* (SL-resolution) in [12] in 1971. The main restriction is effected by a selection function which chooses from each clause a single literal to be resolved upon in that clause. The choice of a literal can be constrained by many conditions. Let it be the literal being resolved using a derived clause and if there are no or many such literals in the current clause, the literal having alphabetically greatest atom, is chosen. The complete SL-resolution tree of $\neg P$ w.r.t. S contains only 12 nodes, see Figure 6.5. For the sake of clarity only resolvents are depicted. Hence, SL-resolution achieves a substantial reduction in the generation of redundant and irrelevant derivations of linear resolution.

SL-resolution is both sound and (with an appropriate search strategy) partially complete, i.e. SL-resolution is complete if it terminates. Many other evaluation strategies base on SL-resolution. One of them is *SL-resolution for Definite programs*, in short SLD. There are two disadvantages of SLD-resolution, they are:

1. SLD-resolution allows no negative literals in a rule body.

2. SLD-resolution does not always terminate even w.r.t. programs with a finite model, like the Prolog program T in Listing 6.1. Figure 6.6 depicts the SLD-resolution tree of the program.

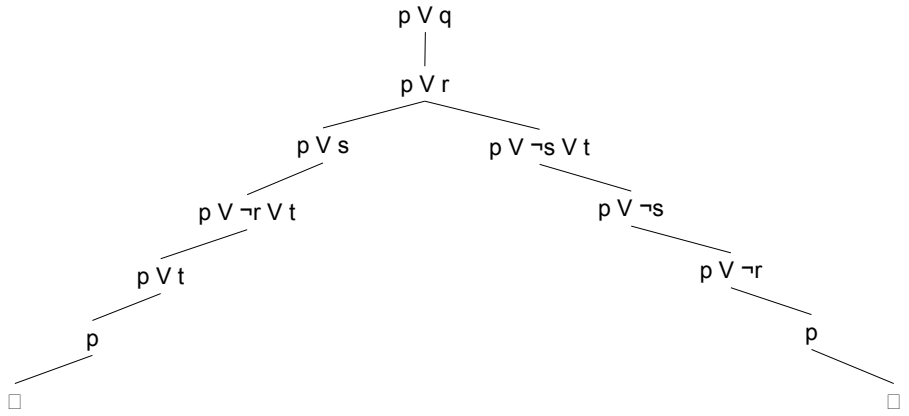

Figure 6.5: SL-resolution tree of $\neg P$ w.r.t. S

Listing 6.1: Prolog program T

```
1 p :- p
2   :- p
```

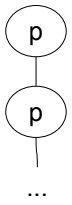

Figure 6.6: SLD-resolution tree for the program T

Apt and van Emden extended SLD-resolution to solve its first problem in [1] in 1982. They called the new strategy *SLD-resolution with Negation as Failure*, in short SLDNF-resolution. The intuition behind negation as failure is quite simple: for a ground atom q it holds that $\neg q$ succeeds if every possible proof of q fails and $\neg q$ fails if q succeeds. A clause can be resolved upon a negative literal only if the literal is ground. The restriction to ground literals was originally introduced to ensure soundness of SLDNF-resolution w.r.t. Clark's completion semantics [9]. Like SLD-resolution, SLDNF does not always terminate. Consider the Prolog program D in Listing 6.2 and its SLDNF-resolution tree in Figure 6.7.

Listing 6.2: Prolog program D

```
1 p :- ¬q
2 q :-  q
3   :-  p
```

Przymusinski suggested *SL-resolution for Stratified programs* (SLS-resolution) in [20] in 1987. The evaluation strategy allows stratified negation and it is sound w.r.t. the per-

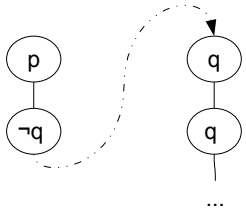

Figure 6.7: SLDNF-resolution tree for the program D

fect model semantics which is well-defined for locally stratified programs (and hence also for stratified programs). In contrast to SLDNF-resolution, SLS-resolution considers infinite branches as *'failed'* rather than *'undefined'*. Because of this property SLS-resolution terminates w.r.t. many programs w.r.t. which SLDNF-resolution does not terminate. For instance, because of the infinite tree under the goal q in the above example, SLS-resolution considers q as failed. Therefore p succeeds and the algorithm terminates. See Figure 6.8 for the SLS-resolution tree for the program D.

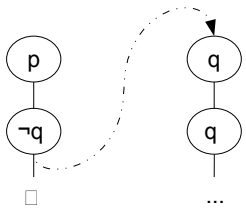

Figure 6.8: SLS-resolution tree for the program D

But unfortunately, this property is not always sufficient for SLS-resolution to determinate the truth value of a goal. Consider the Prolog program R in Listing 6.3 and its respective SLS-resolution tree in Figure 6.9. The left most branch of the tree is infinite and the goal *q(X)* is considered to be false. Like in SLDNF-resolution, a clause can be resolved upon a nagative literal only if the literal is ground. Therefore the clause $\neg q($ *X* $)$ cannot be resolved, SLS-resolution terminates though the truth value of the goal *p(X)* remains unknown.

Listing 6.3: Prolog program R

```
1 p( X ) :-  q( X )
2 p( X ) :- ¬q( X )
3 q( X ) :-  q( X )
4        :-  p( X )
```

Ross suggested *Gobal SLS-resolution* extending Przymusinski's SLS-resolution, in [23] in 1988. In contrast to SLS-resolution, global SLS-resolution may be applied to all programs, whether locally stratified or not. Like SLS-resolution, global SLS-resolution

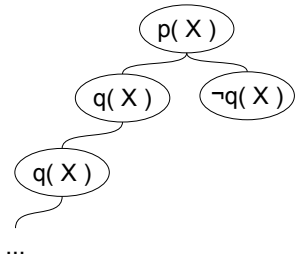

Figure 6.9: SLS-resolution tree for the program R

Solution goal	State	Solutions
$q(\ X\ )$	completed	q(a)
$p(\ X\ )$	completed	p(a)

Table 6.1: Table of solution goals for the OLDT-resolution of M

treats infinite derivations as *'failed'*. Besides, recursion through negation is allowed and treated as *'undefined'*. Global SLS-resolution is sound w.r.t. the well-founded semantics. However, it inherits from SLDNF- and SLS-resolution the problem of infinite loops (and therefore, non-termination) and redundant computations.

To guarantee the termination of top-down computation and to avoid redundant evaluations *tabling* has been proposed. The idea of tabling is to keep a table of goals and their solutions that have been computed. If a current goal is subsumed by a previous one, instead of being resolved using the rules of the program, the current goal is solved using solutions computed for the previous goal. The goals saved in the table are called *solution goals*. A goal which is subsumed by a solution goal and resolved upon its answers, is a *look-up goal*. *OLD-resolution with Tabulation* (OLDT-resolution) suggested by Tamaki and Sato in [28] in 1986, is based on this principle. OLDT-resolution is sound, terminating for programs with a finite model and patially complete if an appropriate search strategy is used. But there are two disadvantages of OLDT-resolution, namely:

1. OLDT-resolution allows no negative literals in a rule body.
2. The approach is not linear as the Prolog program M in Listing 6.4, its respective OLDT-resolution tree in Figure 6.10 and Table 6.1 show.

Listing 6.4: Prolog program M

```
1 q( X ) :- p( X )
2 q( a ) :-
3 p( X ) :- q( X )
4        :- q( X )
```

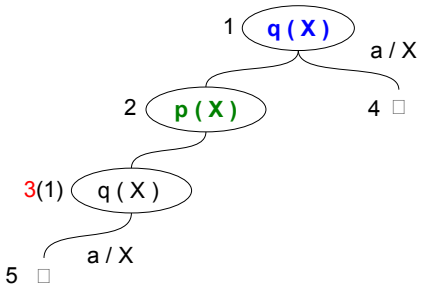

Figure 6.10: OLDT-resolution tree for the program M

Linearity is an advantage of an evaluation strategy because a linear resolution can be implemented using simple and efficient stack-based data structures. For linear resolution only the current branch of the resolution tree must be saved. It is not the case for OLDT-resolution. In the worst case each branch of an OLDT-resolution tree ends with a suspended node and the whole tree must be saved.

To solve the first problem of OLDT-resolution, SLG-resolution was proposed by Chen and Warren in [8] in 1993. SLG-resolution can be characterized by the following features:

1. SLG-resolution resolves queries w.r.t. general logic programs, i.e. the strategy is able to detect and deal with cycles through negation.

2. SLG-resolution is sound and search space partially complete w.r.t. the well-founded partial model.

3. The evaluation strategy consists of seven fundamental transformations which can be applied to transform a query into a set of answers. The result of each transformation is a set of clauses, called a system. Figure 6.11 depicts the system transformations for the program M. See [7] for details.

4. SLG-resolution performs top-down computation, consequenty only relevant answers are computed.

5. SLG-resolution supports answer sharing of subgoals that are variants of each other. A subgoal is guaranteed to be evaluated only once.

6. SLG-resolution terminates for programs with the bounded-term-size property.

7. The algorithm allows maximum freedom in control strategies. A programmer or an implementer can choose an arbitrary computation rule for selecting a literal from the rule body and an arbitrary strategy for selecting which transformation to apply.

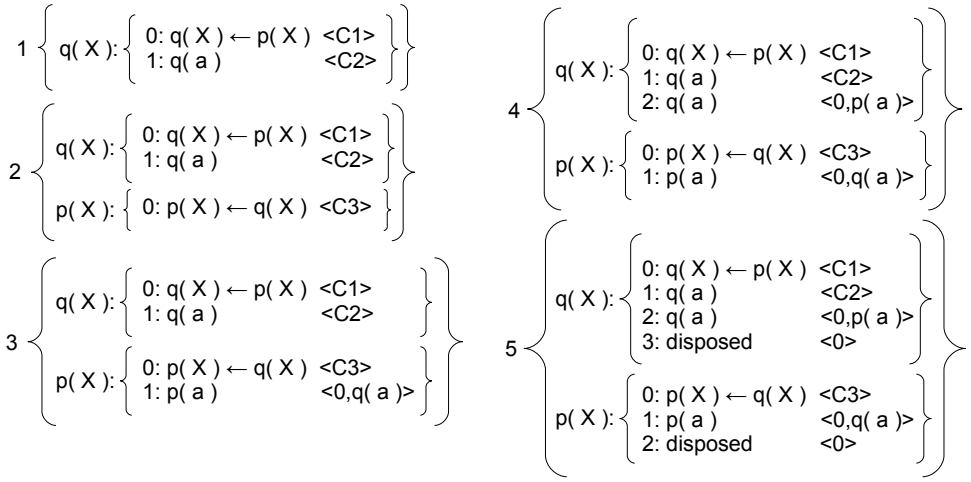

Figure 6.11: SLG-resolution for the program M

Restricted forms of SLG-resolution were identified for definite, locally stratified and modularly stratified programs. Recall that a program is modularly stratified if the resolution of a particular query w.r.t. this program does not contain negative cycles. A program is locally stratified if this statement holds for the resolution of an arbitrary query w.r.t. this program. Consequently, all locally stratified programs are modularly stratified, the converse is not true. Complare Figure 6.1.

Some implementations of SLG-resolution are available. For instance, *SLG-Warren Abstract Machine* (SLG-WAM) described in [27], is the implementation of SLG-resolution for modularly stratified programs using seven stacks. *Extended Stony Brook Prolog* (XSB) is the implementation of SLG-resolution for general programs. It supports multi-treading, indexing, dynamic code, interfaces and HiLog. For details of XSB see [21].

There are two disadvantages of SLG-resolution, they are:

1. Because of the solution-lookup mode the approach is not linear.

2. It uses system transformations which are rather unreadable even for simple definite programs and do not keep the order of query evaluation.

Linear resolution with Selection function and Tabling (SLT-resolution) proposed by Shen, Yuan and You in [26] in 2002, solves both problems of SLG-resolution widely preserving its time complexity (see Table 6.4 for the comparison). SLT-resolution is terminating, sound and complete w.r.t. the well-founded semantics for general programs with the bounded-term-size property. In contrast to SLG-resolution, it uses standard tree-based formulation which is easier to understand and to keep track of the evaluation than system transformations of SLG-resolution. Besides, SLT-resolution resolves infinite loops and redundant computations without sacrificing the linearity of the evaluation. SLT-resolution

Goal	Solutions
q(X)	q(a)
p(X)	p(a)

Table 6.2: Table of goals for the SLT-resolution of M

makes use of tabling but it does not distinguish solution and look-up goals. A node beginning with a goal g is expended by applying existing answers for a variant of g in the table, followed by program clauses which have not been used by the variants of g.

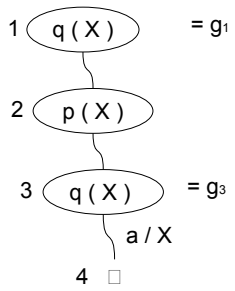

Figure 6.12: SLT-resolution tree for the program M

Consider the SLT-resolution tree for the program M in Figure 6.12. The goal of $Node_3$, let us call it g_3, is a variant of the goal of $Node_1$, let us call it g_1. If there were some solutions for g_1 in the table after the second resolution step, g_3 would use them. All applicable program clauses are used by g_3 except for the program clauses that have been applied to g_1, i.e. the second rule of M is used to expend $Node_3$. Two solutions, namely *q(a)* and *p(a)* are found and saved in the table for the respective goals. See Table 6.2. SLT-resolution moves back to $Node_2$, no further resolution steps are possible for it. The algorithm moves back to $Node_1$. All clauses that have been used by g_3 will no longer be used by g_1, i.e. no rule of M is applicable. SLT-resolution terminates.

Because of linearity SLT-resolution has much better space complexity than SLG-resolution. Besides, SLT-resolution can be implemented as an extension of any existing Prolog abstract machine.

But unfortunately, this idea of SLT-resolution solving the problems of infinite loops, redundant computations and non-linearity, is not applicable to a resolution algorithm for rule languages with rich unification. As an illustration, consider the following example. Let $q_1 = f\{\{\ var\ X\ \}\}$ and $q_2 = f\{\{\ desc\ a\ \}\}$ be two goals so that q_1 appears before q_2 in the evaluation, note that $q_1 \supseteq q_2$. All rules applied to q_2 must be also used by q_1 to avoid the incompleteness of the evaluation. Hence, to many redundant computations will be performed. In contrast, SLG-resolution considers q_1 as a solution goal and q_2 as a look-up goal. The rules of the program are applied only to q_1, i.e. no redundant

computations are performed. Though SLG-resolution is not linear, it is a better basis for an evaluation strategy for rule languages with rich unification than SLT-resolution. We restricted ourselves to locally stratified SLG-resolution because we like the properties of locally stratified programs, especially the totalness of the well-founded model which guarantees that every query can be answered with *yes* or *no*.

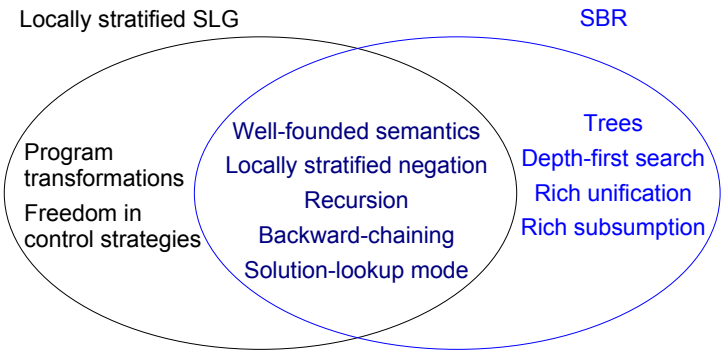

Figure 6.13: Locally stratified SLG-resolution vs. SBR

In particular, SBR maintains the following features of locally stratified SLG-resolution (compare Figure 8.1):

1. SBR is *sound* w.r.t. the well-founded semantics. The well-founded semantics is the most natural semantics for programs with negation because it assigns a unique model to a program which is the minimal model of all three valued models of the program. Two valued semantics, in contrast, run into alternated fixpoints or multiple models for programs with recursion through negation.

2. SBR resolves queries w.r.t. *locally stratified recursive* programs.

3. Because of backward-chaining the algorithm is *goal-directed*, i.e. SBR resolves no queries which are irrelevant for the initial goal.

4. SBR adopts the solution-lookup mode. Consequently, it performs *no redundant computations* but is unfortunately *not linear*.

5. Thanks to the idea of tabling the algorithm *terminates* for programs with the bounded-term-size property.

6. Because of tabling and backtracking SBR is *fair*, i.e. it resolves each query of the resolution tree(s) (if the query is essential for the initial query), and *partially complete*, i.e. complete for terminating query evaluations. SBR returns all answers for the user's query (if the answer set is finite) or 'false' if there are no solutions for it.

There are also some differences between locally stratified SLG-resolution and SBR, they are:

1. For the sake of simplicity SBR does not allow maximum freedom in control strategies in contrast to SLG-resolution. Our algorithm processes *depth-first search*, i.e. it begins the resolution of a query always with its first query term and applies the rules of the program (or the solutions of the respective solution goal) in the order they appear in the program (or in the table). But this limitation does not decrease the expressive power of SBR compared to locally stratified SLG-resolution. Both approaches are Turing-complete.

2. SLG-resolution is based on program transformations which become rather unreadable even for simple definite programs. That is why SBR uses the standard *tree-based formulation* which are easier to understand and to keep track of query evaluation for the reader.

The following advantages of SBR are our contribution:

1. Since rich unification and subsumption are strong means to address variable schemas of different rule languages, SBR is *generic*, i.e. it works for every rule language fulfiling the preconditions of utilisation of SBR (Subsection 7.1.2), particularly unification and subsumption must be implemented for the language.

2. The test whether a term subsumes another term is rather expensive. For instance subsumption between two Xcerpt terms can be decided in $O(\ n!^n\ )$ where n is the sum of the sizes of both query terms [6]. But in most cases this decision is very cheap thanks to the heuristics. For instance $q_1 \not\sqsubseteq q_2$ if q_1 and q_2 have different labels or if q_1 is incomplete and q_2 is complete or if they are both complete and their sizes are not equal.
   Thanks to the consideration of rich subsumption, SBR performes no redundant computations. Due to the amount of information available on the Web repeated evaluations can very much impact the complexity of an algorithm. Hence, in general consideration of rich subsumption of goals contributes to the efficiency of an evaluation strategy.

Table 6.3 compares SBR with the evaluation strategies for logic programming languages considered in this chapter. Only SBR, SLG- and SLT-resolution allow negation and are sound, complete and terminating for programs with the bounded-term-size property. Let us compare the complexity of these strategies. See Table 6.4.

	Resolution	Linear resolution	SL-resolution	SLD	SLDNF	SLS	Global SLS	OLDT	SLG	SLT	SBR
Negation	any	any	any	–	any	stratified	any	–	any	any	locally stratified
Linearity	–	+	+	+	+	+	+	–	–	+	–
Termination	–	–	–	–	–	–	–	+	+	+	+
Partial completeness	+	+	+	+	+	+	+	+	+	+	+
Soundness w.r.t.				fixpoint semantics	completion semantics	perfect model semantics	well-founded semantics	fixpoint semantics	well-founded semantics	well-founded semantics	well-founded semantics

Table 6.3: SBR vs. some evaluation strategies for logic programming languages

	SLG	SLT	SBR
Time complexity	$O(\ \lvert P\rvert\mathcal{N}(s)^{\Pi_P+1}\ )$	$O(\ \lvert P\rvert\mathcal{N}(s)^{\Pi_P+3}\ )$	$O(\ \lvert P\rvert^{\lvert mgu\rvert}\mathcal{N}(s)^{\Pi_P+2}\ RU\ )$
Space complexity	$O(\ \lvert P\rvert\mathcal{N}(s)^{\Pi_P+1}\ )$	$O(\ \mathcal{N}(s)\ )$	$O(\ \lvert P\rvert^{\lvert mgu\rvert}\mathcal{N}(s)^{\Pi_P+1}\ )$

where

P	is a (locally stratified) program with the bounded-term-size property
$\lvert P\rvert$	is the number of rules in P
Π_P	is the maximum number of literals in a rule body in P
s	$\in \mathbb{N}$
$\mathcal{N}(s)$	is the number of distinct atoms (solution goals) with argument size $\leq s$
$\lvert mgu\rvert$	is the maximum number of non-equivalent most general unifiers yield by the unification of a solution goal with the head of a rule in P
RU	is the complexity of rich unification and subsumption of a rule language

Table 6.4: Complexity: SLG vs. SLT vs. SBR

The time complexity of SLT-resolution is worse than the time complexity of SLG-resoloution because for each selected subgoal g SLT-resolution traverses the tree upwards searching for variants of g to skip the looping clauses for g. Since the program clauses are selected in a fixed order, the algorithm has to search only for the closest ancestor variant of g to determinate looping clauses and clauses that have been completely used for variants of g. The number of subgoal comparisons performed during the search for the closest ancestor variants of all selected subgoals in an SLT-tree is $O(\ |P|\mathcal{N}(s)^3\ )$ [26].

There are three points making the time complexity of SBR worse than the time complexity of SLG- and SLT-resolution, they are:

1. Both SLG- and SLT-resolution use a sorted table of solutions. Therefore, the time for searching and inserting a subgoal in the table is $O(\ \log\mathcal{N}(s)\ )$ which is disregarded in O-notation. In contrast, the table used by SBR is unsorted. Thus the time for searching a goal in the table is $O(\ \mathcal{N}(s)\ )$. But the table can be easily optimized.

2. Recall that rich unification of a solution goal g and the head of a rule r in P obtains a substitution set $\{\sigma_1, ..., \sigma_{|mgu|}\}$. Hence, $|mgu|$ resolution steps are possible for the node of g using r and $\sigma_i,\ 1 \leq i \leq |mgu|,\ i, |mgu| \in \mathbb{N}$. But as the example in Section 4.5 shows, the branches of a tree which can never find solutions, are cut very soon.

3. The complexity of standard unification and subsumption is linear in the size of terms s. It is also skipped in O-notation. In contrast, SBR deals with rich unification and subsumption of goals which is rather expensive in worst case. But in many cases the complexity of rich unification and subsumption is also linear thanks to the heuristics. In average case rich subsumption even contributes to the efficiency of SBR. As Table 6.4 shows, the size of a tree (or a system) depends on the number of solution goals (distinct atoms) $\mathcal{N}(s)$ which is reduced by rich subsumption.

While SBR and SLG-resolution save the whole resolution tree or system, SLT-resolution saves only the current branch of a tree and is therefore much more space efficient. All three strategies save a table of solutions which contains exactly $\mathcal{N}(s)$ enties, and is disregarded in O-notation.

The complexity of SLG- and SLT-resolution w.r.t. a function-free program and a function-free query is polynomial since $\mathcal{N}(1)$ is polynomial in the size of the extensional database. SBR, in contrast, is at best exponential in this case because many most general unifiers of a function-free solution goal and a function-free construct term are still possible. There are even function-free rule languages with rich unification, for example SPARQL.

But rich unification and subsumption can make the complexity of SBR w.r.t. a function-free program and query worse than exponential.

Chapter 7

Implementationdetails of SBR

We implemented SBR in Ruby. Section 7.1 describes the implementation abstactly with the help of UML-diagrams. In this section we show that the algorithm is generic, i.e. it works for any rule language fulfilling the precondition of the utilisation of SBR specified in Subsection 7.1.2. In Section 7.2 the core of SBR is illustrated more precisely by means of flow diagrams.

7.1 Genericity of SBR

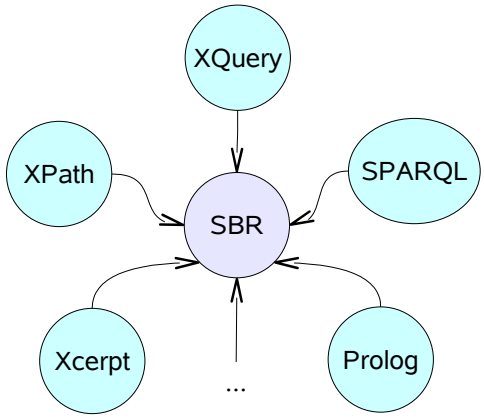

Figure 7.1: Genericity of SBR

In Subsection 7.1.1 the module SBR is introduced. The module encapsulates the implementation of the subsumption-based resolution algorithm. Queries in a rule or query language with rich or poor unification can be resolved by the module if the data model, unification and subsumption are implemented for the language. This language can be Xcerpt, Prolog, SPARQL, Xquery, XPath or any other rule language. See Figure 7.1.

Subsection 7.1.2 presents the preconditions of utilisation of SBR module by a rule language. We have tested the algorithm with two *Xcerpt*^{s} data models. They are described in Subsections 7.1.3 and 7.1.4.

7.1.1 SBR Module

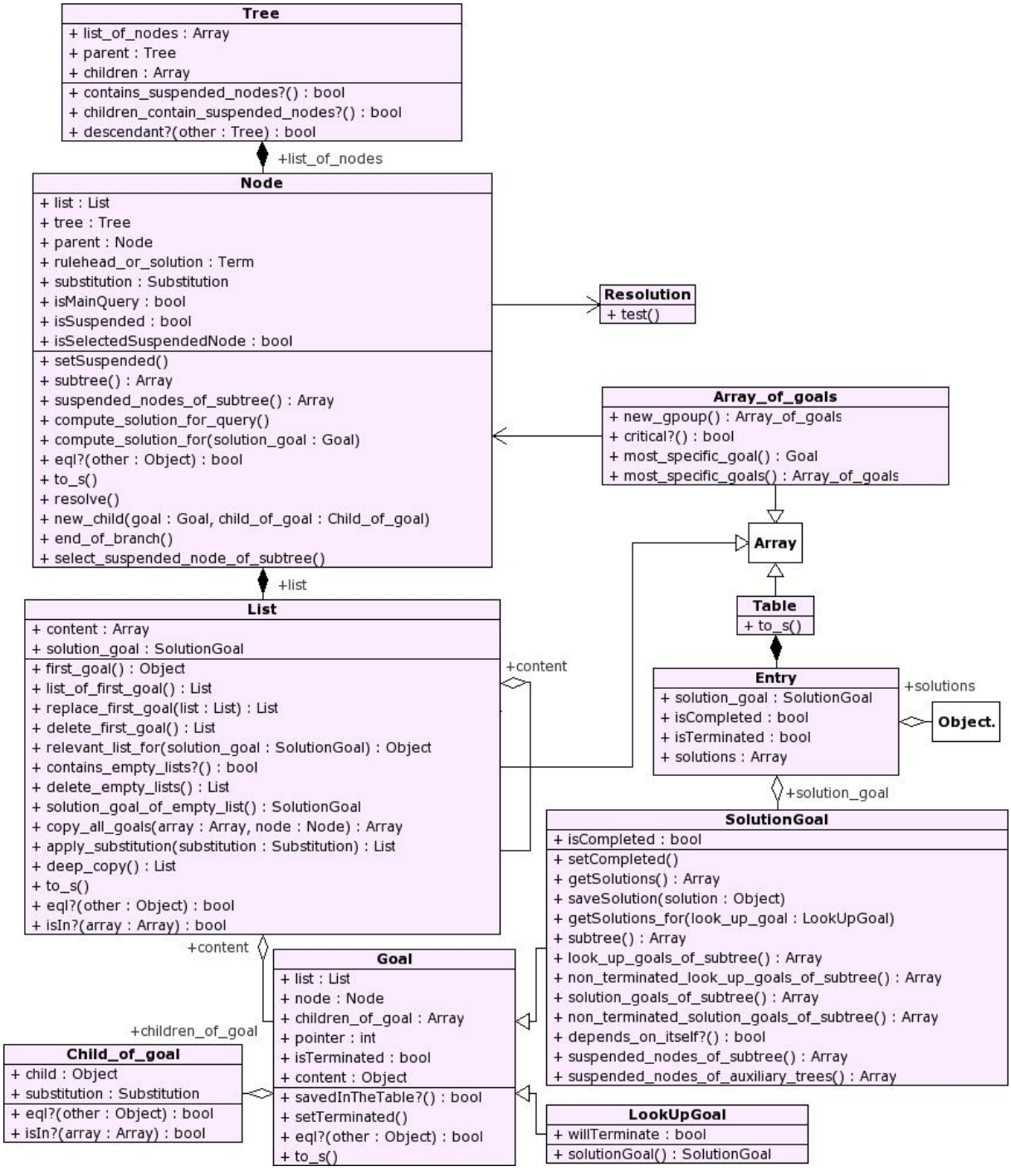

Figure 7.2: UML-diagram of SBR module

The complete UML-diagram of the module is shown in Figure 7.2[1]. SBR manages trees, nodes, lists, goals and the table. As we have already seen in Section 4.3, many resolution trees are sometimes needed for the resolution of a query. Each tree saves a pointer to its caller tree (Definition 73). The algorithm uses the pointer (called *parent* in Figure 7.2) to change between trees.

A tree consists of nodes. Each node saves a pointer to the tree it belongs to and a pointer to its father node. Each node N_2 is the resolvent of its father node N_1 using a rule or a solution and some substitution (Definition 59). The rulehead or the solution and the substitution are saved for N_2 to compute solutions for the first goal of N_1 (Definitions 62 and 74). The value of the boolean variable *isSuspended* describes the state of a node (Definition 64).

Each node contains a list. Each list contains another list and/or goals. The resolution of a list is relevant for some solution goal. Each list saves its respective solution goal. Thanks to this information the algorithm knows for which solution goal a solution is found if all goals of a list are resolved.

A goal can be a term, a string, a number, a variable or whatever the data model of a rule language allows. A goal saves a pointer to the list and the node it appears within. A goal is either a solution goal or a look-up goal. A *child of a solution goal* is a rule such that the solution goal unifies with the head of the rule and a substitution yield by this unification. Since rich unification returns a substitution set, there can be many children of a solution goal with the same rule but different substitutions. A *child of a look-up goal* is a solution of the respective solution goal and a substitution yield by the unification of the look-up goal with the solution. There can be many children of a look-up goal with the same solution but different substitutions. The algorithm makes a resolution step for each child of the first goal of a node. In order to distinguish the children which have already been used to derive a resolvent and other children a pointer is saved. It points to the last used child. The boolean variables *isCompleted* and *isTerminated* describe the state of a goal (Definitions 58, 69, 67 and 68). Each solution goal, its state and solutions are saved in the table.

Array_ of_ goals is an auxiliary class determinating the most specific goal(s) of suspended nodes which is essential for the evaluation order of suspended nodes (Definition 71).

Resolution is also an auxiliary class. It initializes the initial query, the program or data against which the query is to be resolved, creates the root of the main tree, calls the algorithm with the root and puts out the solutions for the initial query or *false* if there

[1]In this and the following UML-diagrams some relations between classes are omitted for the sake of clarity.

are no solutions for it.

The core of SBR are the four methods of the class *Node*. They are:

1. *resolve()*
2. *new_child(goal : Goal, child_of_goal : Child_of_goal)*
3. *end_of_branch()*
4. *select_suspended_node_of_subtree()*

They are explained with the help of flow-diagrams in Section 7.2. All other methods are auxiliary, their explanation is omitted in this work for the sake of brevity.

7.1.2 Preconditions of Utilisation of SBR by a Language

SBR resolves queries w.r.t. a program in a rule language if the language fulfils the following preconditions:

1. *Unification* and *subsumption* are defined and implemented for the language as well as *substitution* since not only variables can be bound (See Subsection 7.1.3).
2. Each class C the instances of which can be goals, implements the following attributes and methods:
   (a) *isNegated : bool* is *true* if the goal is negated, it is *false* otherwise
   (b) *unify(constructentity : Object) : Object* returns a substitution set yield by the unification of the current goal with a *constructentity* or *false* if the unification fails
   (c) *subsume?(queryentity : Object) : bool* returns *true* if the current goal subsumes a *queryentity* or *false* otherwise

   The parameters *queryentity* and *constructentity* in the above methods are of type *Object* because they can be terms, strings, numbers, variables or whatever a language allows.
   (d) *deep_copy() : C* returns a deep copy of the current goal to use the copy in a resolvent
   (e) *apply_substitution(substitution : Substitution) : C* applies a *substitution* to the current goal to use the goal in a resolvent

(f) *variables() : Array* returns an array of variables appearing in the current goal to rename them and to remember the variables which have already been bound

(g) *rename_variable(name : string) : Array* renames a variable *name* in the current goal because *name* has already been bound

(h) *to_s()* for debug outputs

7.1.3 Xcerpt$_1^s$ Module

The complete UML-diagram of the module is shown in Figure 7.3. The Xcerpt^s program P in Listing 4.1 is an example of an Xcerpt$_1^s$ program.

Peculiarities of the Module

There are two peculiarities of $Xcerpt_1^s$ comared to Xcerpt. See Figure 7.4. They are expressions with $\leq$-symbol in query terms and expressions with $>$-symbol in construct terms. As we shall see later, the latter plays an important role only by the computation of simulation, it helps to prevent non termination. But the former must be taken into account both by the computation of simulation and subsumption. Besides, SBR must treat an expression with $\leq$-symbol as a variable. The following example explains why.

Let us evaluate the query term q = *Acquaintance[var P, ≤2]* w.r.t. the program P in Listing 4.1, i.e. we are looking for all persons reachable from the start node *Anna* through not more than 2 edges. During the evaluation of q the resolution tree in Figure 7.5 is built.

SBR treats an expression of the form *≤2* as a variable. It can be bound to a number. This binding is used to compute solutions for a solution goal. For example the algorithm cannot save the term *Acquaintance[bob, ≤2]* as a solution for the blue solution goal. Because if it did so, it would not know whether the look-up goal *Acquaintance[bob, ≤1]* of $Node_8$ is *true* or *false*.

Besides, a substitution of the form { *0/≤1* } is applied to the goals of a resolvent to compute a more specific query but *≤1* is removed not by *0* but by *≤0* because *≤1* of different goals can have different meanings. It is so in our case. For example the first and the second goals of $Node_4$ contain *≤1* but it does not have the same meaning in these goals. Consider the second rule of the program P in Listing 4.1. The goal *Acquaintance[anna, ≤1]* was derived from *Acquaintance[var Person, var Distance]* where *var Distance* was bound to *≤1*. The goal *not(Acquaintance[bob, ≤1])* was derived from *not(Acquaintance[var Friend, ≤var Distance])* where *≤var Distance* was bound to *≤1*. *var Distance* and *≤var Distance* of the initial goals do not have the same meaning. Therefore

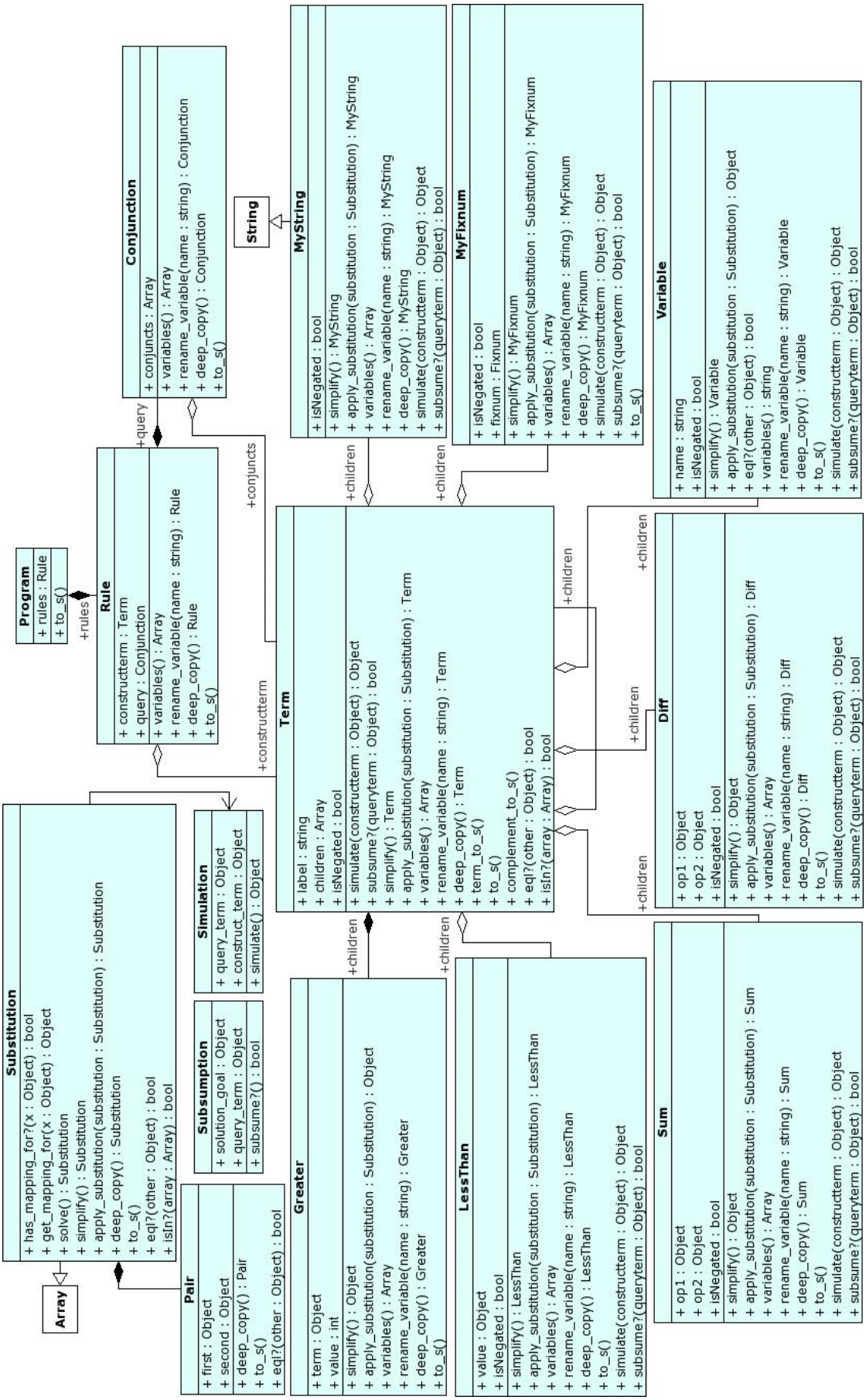

Figure 7.3: UML-diagram of Xcerpt$_1^s$ module

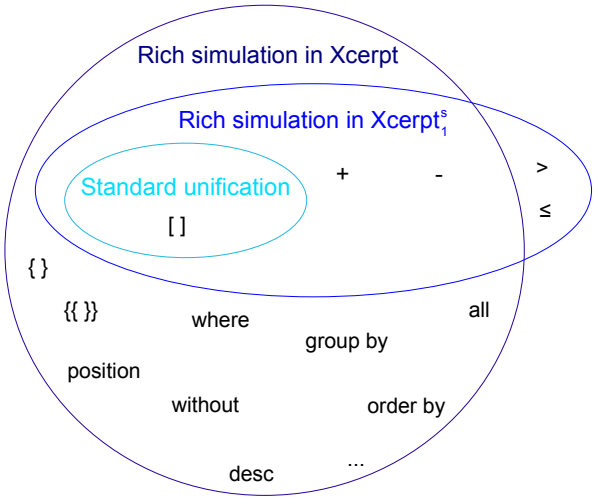

Figure 7.4: Standard unification vs. rich simulation in Xcerpt$^s_1$ vs. rich simulation in Xcerpt

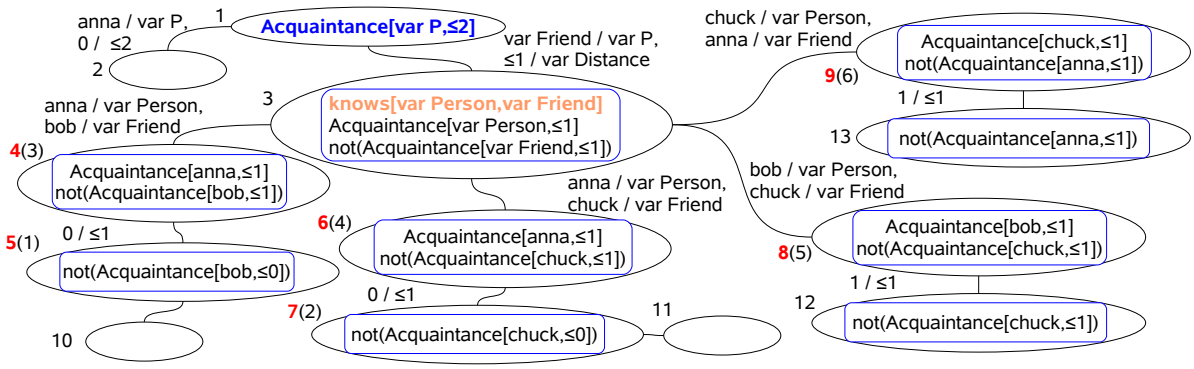

Figure 7.5: Resolution tree for the query *Acquaintance[var P, ≤2]*

≤1 in the derived goals does not have the same meaning as well.

Therefore an Xcerpt$_1^s$ substitution binds not only variables to terms, but also expressions of the form $\leq X$ where X is a number or a variable, to terms.

Another peculiarity of Xcerpt$_1^s$ module is the necessity to simlify a sum or difference in goals and substitutions if possible. For instance, the sum *(var X - 1) + 1* must be simplified to *var X*.

Besides, a substitution must be resolved such that a variable or an expression of the form $\leq X$ where X is a number or a variable, are bound but not a sum, difference or $\leq X$ where X is a sum or difference. For example the substitution {*2/Distance+1*} is resolved to {*1/Distance*} and the substitution {≤ 2*/Distance+1*} is resolved to {≤ 1*/Distance*}.

Xcerpt$_1^s$ Query

Definition 81. (Xcerpt$_1^s$ Query).
An *Xcerpt$_1^s$ query* is defined as follows:

1. Every *string* is a query.
2. Every natural *number* is a query.
3. Every *variable* is a query.
4. Every *sum* of the form $x + y$ where x and y are numbers or variables, is a query.
5. Every *difference* of the form $x - y$ where x and y are numbers or variables, is a query.
6. Every expression of the form $\leq x$ where x is a number, variable, sum or difference, is a query.
7. Every *positive term* of the form $f[x_1, ..., x_n]$ where f is an arbitrary function symbol, $x_1, ..., x_n$ are strings, numbers, variables, sums, differences, expressions with $\leq$-symbol or positive terms and $n \in \mathbb{N}$, is a query.
8. Every *negative term* of the form $\neg f[x_1, ..., x_n]$ where f is an arbitrary function symbol, $x_1, ..., x_n$ are strings, numbers, variables, sums, differences, expressions with $\leq$-symbol or positive terms and $n \in \mathbb{N}$, is a query.
9. Every *conjunction* of queries is a query.

Examples of $Xcerpt_1^s$ queries:

tra-la-la

1000

≤ var X + var Y

Acquaintance[var P, ≤4+1, knows [anna, var X]]

Acquaintance[var P, var D], knows[var P, chuck], not(knows[chuck, var P])

The following queries are not valid for $Xcerpt_1^s$:

knows[[without anna]]

Acquaintance{{ optional var P, not(>4-1-4) }}

Since an Xcerpt$_1^s$ query term is always complete, a substitution set returned by the simulation of a query term into a construct term always contains only one substitution. But the simulation of an Xcerpt$_1^s$ query term into an XML element can return many non-equivalent substitutions, therefore Xcerpt$_1^s$ is a rule language with rich unification.

Subsumption and Simulation in Xcerpt$_1^s$

sg \ ***qt***	b	i	$\leq i$	Variable	Sum or Diff	$\leq w$	$q[y_1, ..., y_m]$
a	$if\ a \neq b \Rightarrow -, else +$	$-$	$-$	$-$	$-$	$-$	$-$
j	$-$	$if\ j \neq i \Rightarrow -, else +$	$-$	$-$	$-$	$-$	$-$
$\leq j$	$-$	$if\ j < i \Rightarrow -, else +$		$-$	$-$	$-$	$-$
Variable	$+$	$+$	$+$	$+$	$+$	$+$	$+$
Sum or Diff	$-$	$+$	$+$	$+$	$+$	$+$	$-$
$\leq v$	$-$	$+$	$+$	$+$	$+$	$+$	$-$
$p[x_1, ..., x_n]$	$-$	$-$	$-$	$-$	$-$	$-$	$if\ p \neq q \Rightarrow -, else$ $if\ (*) \Rightarrow +, else -$

where

sg	is a solution goal
qt	is a query term
a, b	are MyStrings
j, i	are MyFixnums
v, w	are Variables, Sums or Diffs
$p[x_1, ..., x_n]$, $q[y_1, ..., y_m]$	are Terms, $n, m \in \mathbb{N}$
$(*)$	*there is a monotone bijection* $\pi : \{x_1, ..., x_n\} \rightarrow \{y_1, ..., y_m\}$ *such that* $\forall l \in \mathbb{N}, l \leq n.\ x_l \supseteq \pi(x_l)$

Table 7.1: Subsumption in Xcerpt$_1^s$

Xcerpt$_1^s$ subsumption between a solution goal sg and a query term qt is illustrated

by Table 7.1. For instance, if sg is a string a and qt is a string b, then $sg \supseteq qt$ if a and b are equal. If a solution goal is a variable it subsumes any query term. Therefore if there is a solution goal which is a variable X, it is always the last solution goal in the table. This fact can negatively impact the time complexity of the resolution of look-up goals of X because they must pick out the suitable solutions out of the answer set of X which can be very extensive. In the worst case the answer set for X involves all positive atoms of the well-founded model of the respective program. But if SBR did not table variables, the algorithm would not terminate for many programs.

If sg and qt are terms with equal labels, the subsumption of each child of qt at position i by the child of sg at position i is tested. If at least one of these checks fails, it holds that $sg \not\supseteq qt$.

Xcerpt$_1^s$ simulation of a query term qt into a construct term ct is illustrated by Table 7.2. Substitutions returned by some simulations must be resolved due to the peculiarities of the module described in Subsection 7.1.3. This is denoted by the function *eval* in the table.

If a construct term is of the form $w > k$ where w is a number, a variable, sum or difference and k is a number, k is only used if solution goal is a number j or an expression of the form $\leq j$, where j is a number , to compare it with j. Because of this comparision SBR terminates for instance of the query *Acquaintance[chuck, 1]* w.r.t. the program P in Listing 4.1. Without $>$-symbol in the head of the second rule of P, SBR would try to prove the goals *Acquaintance[chuck,* ≤ 0 *]*, *Acquaintance[chuck,* ≤ -1 *]*, *Acquaintance[chuck,* ≤ -2 *]* and so on. This is a good example demonstrating how rich unification prevents non termination. And this is the only task of $> k$ in a construct term. $> k$ does not play any role in substitutions.

7.1.4 Xcerpt$_2^s$ Module

The complete UML-diagram of the module is shown in Figure 7.6. The Xcerpt^s program G in Listing 4.2 is an example of an Xcerpt$_2^s$ program.

Xcerpt$_2^s$ is much less expressive than Xcerpt. In addition to positive or negative, ground or non-ground complete literals of Prolog, Xcerpt$_2^s$ allows incomplete query terms and optional variables as direct children of incomplete terms. Xcerpt$_2^s$ has no features which are not possible in Xcerpt. Therefore Xcerpt$_2^s$ is a real subset of Xcerpt. See Figure 7.7.

qt \ **ct**

	b	i	$i > k$	Variable	Sum or Diff	$w > k$	$q[y_1, ..., y_m]$
a	$if\ a \neq b \Rightarrow -, else +$	$-$	$-$	$qt \leftarrow ct$	$-$	$-$	$-$
j	$-$	$if\ j \neq i \Rightarrow -, else +$	$if\ j \neq i\ or\ j \leq k \Rightarrow -, else +$	$qt \leftarrow ct$	$eval(qt \leftarrow ct)$	$if\ j \leq k \Rightarrow -, else\ eval(qt \leftarrow w)$	$-$
$\leq j$	$-$	$if\ j < i \Rightarrow -, else\ ct \leftarrow qt$	$if\ j < i\ or\ j \leq k \Rightarrow -, else\ i \leftarrow qt$	$qt \leftarrow ct$	$eval(qt \leftarrow ct)$	$if\ j \leq k \Rightarrow -, else\ eval(qt \leftarrow w)$	$-$
Variable	$ct \leftarrow qt$	$ct \leftarrow qt$	$i \leftarrow qt$	$ct \leftarrow qt$	$ct \leftarrow qt$	$w \leftarrow qt$	$ct \leftarrow qt$
Sum or Diff	$-$	$eval(ct \leftarrow qt)$	$eval(i \leftarrow qt)$	$eval(ct \leftarrow qt)$	$eval(ct \leftarrow qt)$	$eval(w \leftarrow qt)$	$-$
$\leq v$	$-$	$eval(ct \leftarrow qt)$	$eval(i \leftarrow qt)$	$eval(ct \leftarrow qt)$	$eval(ct \leftarrow qt)$	$eval(w \leftarrow qt)$	$-$
$p[x_1, ..., x_n]$	$-$	$-$	$-$	$qt \leftarrow ct$	$-$	$-$	$if\ p \neq q \Rightarrow -, else\ if\ (*) \Rightarrow (**), else -$

where

ct	is a construct term
qt	is a query term
a, b	are MyStrings
j, i, k	are MyFixnums
v, w	are Variables, Sums or Diffs
$p[x_1, ..., x_n]$, $q[y_1, ..., y_m]$	are Terms, $n, m \in \mathbb{N}$
$(*)$	*there is a monotone bijection* $\pi : \{x_1, ..., x_n\} \rightarrow \{y_1, ..., y_m\}$ *such that* $\forall l \in \mathbb{N}, l \leq n.\ x_l \preceq \pi(x_l)$
$(**)$	*substitution set yield by the simulation of qt into ct*

Table 7.2: Simulation in Xcerpt$_1^s$

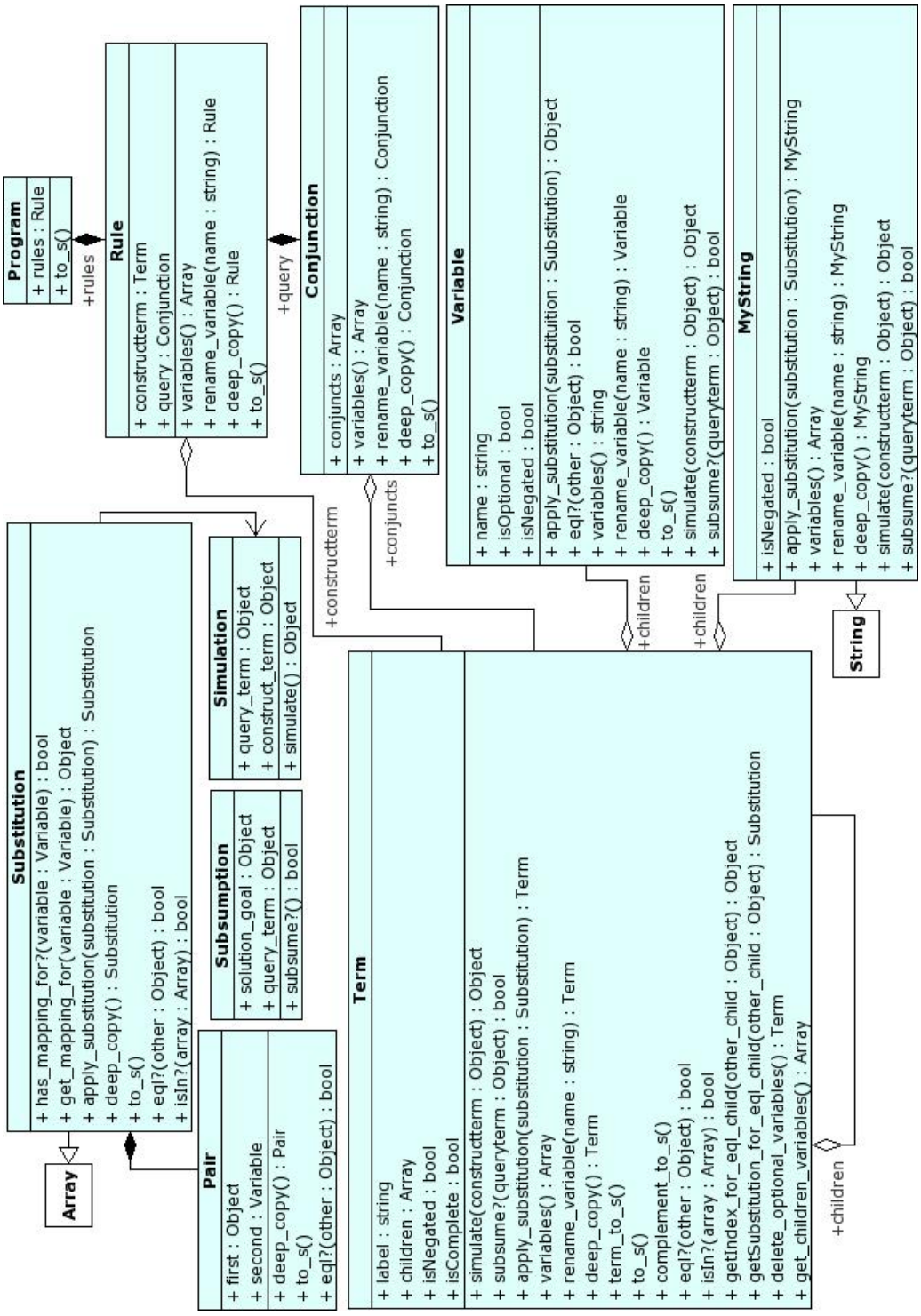

Figure 7.6: UML-diagram of Xcerpt$_2^s$ module

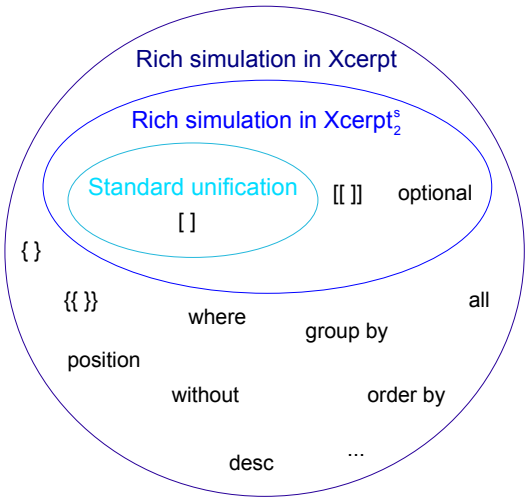

Figure 7.7: Standard unification vs. rich simulation in Xcerpt$_2^s$ vs. rich simulation in Xcerpt

Xcerpt$_2^s$ Query

Definition 82. (Xcerpt$_2^s$ Query).
An *Xcerpt$_2^s$ query* is defined as follows:

1. Every *string* is a query.
2. Every *variable* is a query.
3. Every *optional varaible* of the form *optional* X where X is a variable, is a query.
4. Every *positive complete term* of the form $f[x_1, ..., x_n]$ where f is an arbitrary function symbol, $x_1, ..., x_n$ are strings, variables or positive terms and $n \in \mathbb{N}$, is a query.
5. Every *positive incomplete term* of the form $f[[x_1, ..., x_n]]$ where f is an arbitrary function symbol, $x_1, ..., x_n$ are strings, (optional) variables or positive terms and $n \in \mathbb{N}$, is a query.
6. Every *negative complete term* of the form $\neg f[x_1, ..., x_n]$ where f is an arbitrary function symbol, $x_1, ..., x_n$ are strings, variables or positive terms and $n \in \mathbb{N}$, is a query.
7. Every *negative incomplete term* of the form $\neg f[[x_1, ..., x_n]]$ where f is an arbitrary function symbol, $x_1, ..., x_n$ are strings, (optional) variables or positive terms and $n \in \mathbb{N}$, is a query.
8. Every *conjunction* of queries is a query.

Examples of $Xcerpt_2^s$ queries:

tra-la-la

optional var X

knows[[ ]]

AuxiliaryRule[[n[n[ ]]]]

Acquaintance[[chuck, n[[n[[ ]]]]]]

Acquaintance[var P, var D], knows[var P, chuck], not(knows[chuck, var P])

The following queries are not valid for $Xcerpt_1^s$:

knows{{ without anna }}

Acquaintance[optional var P, not(>4-1-4)]

Since an Xcerpt$_2^s$ query term can be incomplete, a substitution set returned by the simulation of a query term into a construct term can contain many substitutions. See an example in Section 4.5. Xcerpt$_2^s$ is a rule language with rich unification.

Subsumption and Simulation in Xcerpt$_2^s$

sg \ ***qt***	b	Variable	$q[y_1, ..., y_m]$	$q[[y_1, ..., y_m]]$
a	$if\ a \neq b \Rightarrow -, else +$	$-$	$-$	$-$
Variable	$+$	$+$	$+$	$+$
$p[x_1, ..., x_n]$	$-$	$-$	$if\ p \neq q \Rightarrow -, else\ if\ (*) \Rightarrow +, else -$	$-$
$p[[x_1, ..., x_n]]$	$-$	$-$	$if\ p \neq q \Rightarrow -, else\ if\ (**) \Rightarrow +, else -$	$if\ p \neq q \Rightarrow -, else\ if\ (**) \Rightarrow +, else -$

where

sg	is a solution goal
qt	is a query term
a, b	are MyStrings
$p[x_1, ..., x_n],\ q[y_1, ..., y_m]$	are complete terms, $n, m \in \mathbb{N}$
$p[[x_1, ..., x_n]],\ q[[y_1, ..., y_m]]$	are incomplete terms, $n, m \in \mathbb{N}$
$(*)$	*there is a monotone bijection* $\pi : \{x_1, ..., x_n\} \rightarrow \{y_1, ..., y_m\}$ *such that* $\forall l \in \mathbb{N}, l \leq n.\ x_l \supseteq \pi(x_l)$
$(**)$	*there is a total, monotone and injective mapping* $\pi : \{x_1, ..., x_n\} \rightarrow \{y_1, ..., y_m\}$ *such that* $\forall l \in \mathbb{N}, l \leq n.\ x_l \supseteq \pi(x_l)$

Table 7.3: Subsumption in Xcerpt$_2^s$

Xcerpt$_2^s$ subsumption and simulation are illustrated by Tables 7.3 and 7.4 respectively. In addition to subsumption and simulation of strings, variables and complete terms which are defined like in Xcerpt$_1^s$, Xcerpt$_2^s$ considers subsumption and simulation involving incomplete query terms. Optional variables are not considered extra because of the following

qt \ ct	b	Variable	$q[y_1, ..., y_m]$
a	$if\ a \neq b \Rightarrow -, else +$	$-$	$-$
Variable	$qt \leftarrow sg$	$qt \leftarrow sg$	$qt \leftarrow sg$
$p[x_1, ..., x_n]$	$-$	$-$	$if\ p \neq q \Rightarrow -, else\ if\ (*) \Rightarrow (***), else -$
$p[[x_1, ..., x_n]]$	$-$	$-$	$if\ p \neq q \Rightarrow -, else\ if\ (**) \Rightarrow (***), else -$

where

ct	is a construct term
qt	is a query term
a, b	are MyStrings
$p[x_1, ..., x_n],\ q[y_1, ..., y_m]$	are complete terms, $n, m \in \mathbb{N}$
$p[[x_1, ..., x_n]]$	is an incomplete term, $n \in \mathbb{N}$
$(*)$	*there is a monotone bijection* $\pi : \{x_1, ..., x_n\} \rightarrow \{y_1, ..., y_m\}$ *such that* $\forall l \in \mathbb{N}, l \leq n.\ x_l \preceq \pi(x_l)$
$(**)$	*there is a total, monotone and injective mapping* $\pi : \{x_1, ..., x_n\} \rightarrow \{y_1, ..., y_m\}$ *such that* $\forall l \in \mathbb{N}, l \leq n.\ x_l \preceq \pi(x_l)$
$(***)$	*substitution set yield by the simulation of qt into ct*

Table 7.4: Simulation in Xcerpt$_2^s$

reasons:

1. If a query or construct term t is an optional variable it is treated analogously to a non-optional variable with the only difference that an optional variable is *true* if there are no solutions for it because an optional variable can be left unbounded.

2. If an optional variable X appears within a term t, X is important neither by the subsumption checks nor by the computation of simulation of t because X must not be bound. Therefore X can be omitted without changes of the semantic of t. For example the query term *Acquaintance[anna, n[[optional var D]]]* is equivalent to the query term *Acquaintance[anna, n[[ ]]]*, i.e. the answer sets for the both terms are equal.

7.2 The Core of SBR

The core of SBR involves the following methods (compare Figure 7.2):

1. *resolve()*

2. *new_ child(goal : Goal, child_ of_ goal : Child_ of_ goal)*

3. *end_ of_ branch()*

4. *select_suspended_node_of_subtree()*

Since their pseudo-code is very complex, the methods are explained with the help of flow diagrams. Each flow diagram is an abstraction of the real code. Before we start explaining the methods, let us take a look at the notation used in the flow charts:

1. Rectangles represent actions. A rectangle has one or many incoming arrows and zero or one outcoming arrows. The arrows are not labeled. Each flow diagram has a start rectangle. It is marked with an incoming arrow. The label of a start rectangle is coloured. Thanks to the start node each flow chart can be called within another flow diagram. A coloured rectangle A represents a call of the flow diagram with the start rectangle A.

2. Rhombs represent conditions. A rhomb has one or many incoming arrows and two outcoming arrows. One outcoming arrow is labeled with *yes*, the other one with *no*.

3. Arrows determinate the order of actions and conditions. Please notice that all flow charts used in this section are deterministic.

Definition 71 specifies the evaluation order of suspended nodes. Subsection 4.1.5 and Sections 4.3 and 4.4 illustrate the method *select_suspended_node_of_subtree()* by numerous examples. This section is devoted to the explanation of the other three core methods of SBR.

7.2.1 Resolution of a Node

The method *resolve()* is the most difficult and the most important method of the class *Node*. Flow diagrams 7.8, 7.9 and 7.10 describe the method.

Consider Flow chart 7.8. At first SBR determinats whether the current node begins with a solution or a look-up goal.

If the first goal of the current node is negative, let us call it $\neg g$, so that its complement g is subsumed by no previous solution goal, $\neg g$ is a look-up goal. An auxiliary resolution tree with g as its root, is created. The method *resolve()* is called recursively with the new root.

If the first goal is a solution goal it is saved in the table and the children of the goal are initialized (recall that a child of a solution goal is a rule such that the solution goal unifies with the head of the rule and a substitution yield by this unification). If there are some children of the goal, a resolution step using the first child is made. As a result a new resolvent of the current node is created by calling the other flow diagram named *Create*

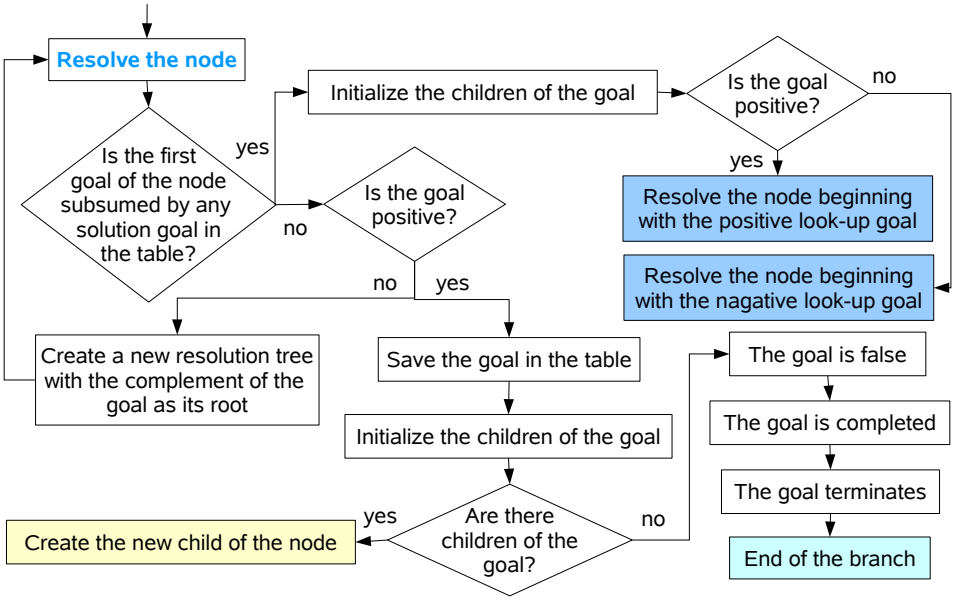

Figure 7.8: Resolution of a node

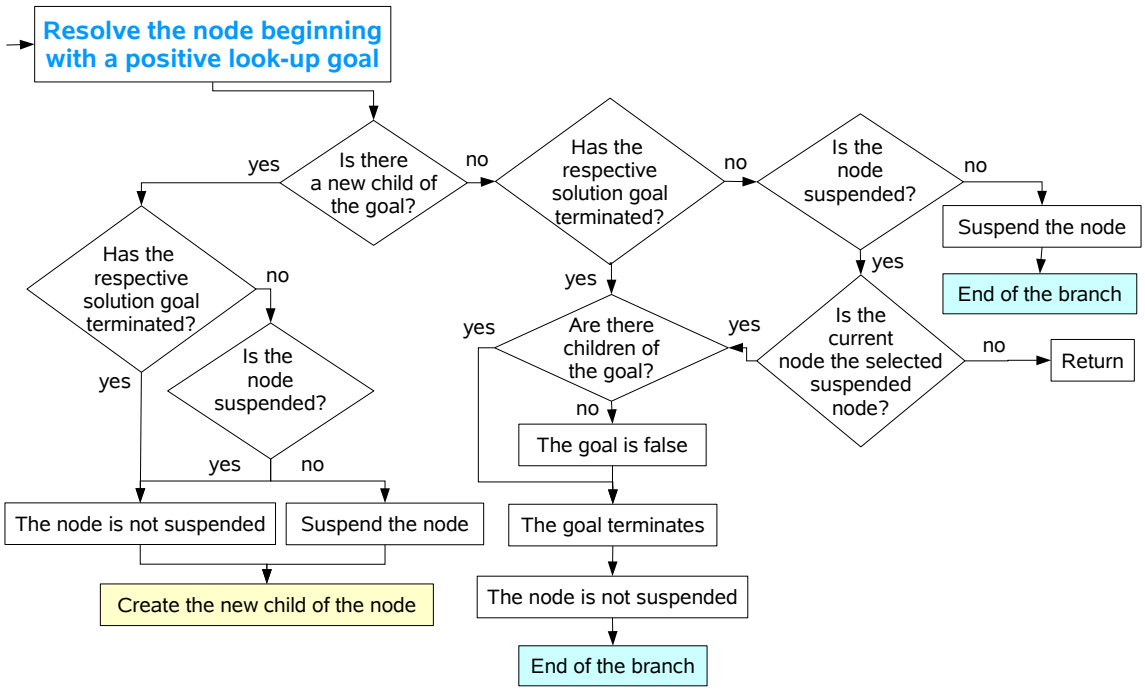

Figure 7.9: Resolution of a node beginning with a positive look-up goal

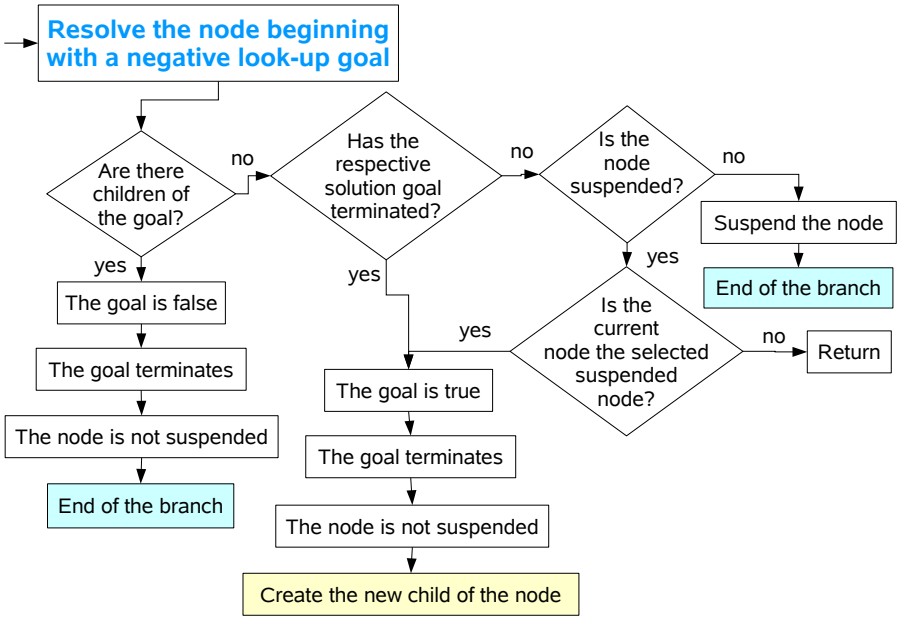

Figure 7.10: Resolution of a node beginning with a negative look-up goal

the new child of the node. If no resolution step is possible, the solution goal is interpreted to be *false* and terminates. The flow diagram *End of the branch* processing backtracking, is called.

If the first goal is a look-up goal the children of the goal are initialized (a child of a look-up goal is a solution for its respective solution goal and a substitution yield by the unification of the look-up goal with the solution). Depending on the polarity of the goal Flow diagram 7.9 or 7.10 is called.

If a solution goal S has not terminated yet and it is still possible to find new solutions for it, the nodes beginning with the respective look-up goals of S must be suspended. A node can be suspended only once because a suspended node is resolved only if it is a selected suspended node (see Definition 71) or if S is terminated. In both cases the first goal of the node is interpreted to be *true* or *false* and terminates. Therefore the node cannot be suspended any more (compare Definition 64). Hence, there are at most two phases in the evaluation of a look-up goal, namely before its node is suspended and afterwards.

Flow diagram 7.9 describes the resolution of a node N beginning with a positive look-up goal L. Let S be the respective solution goal of L. If L has a new child a resolution step is made for N by calling Flow chart 7.11. If L has no new child and S has terminated, L is interpreted to be *false* if it has no children, it is *true* otherwise. L terminates and Flow chart 7.12 is called. During the first phase of the evaluation of L, N is suspended if S has not terminated yet. During the second phase of the evaluation of L (i.e. N is the selected suspended node), L is interpreted to be *false* if it has no children. It is *true* otherwise. L

terminates and N is not suspended any more. The method *end_ of_ branch()* is called for N.

During traversing a resolution tree upwards SBR passes all non-selected suspended nodes without considering them if no resolution steps are possible for them. That is why the method *resolve()* has no effect on a non-selected suspended node N beginning with a positive look-up goal L if L has no new child and S is non-terminated. Otherwise all resolution steps possible for N and changes of the state of L and N are processed.

The resolution of a node beginning with a negative look-up goal is a converse of the evaluation of a node beginning with a positive look-up goal. Compare Figures 7.9 and 7.10.

7.2.2 New Node of a Resolution Tree

Flow chart 7.11 describes the creation of a resolvent of a node N. Let G be the first goal of N. The method uses a child of G to make a resolution step for a node N. A child of G is a rule and a substitution yield by the unification of G with the rule head (if G is a solution goal) or a solution for the respective solution goal of G and a substitution yield by the unification of G with the solution (if G is a look-up goal). Recall that there can be many substitutions for G and a rule head or G and a solution. That is why a substitution is a part of the child of a goal.

Let N' be a resolvent of N. If G is a solution goal N' is a copy of N but with a list of goals G depends on instead of G. If G is a look-up goal N' is a copy of N without G. In both cases the substitution is applied to all goals of N'. Compare Definition 59.

If N' contains an empty list a solution for the respective solution goal of the list is computed and saved in the table (Definition 62). If N' becomes empty and it appears within the main resolution tree a solution for the initial query is computed (Definition 74). The method *end_ of_ branch()* is called for the empty node N'. If N' is not empty Flow chart 7.8 processing the resolution of N', is called.

7.2.3 Backtracking

If no resolution step is possible for a node N the method *end_ of_ branch()* is called for it. The method processes backtracking. See Figure 7.12.

Let G be the first goal of N. If N is empty or G is terminated there is nothing to do for N any more. If N is not a root of a resolution tree, SBR goes to the father node of N. If a resolution step is possible for the father node, SBR makes it by calling Flow chart 7.11. Otherwise the method *end_ of_ branch()* is called recursively for the father node of

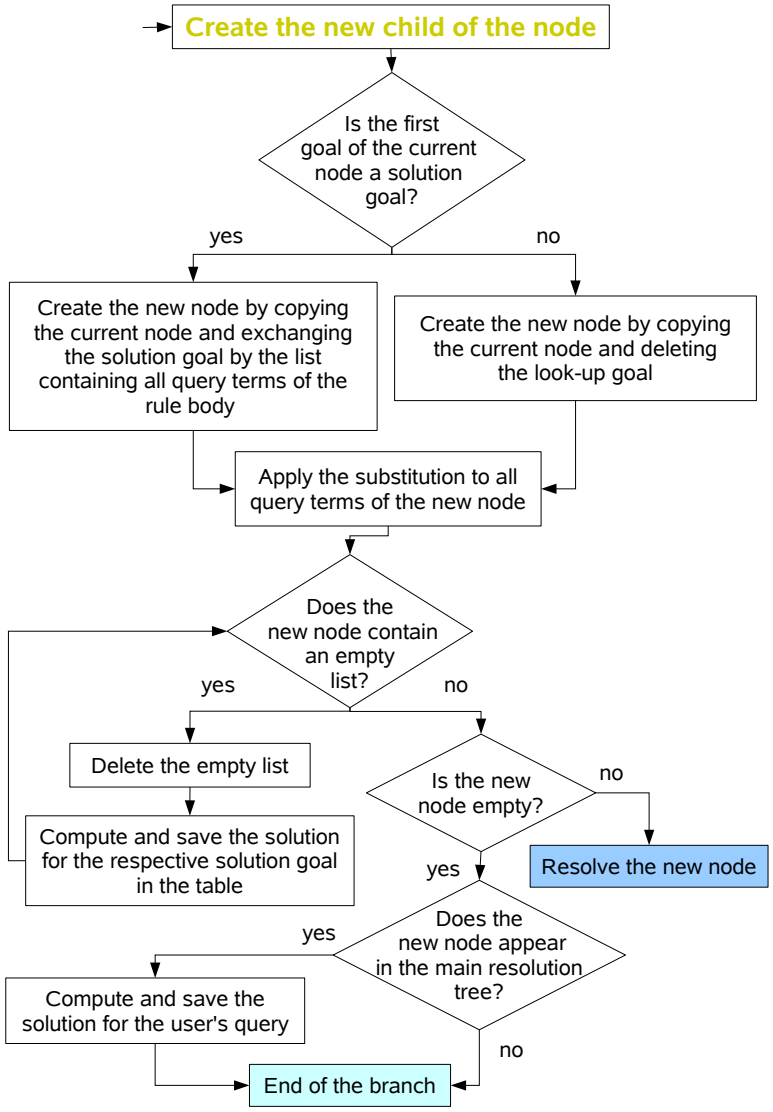

Figure 7.11: New node of a resolution tree

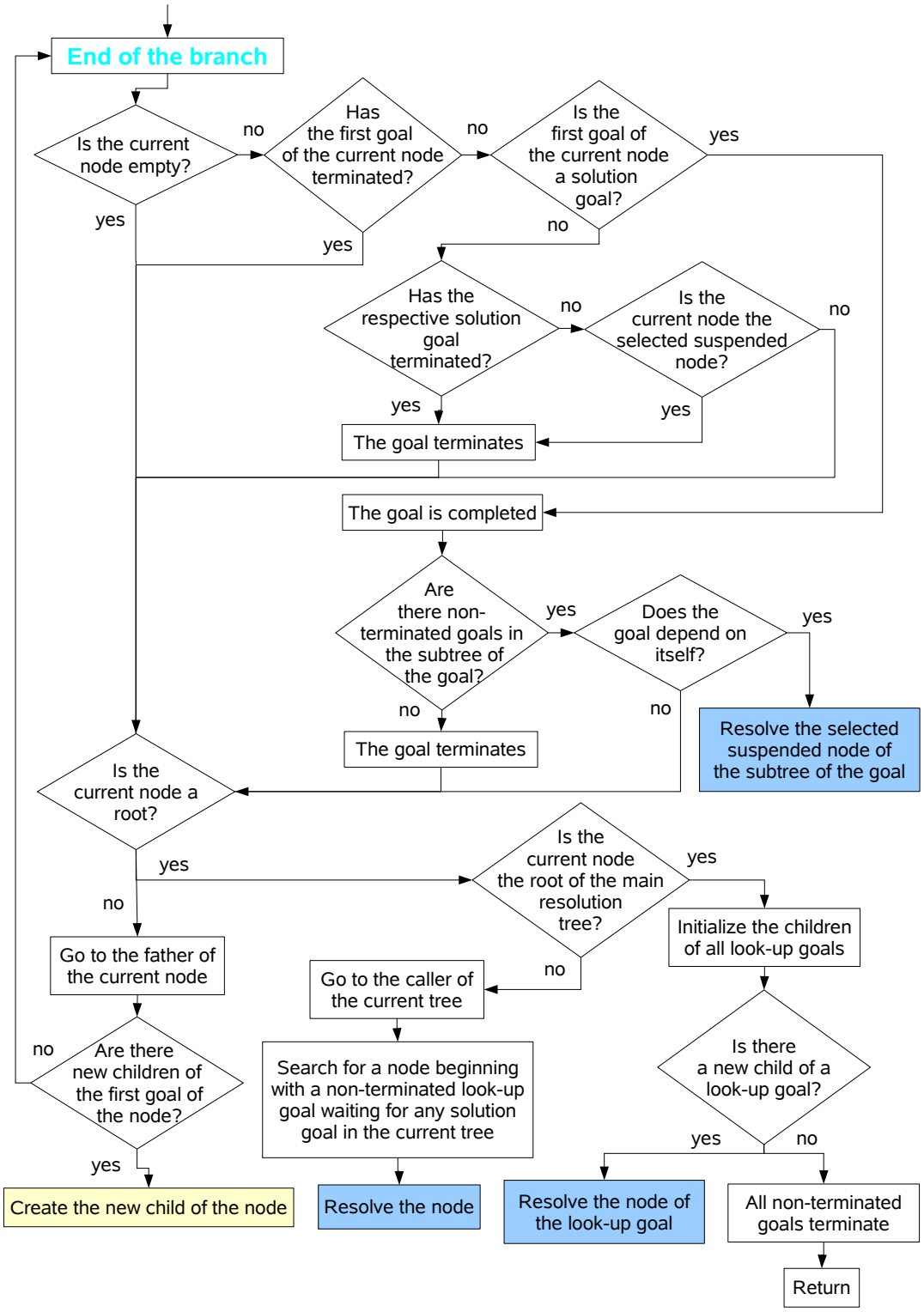

Figure 7.12: Treatment of a brunch end of a resolution tree

N.

If G is non-terminated, the method tries to terminate it. If G is a look-up goal it terminates if the respective solution goal has terminated or if N is the selected suspended node. If G is a solution goal it terminates if there are no non-terminated goals in its subtree. Otherwise SBR tests whether the solution goal G depends on itself. In this case the method *resolve()* is called for the selected suspended node of the subtree of G. (The selected suspended node is determinated by Flow chart 4.7.)

If N is the root of the main resolution tree, SBR tests whether some new resolution steps are possible for a node beginning with a look-up goal. If the algorithm finds such a node, the node is completely evaluated. Otherwise all non-terminated goals terminate.

If N is the root of an auxiliary resoltution tree T, SBR goes to the caller tree of T, searches for a non-terminated look-up goal waiting for any solution goal of T and resolves it.

Chapter 8

Conclusion

In this chapter we summarise the extension of the SPARQL semantics and the development of the subsumption-based resolution strategy for rule languages with rich unification. In Section 8.1 we enumerate the benefits of SPARQL programs as well as the features of SBR and say which of these features are adopted from locally stratified SLG-resolution and which are our contribution. In Section 8.2 some improvements of SBR are suggested.

8.1 Results of this Work

One of the main contributions of this work is the extension of the semantics of SPARQL from single SPARQL rules to possibly recursive SPARQL programs with negation as failure. We defined the declarative and the procedural semantics of SPARQL, in particular the following points have been considered:

1. SPARQL programs and their benefits compared to single SPARQL rules evaluated independetly from each other, they are:
   (a) Turing-completeness of SPARQL,
   (b) convenient possibility of reuse, viewing and dynamic networking of RDF graphs constructed by other rules,
   (c) low error rate because of the short and clear patterns in a rule body,
   (d) more efficient evaluation because of tabling.
2. The well-founded semantics of SPARQL programs
3. The local stratification of SPARQL programs

4. The procedural semantics for SPARQL defined by SBR

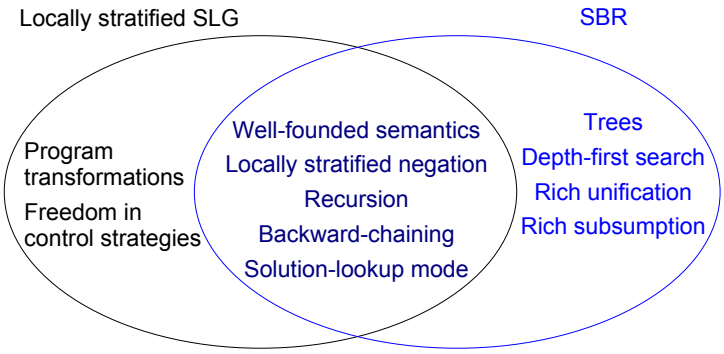

Figure 8.1: Locally stratified SLG-resolution vs. SBR

Summarising the development of the SBR algorithm we would like to emphasize that SBR maintains the following features of locally stratified SLG-resolution (compare Figure 8.1):

1. SBR is *sound* w.r.t. the well-founded semantics.
2. SBR resolves queries w.r.t. *locally stratified recursive* programs.
3. Because of backward-chaining the algorithm resolves *no irrelevant queries*.
4. SBR adopts the solution-lookup mode. Consequently, it performs *no redundant computations* but is unfortunately *not linear*.
5. Thanks to the idea of tabling the algorithm *terminates* for programs with the bounded-term-size property.
6. Because of tabling and backtracking SBR is *fair* and *partially complete*. It returns all answers for the user's query or 'false' if there are no solutions for it.

There are also some differences between locally stratified SLG-resolution and SBR, they are:

1. For the sake of simplicity SBR does not allow maximum freedom in control strategies in contrast to SLG-resolution. Our algorithm processes *depth-first search*. But this limitation does not decrease the expressive power of SBR compared to locally stratified SLG-resolution. Both approaches are Turing-complete.
2. SLG-resolution is based on program transformations, whereas SBR uses the standard *tree-based formulation* which are easier to understand and to keep track of the query evaluation for the reader.

The following advantages of SBR are our contribution:

1. Since rich unification and subsumption are strong means to address variable schemas of different rule languages, SBR is *generic*.
2. Since our evaluation strategy bases on rich subsumption between goals, it performs no redundant computations. SBR is therefore in average case *more efficient* than locally stratified SLG-resolution.

8.2 Future Work

Locally stratified SLG-resolution could be improved in the following ways:

1. The algorithm could be extended to a broader class of programs than locally stratified programs. In this case it must detect and treat cycles through negation in a resolution tree.
2. The evaluation strategy should perform efficient tabling.
   SBR saves all solution goals in the table. The space complexity of the algorithm can be improved by tabling only recursive solution goals. A solution goal S is *recursive* if it depends on a goal S' such that $S' \subseteq S$. For instance because of the second rule of the Xcerpt^s program P in Listing 4.1 the solution goals with the label *Acquaintance* are recursive and the solution goals with the label *knows* are not recursive. If SBR tabled only recursive solution goals the table would contain less information. But repeated computations of non-recursive goals would be possible in this case.
   The other way to better profit from the idea of tabling is to reuse the intermediate results of previous resolutions instead of computing them anew for each query. In this case one must decide when the table is to be updated due to the changeability of Web data.

Other improvements of SBR:

1. For the sake of brevity the optimization of SBR is not considered in this work. But there are many ways to make the algorithm more efficient. They are, for instance:
   - SBR saves the complete resolution forest for each query. Its space complexity can be improved if the algorithm deletes the branches of a resolution tree containing no suspended nodes.
   - SBR should use more efficient data structures than lists.

2. We have tested SBR with a variant of Xcerpt. The algorithm should be tested with other rule languages with rich unification as well.

List of Figures

List of Tables

Bibliography

[1] Apt, K. R., van Emden, M.: Contributions to the Theory of Logic Programming, ACM, vol. 29, Issue 3, 1982, pages 841-862

[2] Bry, F., Eisinger, N., Eiter, T., Furche, T., Gottlob, G., Ley, C., Linse, B., Pichler, R., Wei, F.: Foundations of Rule-Based Query Answering. In: vol. 4636 of Lecture Notes In Computer Science, Springer, 2007, pages 1-153

[3] Bry, F., Eisinger, N.: Übersetzerbau - Abstrakte Maschinen. Lecture notes, Institute of Computer Science, LMU, Munich, 2004

[4] Bry, F., Estekhina, O., Furche, T., Linse, B.: Well-Founded Semantics and Local Stratification for Xcerpt Programs. Project thesis, University of Munich, 2008

[5] Bry, F., Furche, T., Ley, C., Linse, B., Marnette, B.: RDFLog: It's like Datalog for RDF. In: Proc. 22^{nd} Workshop on (Constraint) Logic Programming, 2008

[6] Bry, F., Furche, T., Linse, B.: Simulation Subsumption or Déjà vu on the Web. In: Proc. 2^{nd} Int'l. Conf. on Web Reasoning and Rule Systems, vol. 5341 of Lecture Notes In Computer Science, Springer, 2008, pages 28-42

[7] Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Progams. ACM vol. 43, Issue 1, 1996, pages 20-74

[8] Chen, W., Warren, D.S.: Towards Effective Evaluation of General Logic Programs. In: Proc. 12^{th} PODS, 1993

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

[10] Hayes, P., McBride, B.: RDF Semantics. Recommendation, W3C 2004

[11] Klyne, G., Carroll, J.J., McBride, B.: Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C 2004

[12] Kowalski, R., Kuehner, D.: Linear Resolution with Selection Function. Artificial Intelligence, vol. 2, 1971, pages 227-260

[13] Loveland, D. W.: A Linear Format for Resolution. Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer, Berlin and New York, 1970, pages 147-163

[14] Luckham, D.: Refinement Theorems in Resolution Theory. Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer, Berlin and New York, 1970, pages 163-191

[15] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: 5^{th} Int'l. Semantic Web Conf., vol. 4273 of Lecture Notes in Computer Science, Springer, 2006, pages 30-43

[16] Polleres, A., Feier, C., Harth, A.: Rules with Contextually Scoped Negation. In: Proc. 3^{rd} European Semantic Web Conf., Springer, 2006

[17] Polleres, A.: From SPARQL to Rules (and Back). In: Proc. Int'l. World Wide Web Conf. (WWW), New York, NY, USA, ACM, 2007, pages 787-796

[18] Polleres, A.: SPARQL Rules!. GIA Technical report, University Rey Juan Carlos, Madrid, 2006

[19] Prud'hommeaux, E., Seaborne, A., Laboratories, H.-P.: SPARQL Query Language for RDF. Recommendation, W3C 2008

[20] Przymusinski, T. C.: On the Declarative and Procedual Semantics of Logic Programs. Journal of Automated Reasoning, vol. 5, Issue 2, 1987, pages 167-205

[21] Rao, P., Sagonas, K., Swift, T., Warren, D. S., Freire, J.: XSB: A System for Efficiently Computing Well-Founded Semantics. In: Proc. 4^{th} Int'l Conf. on Logic Programming and NonMonotonic Reasoning, Springer, 1997, pages 430-440

[22] Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. ACM, vol. 12, Issue 1, 1965, pages 23-41

[23] Ross, K. A.: A Procedural Semantics for Well Founded Negation in Logic Programs. Journal of Logic Programming, 1989, pages 22-33

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

[25] Schenk, S., Staab, S.: Networked Graphs: a Declarative Mechanism for SPARQL Rules, SPARQL Views and RDF Data Intergration on the Web. In: Proc. Int'l. World Wide Web Conf. (WWW), New York, NY, USA, ACM, 2008, pages 585-594

[26] Shen, Y.-D., Yuan, L.-Y., You, J.-H.: SLT-Reolution for the Well-Founded Semantics. In: Journal of Automated Reasoning, vol. 28, Issue 1, 2002, pages 53-97

[27] Swift, T., Warren, D. S.: Efficiently Implementing SLG Resolution, 1994, ftp://ftp.cs.sunysb.edu/pub/TechReports/swift/auxiliary/outdated/xsb-impl_old.ps.Z

[28] Tamaki, H., Sato, T.: OLD-Resolution with Tabulation. In: Proc. 3^d Int'l Conf. on Logic Programming, Springer, New York, 1986, pages 84-98

[29] Zamov, N.K., Sharonov, V.I.: On a class of strategies which can be used to establish decidability by the resolution princilpe. (In Russian) Issled, po konsruktivnoye matematikye i matematicheskoie logikye III, 16, 1969, pages 54-64