INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Projektarbeit

Well-Founded Semantics and Local Stratification for Xcerpt Programs

Olga Estekhina

Aufgabensteller:

Prof. Dr. François Bry

Betreuer:

Tim Furche

Benedikt Linse

August 25, 2008

Declaration

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Signature:

Date:

Abstract

One of the most important requirements to Xcerpt is the ability to answer each Xcerpt query which is relevant for an Xcerpt program with *yes, true* or *no, false*. And there should be no query which is answered with *I do not know, it is not defined*. This requirement is satisfied if the program has a total well-founded model. Total well-founded model is a semantic feature of a program which is difficult to analyse. That is why we search for a syntactic feature which guarantees a total well-founded model.

All stratifiable logic programs have a total well-founded model. But there are many unstartifiable programs which also have a total well-founded model. That is why we define a broader syntactic condition for Xcerpt programs, namely local stratification which also guarantees the total well-founded model. In this work we prove that every locally stratifiable Xcerpt program has a total well-founded model.

Contents

Chapter 1

Motivation

One of the most important requirements to query languages is the ability to answer user's queries. In order to realize this task a query language must have tools which make it possible to analyse the given data base and to derive new information. (In classical logic the fixpoint operator helps to carry out this task.)

Note that each Xcerpt query which is relevant for an Xcerpt program, must be answered with *yes, true* or *no, false*. And there is no query which is answered with *I do not know, it is not defined*. This requirement is satisfied if the program has a total well-founded model. That means, each ground atom is either ´true´ or ´false´ in the well-founded model of the program. There is no ground atom which is undefined.

All stratifiable logic programs have a total well-founded model. But there are many unstartifiable programs which also have a total well-founded model. That is why we define a broader condition for Xcerpt programs which guarantees the total well-founded model.

Besides, the computation must be efficient. No information is evaluated which is irrelevant for the query (backward-oriented computation). And each information is computed only once.

The aims of this work are:

1. Definition of the well-founded semantics for Xcerpt programs
2. Extension of the formal semantics of Xcerpt from stratifiable programs to locally stratifiable programs
3. An efficient and sound evaluation strategy for locally stratifiable Xcerpt programs

At the beginning let us consider some logic notions which are relevant for this work. Many other logic notions are used in this work as well. See [1].

1.1 Immediate Consequence Operator and Stratification for Sets of Normal Clauses

Consider the set of normal clauses Z in Listing 1.1. The head of a normal clause is an atom. The body of a normal clause is a conjunction of literals.

Listing 1.1: Set of normal clauses Z

```
p :- q
q :- not r
r :- s
s
```

Before we consider the fixpoint semantics of the program, let us remember how the notion 'operator' is defined.

Definition 1 (Operator). Let S be a set. Let $\mathcal{P}(\mathcal{S})$ denote the powerset of S, i.e. the set of subsets of S. An operator on S is a mapping Γ: $\mathcal{P}(\mathcal{S}) \rightarrow \mathcal{P}(\mathcal{S})$. [1]

Definition 2 (Monotonic operator). An operator Γ on a set S is monotonic, iff for all subsets $M \subseteq M' \subseteq S$ holds: $\Gamma(M) \subseteq \Gamma(M')$. [1]

For the definition of the formal semantics of a logic program, a monotonous operator is advantageous, because no computation of a monotonic operator is in vain.

Definition 3 (Immediate consequence operator for sets of normal clauses). Let S be a set. Let $\mathcal{HB}$ be its associated Herbrand base and $\mathcal{B} \subseteq \mathcal{HB}$ a set of ground atoms. The immediate consequence operator T_S for S is:
T_S: $\mathcal{P}(\mathcal{HB}) \rightarrow \mathcal{P}(\mathcal{HB})$
$\mathcal{B} \rightarrow \{$ A$\in \mathcal{HB} \mid$ there is a ground instance A$\leftarrow \varphi$ of a member of S with $\mathcal{HI}(\mathcal{B})$[1]$\models \varphi$ $\}$

Definition 4 (Fixpoint). Let Γ be an operator on a set S. A subset $M \subseteq S$ is a fixpoint of Γ iff $\Gamma(M)=M$. [1]

Let us compute $\mathbf{T}_Z$:
$\mathbf{T}_Z \uparrow 0 = \oslash$
$\mathbf{T}_Z \uparrow 1 = \{s,q\}$
`s` is true, because it is a fact. We don't know whether `r` is true or false, we asume that it is false and its negation is true (negation as failure). That is why `q` is also true.

[1] Herbrand interpretation of S with respect to B, see [1] for the definition

$\mathbf{T}_Z \uparrow 2$ = {s,q,r,p}
But later the atom `r` turns out to be true and its negation is false. And we must exclude all the atoms depending on the negation of `r`, namely `q`. And some other atoms depending on `q`, namely `p`.
$\mathbf{T}_Z \uparrow 3$ = {s,r,p}
$\mathbf{T}_Z \uparrow 4$ = {s,r} = $\mathbf{T}_Z \uparrow 5 \Rightarrow$ A fixpoint is reached.

Consider M={s, q:-¬r}⊆M'={s, q:-¬r, r:-s}⊆Z, but $\mathbf{T}_Z$(M)={s,q}$\nsubseteq$ $\mathbf{T}_Z$(M')={s,r}
Therefore $\mathbf{T}_Z$ is for a set of normal clauses not monotonic. That is the reason why we need stratification. One tries to order the clauses and to compute $\mathbf{T}_Z$ in this order. The order requires that all clauses a negative literal depends on, are evaluated before the clauses with the negative literal in their bodies are computed. Such an order for the program Z can be defined as follows:
Z_2:
p :- q
q :- ¬ r

Z_1:
r :- s
s

Let us compute $\mathbf{T}_Z$ in this order. That means $\mathbf{T}_Z(Z_1)$ is computed till a fixpoint is reached, then $\mathbf{T}_Z(Z_1 \cup Z_2)$ is computed till a fixpoint is reached and so on:
$\mathbf{T}_Z \uparrow 0 = \emptyset$
$\mathbf{T}_Z \uparrow 1$ = {s}
$\mathbf{T}_Z \uparrow 2$ = {s,r} = $\mathbf{T}_Z \uparrow 3 \Rightarrow$ A fixpoint is reached.
In this case the operator is monotonic, bacause $\mathbf{T}_Z(Z_1) \subseteq \mathbf{T}_Z(Z_1 \cup Z_2)$.

Definition 5 (Stratification for sets of normal clauses). Let S be a set of normal clauses. A stratification is a partition $S_1,...,S_k$ with k∈ ℕ of S such that

- For each relation symbol `p` there is a stratum S_i with i∈ ℕ, i≤k, such that all clauses of S containing `p` in their consequent are members of S_i.
  In this case one says that the relation symbol `p` is defined in stratum S_i.
- For each stratum S_j with j∈ ℕ, j≤k and for each positive literal A in the antecedents of members of S_j, the relation symbol of A is defined in a stratum S_i with i≤j.
- For each stratum S_j and for each negative literal ¬A in the antecedents of members of S_j, the relation symbol of A is defined in a stratum S_i with i<j. [1]

Definition 6 (Stratifiable set of normal clauses). A set of normal clauses is called stratifiable, if there exists a valid stratification of it. [1]

Z is an example for a stratifiable logic program.

1.2 Total Well-Founded Model of Unstratifiable Programs

All stratifiable logic programs have a total well-founded model. That means, each ground atom occurring in the Herbrand base of a stratifiable program is either ´true´ or ´false´ in the well-founded model (see [1]) of the program. There is no ground atom of the Herbrand base which is undefined in the well-founded model of the program.

But there are many useful unstratifiable programs, which also have a total well-founded model. For example, the program *P'* in Listing 1.2 computes the shortest distance between the node `anna` and all other nodes in the directed graph in Figure 1.1.

Listing 1.2: Program *P'*

```
Ac[ anna, 0 ]
Ac[ var Aq, n[ var D ] ]  :- knows[ var Person, var Aq ],
                             Ac[ var Person, var D ],
                             not ( Ach[ var Aq, var D ] )
Ach[ var Aq, var D ]      :- Ac[ var Aq, var D ]
Ach[ var Aq, n[ var D ] ] :- Ach[ var Aq, var D ]
knows[ anna,  bob ]
knows[ anna, chuck ]
knows[ bob,  chuck ]
knows[ chuck, anna ]
```

P' is not stratifiable, because there is no valid stratification for it.

If the function symbol *Ac* is defined in stratum P'_i with $i \in \mathbb{N}$, then the first and the second rule of *P'* are in the same stratum. There is a negative literal `not(Ach[var Aq,var D])` in the body of the second rule. That is why the function symbol *Ach* must be defined in a lower stratum as P'_i. Let the third and the fourth rule of *P'* be in stratum P'_j with $j \in \mathbb{N}$, $j<i$. But there is a positive literal `Ac[var Aq,var D]` in the body of the third rule. Therefore the function symbol *Ac* must be defined in a lower or in the same stratum as P'_j. See the dependency graph for the function symbols of *P'* in Figure 1.2.

But *P'* has a total well-founded model which is agnostic of the order of rule application.

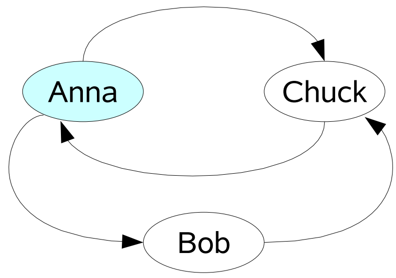

Figure 1.1: Directed graph for *P'*

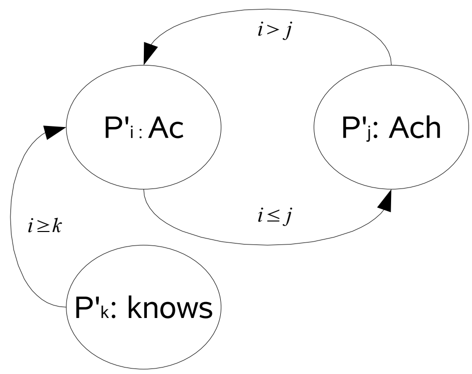

Figure 1.2: Dependency graph for the function symbols of *P'*

Let P'_H be the Herbrand instantiation of *P'*. It is infinite and consists, for example, of the following ground instances of the rules of *P'*.

Listing 1.3: P'_H is the Herbrand instantiation of *P'*

```
Ac[ anna, 0 ]
Ac[ anna, n[ 0 ] ] :- knows[ chuck, anna ],
                      Ac[ chuck, 0 ],
                      not ( Ach[ anna, 0 ] )
Ac[ bob, n[ 0 ] ] :-  knows[ anna, bob ],
                      Ac[ anna, 0 ],
                      not ( Ach[ bob, 0 ] )
...
Ach[ anna, 0 ]     :- Ac[ anna, 0 ]
Ach[ bob, 0 ]      :- Ac[ bob, 0 ]
...
knows[ anna,   bob ]
knows[ anna, chuck ]
knows[ bob,  chuck ]
knows[ chuck, anna ]
```

$\mathcal{HB}$ is also infinite and contains, for example, the following ground terms:
{ 0, anna, ...,
Ac[anna,0], ..., Ac[anna,anna], ..., Ac[0,0], ..., Ac[anna,n[0]], ...,
Ach[anna,0], ..., Ach[anna,anna], ..., Ach[0,0], ..., Ach[anna,n[0]], ...,
n[0], ..., n[anna], ..., n[n[0]], ...,
knows[anna,anna], ..., knows[anna,0], ..., knows[0,0], ..., knows[anna,n[0]], ... }

Let us compute the well-founded semantics for the rule instances in Listing 1.3.
$\mathbf{W}_P \uparrow 0 = \emptyset$

$\mathbf{T}_P \uparrow 1 = \{$ Ac[anna,0], knows[anna,bob], knows[anna,chuck], knows[bob,chuck], knows[chuck,anna] $\}$
$\mathbf{U}_P \uparrow 1 = \{$ Ac[chuck,0], Ac[bob,0] $\}$
$\mathbf{W}_P \uparrow 1 = \{$ Ac[anna,0], knows[anna,bob], knows[anna,chuck], knows[bob,chuck], knows[chuck,anna], ¬Ac[chuck,0], ¬Ac[bob,0] $\}$

$\mathbf{T}_P \uparrow 2 = \mathbf{T}_P \uparrow 1 \cup \{$ Ach[anna,0] $\}$
$\mathbf{U}_P \uparrow 2 = \mathbf{U}_P \uparrow 1 \cup \{$ Ach[bob,0] $\}$
$\mathbf{W}_P \uparrow 2 = \mathbf{W}_P \uparrow 1 \cup \{$ Ach[anna,0], ¬Ach[bob,0] $\}$

$\mathbf{T}_P \uparrow 3 = \mathbf{T}_P \uparrow 2 \cup \{$ Ac[bob,n[0]] $\}$

$\mathbf{U}_P \uparrow 3 = \mathbf{U}_P \uparrow 2 \cup \{$ Ac[anna,n[0]] $\}$
$\mathbf{W}_P \uparrow 3 = \mathbf{W}_P \uparrow 2 \cup \{$ Ac[bob,n[0]], $\neg$Ac[anna,n[0]] $\} = \mathbf{W}_P \uparrow 4 \Rightarrow$ A fixpoint is reached

Let $\mathcal{B}$ be the set of all ground terms appearing in Listing 1.3.
$\mathcal{B} = \{$ Ac[anna,0], knows[anna,bob], knows[anna,chuck], knows[bob,chuck], knows[chuck,anna], Ac[chuck,0], Ac[bob,0], Ach[anna,0], Ach[bob,0], Ac[bob,n[0]], Ac[anna,n[0]] $\} \subset \mathcal{HB}$

Each term of $\mathcal{B}$ is either true or false in the well-founded model for the rule instances in Listing 1.3: { Ac[anna,0], knows[anna,bob], knows[anna,chuck], knows[bob,chuck], knows[chuck,anna], $\neg$Ac[chuck,0], $\neg$Ac[bob,0], Ach[anna,0], $\neg$Ach[bob,0], Ac[bob,n[0]], $\neg$Ac[anna,n[0]] }

Listing 1.3 contains one false and one true instance of the second rule of *P'*. All the terms of both instances are defined in the well-founded model of the rule instances in Listing 1.3. All other rules of P'_H are also either true or false and all the terms appearing in these rules are also defined by the model just analogue to the terms appearing in the rule instances above. That is why the well-founded model of *P'* is infinite but still total.

1.3 Logic Programs vs. Xcerpt Programs

Every logic program can be transformed in an equivalent Xcerpt program. See Definitions 21 and 25. For example the program *P'* in Listing 1.2 is just equivalent to the following Xcerpt program *P*.

Listing 1.4: Xcerpt program *P*

```
CONSTRUCT Ac[ anna, 0 ] END
CONSTRUCT
  Ac[ var Aq, n[ var D ] ]
FROM
  and (
      knows[ var Person, var Aq ],
      Ac[ var Person, var D ],
      not ( Ach[ var Aq, var D ] )
  )
END
CONSTRUCT
  Ach[ var Aq, var D ]
FROM
```

```
  Ac[ var Aq, var D ]
END
CONSTRUCT
  Ach[ var Aq, n[ var D ] ]
FROM
  Ach[ var Aq, var D ]
END
CONSTRUCT knows[ anna,   bob  ] END
CONSTRUCT knows[ anna, chuck ] END
CONSTRUCT knows[ bob,  chuck ] END
CONSTRUCT knows[ chuck, anna ] END
```

P has the same Herbrand instantiation and Herbrand base as *P'*. See Listing 2.1. We shall see later that the well-founded models are also identic.

But there are some constructs in Xcrept, for instance `all`, `optional`, `without`, `except` which cannot be expressed in logic. There are Xcerpt programs which cannot be transformed in an equivalent logic program. That is why Xcrept programs require special treatment.

1.4 Grouping Stratification for Xcerpt Programs

Consider the following Xcerpt program *H*.

Listing 1.5: Xcerpt program *H*

```
CONSTRUCT
  f{ all var X }
FROM
  g{{ var X }}
END
CONSTRUCT
  g{ var Y }
FROM
  h{{ var Y }}
END
CONSTRUCT g{ a } END
CONSTRUCT h{ b } END
```

Rules with grouping constructs usually require that all rules they depend on are completely evaluated, because `all` expresses to collect all possible results, and a too early evaluation of a grouping construct might yield terms that do not properly reflect the meaning of the grouping constructs.[5] `f{a}` might be an answer for *H*, but it is not the complete answer intendet by the program.
That is why grouping stratification was introduced. Grouping stratification spacifies the order of rules in which they should be evaluated. Such an order for *H* can be defined as follows:
H_2:
f { all var X } ← g{{ var X }}

H_1:
g{ var Y } ← h{{ var Y }}
g{ a }
h{ b }

If we evaluate the clauses in this order, the answer for *H* will be complete: `f{a,b}`.

Definition 7 (Grouping stratification for Xcerpt programs). Given a program P consisting of rules/goals {R_1,...,R_m} (m≥1).

1. A rule R=t^c ←Q depends on a rule R'=$t^{c'}$←Q', if there exists a query term t^q in Q such that t^q simulation unifies in $t^{c'}$, i.e. simulation unification of t^q in $t^{c'}$ yields a non empty substitution set Σ

2. P is called grouping stratifiable, if there exits a partioning P_1,...,P_n (n≥1) of P such that for every stratum P_i (1≤i≤n) and every rule R∈ P_i holds:

   - if the rule head of R contains no grouping constructs and R depends on a rule R' then R' is either in the same stratum as R or in a lower stratum than R
   - if the rule head of R contains grouping constructs and R depends on a rule R' then R' is in a strictly lower stratum than R

The partition of P in P_1,...,P_n is called a grouping stratification of P, and the P_i are called grouping strata of P. [5]

H is an example for a grouping stratifiable Xcerpt program.

1.5 Intuitive Semantics of Grouping Unstratifiable Xcerpt Programs

There are many useful grouping unstratifiable Xcerpt programs which have an intuitive semantics. (We speak neither of well-founded semantics nor of total model, because these notions are not defined for the progarms with grouping constructs.) Consider for example the Xcerpt program G in Listing 1.6. G computes all possible paths from the node `anna` to each other node in the directed graph in Figure 1.1.

Listing 1.6: Xcerpt program G

```
CONSTRUCT Ac[ anna, path[ ] ] END
CONSTRUCT
  Ac[ var Aq, path[ all var Node, var Person] ]
FROM
  and (
    knows[ var Person, var Aq ],
    Ac[ var Person, path[[ optional var Node ]] ]
  )
END
CONSTRUCT knows[ anna,   bob ] END
CONSTRUCT knows[ anna, chuck ] END
CONSTRUCT knows[ bob,  chuck ] END
CONSTRUCT knows[ chuck, anna ] END
```

G is not grouping stratifiable, because there is no valid grouping stratification for it.

The head of the second rule of G contains a grouping construct '`all`' and the rule depends on itself, because the query term `Ac[var Person, path[[optional var Node]]]` in the body of the rule simulation unifies in the head of the same rule `Ac[var Aq, path[all var Node, var Person]]`. But the same rule cannot be defined in different strata. See the dependency graph for the function symbols of G in Figure 1.3.

But G has an intuitive semantics which is agnostic of the order of rule application.

Let G_H be the Herbrand instantiation of G (Definition 34). It is infinite and contains for example the following ground instances of the rules of G.

Listing 1.7: G_H is the Herbrand instantiation of G

```
CONSTRUCT  Ac[ anna, path[] ] END
CONSTRUCT
  Ac[ anna, path[ chuck ] ]
```

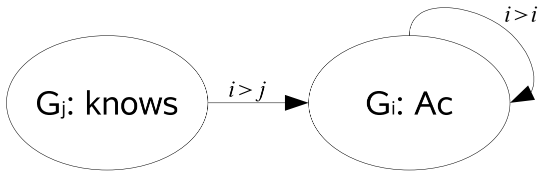

Figure 1.3: Dependency graph for the function symbols of G

```
FROM
  and (
    knows[ chuck, anna ],
    Ac[ chuck, path[] ]
  )
END
CONSTRUCT
  Ac[ bob, path[ anna ] ]
FROM
  and (
    knows[ anna, bob ],
    Ac[ anna, path[] ]
  )
END
...
CONSTRUCT knows[ anna,   bob ] END
CONSTRUCT knows[ anna, chuck ] END
CONSTRUCT knows[ bob,  chuck ] END
CONSTRUCT knows[ chuck, anna ] END
```

$\mathcal{HB}$ (Definition 13) is also infinite and contains for example the following ground terms:
{ anna, ...,
path[], path[anna], ..., path[path[anna]], ..., path[Ac[anna,path[anna]],bob], ...,
Ac[anna,path[]], ..., Ac[anna,anna], ..., Ac[path[anna],path[]], ...,
knows[anna,anna], ..., knows[anna,path[bob]], ..., knows[path[anna],path[bob]], ... }

Let $\mathcal{B}$ be the set of all ground terms appearing in Listing 1.7.
$\mathcal{B}$ = { Ac[anna,path[]], knows[anna,bob], knows[anna,chuck], knows[bob,chuck], knows[chuck,anna], Ac[chuck,path[]], Ac[anna,path[chuck]], Ac[bob,path[anna]] } $\subset \mathcal{HB}$

Each term of $\mathcal{B}$ is either true or false in the intuitive semantics for the rule instances in Listing 1.7: { Ac[anna,path[]], knows[anna,bob], knows[anna,chuck], knows[bob,chuck], knows[chuck,anna], ¬Ac[chuck,path[]], ¬Ac[anna,path[chuck]], Ac[bob,path[anna]] }

Listing 1.7 contains one false and one true instance of the second rule of G. Thus, both instances have an intuitive meaning. All other rules of G_H are also either true or false and all the terms appearing in these rules are also defined by the intuitive semantics just analogue to the terms appearing in the rule instances above. That is why G has an intuitive meaning which is infinite.

1.6 The Goal and the Structure of this Thesis

In this work we shall prove that every locally stratifiable Xcerpt program has a total well-founded model.

At the beginning we give the definitions of some Xcerpt notions which we use later. It is necessary, because Xcerpt programs have other properties as logic programs. For example, there are no atoms in Xcerpt and Xcerpt data and query terms must be treated differently.
In the third chapter well-founded semantics for Xcerpt programs without grouping constructs is defined.
In the fourth chapter we explain why we have chosen SLG-Resolution for evaluation of Xcerpt programs, introduce the algorithm computing SLG-Resolution which we intend to implement and give same examples of the evaluation of the algorithm.
We shall also define some new notions namely local stratification and local grouping stratification for Xcerpt programs and give some examples.
Afterwards we prove that every locally stratifiable Xcerpt program has a total well-founded model.

There are some new ideas we intend to work out in this paper, for instance

1. Notions for Xcerpt, such as Xcerpt signature, Herbrand universe, model relationship, dependency graph and so on
2. Well-founded semantics for Xcerpt programs without grouping constructs
3. Local stratification for Xcerpt programs
4. Proof that every locally stratifiable Xcerpt program has a total well-founded model
5. Local grouping stratification for Xcerpt programs

6. SLG-Algorithm for Xcerpt programs

Chapter 2

Xcerpt Notions

In this chapter we define Xcerpt notions such as signature, interpretation, model relationship, dependency graph and so on, which are be used in definitions, examples and a proof later. We shall also use some other notions, which are not defined in this chapter. In this case the meaning of these notions is the same as in logic (see [1]).

Definition 8 (**Xcerpt signature**)**.** An Xcerpt signature or vocabulary is a computably enumerable symbol set L=$\{\mathrm{Fun}_L^n\}$ with n$\in \mathbb{N}$, called n-ary function symbols of L. The 0-ary function symbols are called constants of L. The same function symbol may have different arities.

Definition 9 (**Signature of an Xcerpt program**)**.** A signature of an Xcerpt program is a computably enumerable symbol set L=$\{\mathrm{Fun}_L^n\}$ with n$\in \mathbb{N}$, of n-ary function symbols appearing in the program. The 0-ary function symbols are called constants of L. The same function symbol may have different arities.

The signature of the Xcerpt program P in Listing 1.4 consists of the following function symbols: { $\mathtt{Ac}^2$, $\mathtt{Ach}^2$, $\mathtt{knows}^2$, $\mathtt{n}^1$, $\mathtt{anna}^0$, $\mathtt{bob}^0$, $\mathtt{chuck}^0$, $\mathtt{0}^0$ }

The signature of the Xcerpt program G in Listing 1.6 contains the following function symbols: { $\mathtt{path}^n$ with n$\in \mathbb{N}$, $\mathtt{Ac}^2$, $\mathtt{knows}^2$, $\mathtt{anna}^0$, $\mathtt{bob}^0$, $\mathtt{chuck}^0$ }
The function symbol `path` may have different arities, for example `path[]`, `path[anna]` or `path[anna,bob,chuck]` are legal ground terms.

Let L be an Xcerpt signature.

Definition 10 (**L-term**)**.** An L-term is defined inductively as follows:

1. Each variable `X` is an L-term.

2. Each constant `c` of L is an L-term.

3. For each n$\in \mathbb{N}$, if `f` is an n-ary function symbol of L and $t_1,...,t_n$ are L-terms, then $f\{t_1,...,t_n\}$, $f\{\{t_1,...,t_n\}\}$, $f[t_1,...,t_n]$ and $f[[t_1,...,t_n]]$ are L-terms.

There are construct, query and data terms in Xcerpt. The differences between them are discribed in [5].

There are many other constructs in query and construct terms such as `optional`, `descendant`, `without`, but they are not important for this thesis (see [5]).

Definition 11 (Ground L-term). A ground L-term is an L-term without variables.

Definition 12 (Data term). A data term is a ground term without `optional`, `descendant`, `without`, `all`, double curly braces and double square brackets.

Definition 13 (Herbrand universe). The Herbrand Universe of an Xcerpt program P, denoted $\mathcal{HU}_L$, is the set of all ground L-terms, where L is the signature of P.

The Herbrand Universe includes both data and query terms. We have made this desicion because it is easier to define dependency relation (Definition 36) between all ground terms of an Xcerpt program as to consider this relation only between data terms of a program.

As there are no relation symbols in Xcerpt, the notions 'term' and 'atom' as well as 'Herbrand Universe' and 'Herbrand Base' have the same meaning for Xcerpt. See the example of Herbrand base for the Xcerpt program G in Listing 1.6 at page 10.

In most cases, the signature L is clear from the context, and we simply speak of terms, $\mathcal{HU}$ and $\mathcal{HB}$ without 'L'.

Definition 14 (Positive and negative literal, complement). If T is a term, both T and $\neg T$ are literals. The literal T is positive, the literal $\neg T$ is negative, and the two are a pair of complementary literals. The complement of T, written $\overline{T}$, is $\neg T$, the complement of $\neg T$, written $\overline{\neg T}$, is T.

Definition 15 (Ground literal). A ground literal is a literal without variables.

Definition 16. Let I be a set of ground literals. We define:

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

Definition 17 (**Consistent set of ground literals**). A set I of ground literals is consistent, iff $pos(I) \cap neg(I) = \emptyset$ holds, that is, there is no ground term T with $T \in I$ and $\neg T \in I$.

Definition 18 (**Partial interpretation**). A partial interpretation is a consistent set of data terms.

Let I be a partial interpretation and $D \subseteq \mathcal{HB}$ the set of all data terms of $\mathcal{HB}$. (Note $pos(I) \cup neg(I) \subseteq D$.)

Definition 19 (**Total interpretation**). I is called total, iff $pos(I) \cup neg(I) = D$ holds, that is, for each data term $d \in D$ either $d \in I$ or $\neg d \in I$.

Definition 20 (**Model relationship between an interpretation and a ground literal**). Let T be a ground literal. We define the model relationship between I and T as follows:

- If $T \in D$:
  - I satisfisies T or I is a model of T, denoted $I \vDash T$, iff $T \in pos(I)$.
  - I falsifies T or I is not a model of T, denoted $I \nvDash T$, iff $T \in neg(I)$.
  - T is undefined in I iff $T \notin pos(I) \cup neg(I)$.
- If T is a positive query term:
  - $I \vDash T$ iff $\exists \mathtt{d} \in D.\ T \preceq \mathtt{d}$ (see [6]) and $I \vDash \mathtt{d}$.
  - $I \nvDash T$ iff $\forall \mathtt{d} \in D.\ T \preceq \mathtt{d} \Rightarrow I \nvDash \mathtt{d}$.
  - T is undefined in I iff $\forall \mathtt{d} \in D.\ T \preceq \mathtt{d} \Rightarrow \mathtt{d}$ is undefined in I.
- If T is a negative query term:
  - $I \vDash T$ iff $\forall \mathtt{d} \in D.\ \overline{T} \preceq \mathtt{d} \Rightarrow I \nvDash \mathtt{d}$.
  - $I \nvDash T$ iff $\exists \mathtt{d} \in D.\ \overline{T} \preceq \mathtt{d}$ and $I \vDash \mathtt{d}$.
  - T is undefined in I iff $\forall \mathtt{d} \in D.\ \overline{T} \preceq \mathtt{d} \Rightarrow \mathtt{d}$ is undefined in I.

Definition 21 (**Xcerpt rule and goal**). Xcerpt rules and goals are of the following form:

```
CONSTRUCT contruct term
FROM query
```

```
END

GOAL contruct term
FROM query
END
```

where a construct term is constructed depending on the evaluation of a query. A query is a connection (using ANDs and ORs) of query terms [2]. A construct term is called the head and a query the body of the rule or goal.

Definition 22 (**Query**)**.** A query can be

- a query term
- a disjunction of queries
- a conjunction of queries
- a negation of a query

An Xcerpt rule with disjunction in the query can be transformed into an equivalent set of Xcerpt rules without disjunctions in the rule body. Therefore we only consider disjunction-free queries in this thesis.

Definition 23 (**Ground query**)**.** A ground query is a query without variables.

Definition 24 (**Model relationship between an interpretation and a ground query**)**.** Let Q be a ground query. We define the model relationship between I and Q as follows:

- If Q is a ground query term, then the model relationship is defined in Definition 20.
- If Q is a conjunction of ground queries, then
  - $I \models Q$ iff I satisfies all the conjuncts of Q.
  - $I \not\models Q$ iff I falsifies at least one conjunct of Q.
  - Q is undefined in I iff Q contains a query term which is undefined in I.
- If Q is a negation of a ground query Q', then
  - $I \models Q$ iff $I \not\models Q'$.
  - $I \not\models Q$ iff $I \models Q'$.

– Q is undefined in I iff Q contains a query term which is undefined in I.

Definition 25 (Xcerpt program). A finite sequence of zero or more Xcerpt rules, one or more Xcerpt goals and zero or more Xcerpt data terms is called an Xcerpt Program.

Definition 26 (Substitution). A substitution is a function ϑ, written in postfix notation, that maps variables to terms.

A substitution ϑ is represented by the finite set $\{X_1 \to T_1, ..., X_k \to T_k\}$ with $k \in \mathbb{N}$ where $\{X_1, ..., X_k\}$ is its domain and $\{T_1, ..., T_k\}$ with $T_i = X_i\vartheta$, $i \in \mathbb{N}$, $i \leq k$, is its codomain.

Definition 27 (Application of a substitution to a constant). A substitution ϑ applied to a constant `c` is an identical mapping, i.e. $\mathtt{c}\vartheta = \mathtt{c}$.

Definition 28 (Application of a substitution to a compound term). A substitution ϑ applied to a compound term is a homomorphous, i.e.
$f\{t_1,...,t_n\}\vartheta = f\{t_1\vartheta,...,t_n\vartheta\}$,
$f\{\{t_1,...,t_n\}\}\vartheta = f\{\{t_1\vartheta,...,t_n\vartheta\}\}$,
$f[t_1,...,t_n]\vartheta = f[t_1\vartheta,...,t_n\vartheta]$,
$f[[t_1,...,t_n]]\vartheta = f[[t_1\vartheta,...,t_n\vartheta]]$, $n \in \mathbb{N}$.

Definition 29 (Application of a substitution to a query). A substitution ϑ applied to a query Q is defined as follows:

- if Q is a query term, then the application of a substitution is defined in Definition 28
- if Q is a conjunction of queries $\mathtt{and}(Q_1, ..., Q_n)$, $i \in \mathbb{N}$, then $(\mathtt{and}(Q_1, ..., Q_n))\vartheta = \mathtt{and}(Q_1\vartheta, ..., Q_n\vartheta)$
- if Q is a negation of a query $\neg Q'$, then $(\neg Q')\vartheta = \neg(Q'\vartheta)$

Definition 30 (Application of a substitution to an Xcerpt rule). A substitution ϑ applied to an Xcerpt rule is defined as follows:

(CONSTRUCT c FROM q END)ϑ = CONSTRUCT cϑ FROM qϑ END,

where `c` is a construct term and `q` is a query.

Ground query term simulation is defined in [6]. The definition describes what conditions must be satisfied in order that a query term `q` simulates into a data term `d`, denoted $\mathtt{q} \preceq \mathtt{d}$. The construction of every term can be described as a tree. For instance the query term `q = a{{ b{{ d, without e }}, c{{ desc f{{ /.*/ }} }}` has the construction described by the left tree in Figure 2.1. If there is a data term `d` such that the construction of `q` can be found in `d`, we say that `q` simulates into `d`. See Figure 2.1.

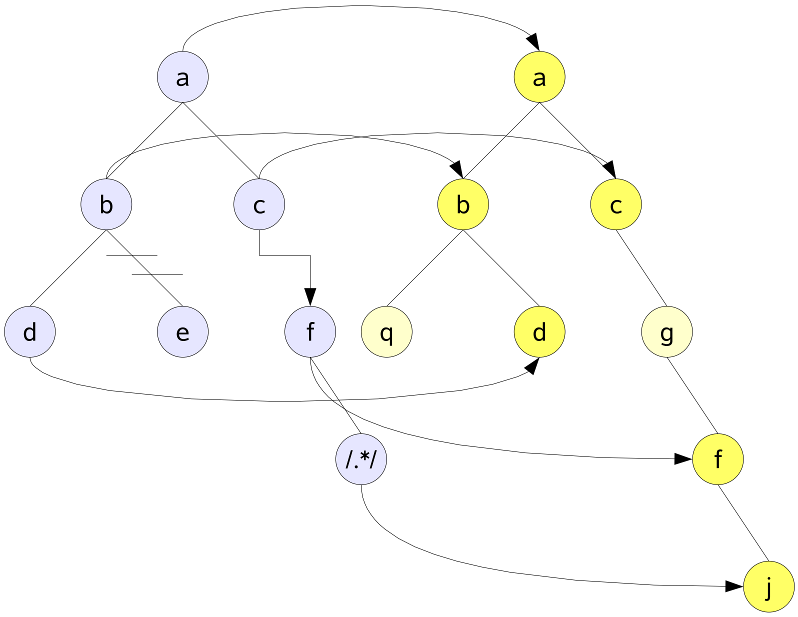

Figure 2.1: Simulation of a query term into a data term

Definition 31 (Simulation subsumption). A query term q_1 subsumes another query term q_2 if all data or query terms that q_2 simulates with are also simulated by q_1. [6]

Let P be an Xcerpt program.

Definition 32 (Instance of an Xcerpt rule, goal or term). Let R be a rule, a goal or a term in P. R' is an instance of R iff there exists a substitution ϑ with $R\vartheta$=R'.

Definition 33 (Ground instance of an Xcerpt rule or goal). An instance of an Xcerpt rule or goal is ground, if it does not contain any variables.

Definition 34 (Herbrand instantiation of an Xcerpt program). A Herbrand instantiation of an Xcerpt program P, denoted P_H, consists of all ground instances of all the rules and goals of P.

Consider be the Herbrand instantiation of the Xcerpt program P in Listing 1.4. It is infinite and consists, for example, of the following ground instances of the rules of P.

Listing 2.1: P_H is the Herbrand instantiation of P

```
CONSTRUCT Ac[ anna , 0 ] END
CONSTRUCT
   Ac[ anna , n[ 0 ] ]
FROM
```

```
  and (
    knows[ chuck, anna ],
    Ac[ chuck, 0 ],
    not ( Ach[ anna, 0 ] )
  )
END
CONSTRUCT
  Ac[ bob, n[ 0 ] ]
FROM
  and (
    knows[ anna, bob ],
    Ac[ anna, 0 ],
    not ( Ach[ bob, 0 ] )
  )
END
...
CONSTRUCT
  Ach[ anna, 0 ]
FROM
  Ac[ anna, 0 ]
END
CONSTRUCT
  Ach[ bob, 0 ]
FROM
  Ac[ bob, 0 ]
END
...
CONSTRUCT knows[ anna,   bob   ] END
CONSTRUCT knows[ anna,  chuck  ] END
CONSTRUCT knows[ bob,   chuck  ] END
CONSTRUCT knows[ chuck, anna   ] END
```

The terms `knows[anna,bob]`, `knows[anna,chuck]`, `knows[bob,chuck]`, `knows[chuck,anna]` and `Ac[anna,0]` do not depend on any other terms of P_H. They are facts, thay are always 'true'.
But this statement does not hold for the term `Ac[bob,n[0]]`. We must find out its truth value. In order to do this we have to know whether the terms `knows[anna,bob]`, `Ac[anna,0]` and `not(Ach[bob,0])` are 'true' or 'false'. One can say the truth value of

the term `Ac[bob,n[0]]` *depends* on the truth values of the terms `knows[anna,bob]`, `Ac[anna,0]` and `not(Ach[bob,0])`.
We formalise this intuition with the aid of the following definitions.

Definition 35 (**Dependency graph**). Let $R'_1,...,R'_n \in P_H$ with $n \in \mathbb{N}$.
The set of vertices V of the dependency graph G_{P_H}=(V,E,L) of P_H is the set of terms that appear either positively or negatively in at least one R'_i, $1 \leq i \leq n$.
$E \subseteq (V \times V)$ is the set of edges of the dependency graph and
L: $E \rightarrow \{1,-1\}$ is a labeling function on the set of edges, giving each edge either a positive or a negative value.

For each R'_i, $1 \leq i \leq n$, of the form $h \leftarrow l_1,...,l_k$, $k \in \mathbb{N}$, with a construct term h and query terms l_j, $1 \leq j \leq k$, k edges are constructed in the dependency graph.

- If l_j is a positive query term in the body of R'_i, then $(l_j,h) \in E$ with $L(l_j,h)=1$.
- If $l_j := \neg l'_j$ is a negative query term in the body of R'_i, then $(l'_j,h) \in E$ with $L(l'_j,h)=-1$.

For each query term l_j and each data term `d` appearing in a head of a rule instance in P_H the following edges are constructed in the dependency graph.

- If l_j is a positive query term in the body of R'_i and $l_j \preceq$`d`, then $(d,l_j) \in E$ with $L(d,l_j)=1$.
- If $l_j := \neg l'_j$ is a negative query term in the body of R'_i and $l'_j \preceq$`d`, then $(d,l'_j) \in E$ with $L(d,l'_j)=1$.

The dependency graph in Figure 2.2 illustrates some rule instances in Listing 2.1. The facts are lilac, all the other terms are white.
P_H is infinite. Therefore its dependency graph G_{P_H} would be also infinite.

Definition 36 (**Dependency relation between two ground terms**). Let $t_1,t_2 \in \mathcal{HB}$
t_1 depends negatively on t_2, iff there is a path from t_2 to t_1 in G_{P_H} involving at least one negative edge.
t_1 depends positively on t_2, iff there is a path from t_2 to t_1 in G_{P_H} without negative edges.

For example the term `Ac[anna,n[0]]` depends positively on the terms `knows[chuck,anna]` and `Ac[chuck,0]` (see the dependency graph in Figure 2.2) and it depends negatively on the terms `Ach[anna,0]` and `Ac[anna,0]`.

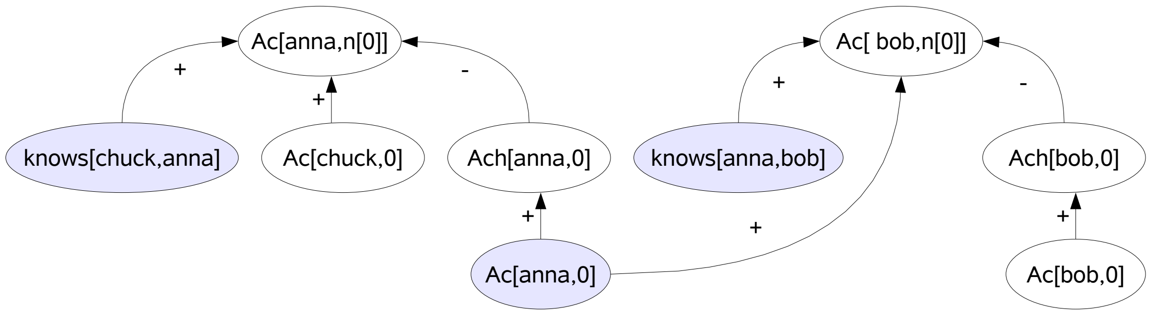

Figure 2.2: Dependency graph for some rule instances in Listing 2.1

Definition 37 (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.

The last definition is very important for this research. It is frequently used in the definitions and examples later.

Chapter 3

Well-Founded Semantics for Xcerpt Programs

In this chapter the well-founded semantics for Xcerpt programs is defined. The definition is analogue to the definition of this notion in logic.

Let P be an Xcerpt program, P_H its Herbrand instantiation, $\mathcal{HB}$ its associated Herbrand base, $D \subseteq \mathcal{HB}$ the set of all data terms of $\mathcal{HB}$ and I a partial interpretation.

Definition 38 (Unfounded set of ground terms). A set of data terms $U \subseteq D$ is unfounded with respect to P and I, if for each $\mathtt{c} \in U$ and for each ground instance R' in P_H with construct term $\mathtt{c}$ and query Q at least one of the following holds:

1. $I \not\models Q$
2. $\exists \mathtt{q} \in Q\ \forall \mathtt{d} \in D.\ \mathtt{q} \preceq \mathtt{d} \Rightarrow \mathtt{d} \in U$ or $I \not\models \mathtt{d}$

Definition 39 (Greatest unfounded set). The greatest unfounded set of P with respect to I, denoted $\mathbf{U}_P(I)$, is the union of all sets that are unfounded with respect to I. [3]

Definition 40 (Well-founded semantics for Xcerpt programs). The well-founded semantics of P is the 'meaning' represented by the least fixed point of the operator $\mathbf{W}_P$, denoted $\mathrm{lfp}(\mathbf{W}_P)$. We define the operator as follows:

- $\mathtt{c} \in \mathbf{T}_P(I)$, iff there is a ground instance R' in P_H with construct term $\mathtt{c}$ and query Q such that $I \models Q$
- $\mathbf{U}_P(I)$ is the greatest unfounded set of P with respect to I

- $\mathbf{W}_P(I) = \mathbf{T}_P(I) \cup \neg \mathbf{U}_P(\mathrm{I})$

Definition 40 looks similar to the definition of the well-founded semantics in logic. But the notions are different because of the definition of model relationship between an interpretation and a ground query. See Definitions 20 and 24. We have to take Xcerpt query terms and their simulation into data terms in account.

Let $\mathcal{PI} = \{\mathrm{I} \subseteq D \cup \overline{D} | \mathrm{I} \text{ is consistent}\}$ where $\overline{D} = \{\neg \mathrm{A} \mid \mathrm{A} \in D\}$.

An Xcerpt program P without grouping constructs can be easily transformed into a set of normal clauses. That is why the operators $\mathbf{T}_P$, $\mathbf{U}_P$ and $\mathbf{W}_P$ are monotonic and the operator $\mathbf{W}_P$ has a least fixpoint lfp($\mathbf{W}_P$) with
lfp($\mathbf{W}_P$) $= \cap\{\mathrm{I} \in \mathcal{PI} | \mathbf{W}_P(\mathrm{I}) = \mathrm{I}\} = \cap\{\mathrm{I} \in \mathcal{PI} | \mathbf{W}_P(\mathrm{I}) \subseteq \mathrm{I}\}$.
Moreover, lfp($\mathbf{W}_P$) is a partial interpretation and a partial model of P. [1]

Definition 41 (Well-founded model for Xcerpt programs). The well-founded model of an Xcerpt program P is its partial model lfp($\mathbf{W}_P$).

The well-founded model of the Xcerpt program P in Listing 1.4 is identical to the well-founded model of the logic program *P'* in Listing 1.2. See page 6.

Note that the well-founded semantics is not defined for the programs with grouping constructs.

Chapter 4

SLG-Resolution for Xcerpt Programs

In this chapter we explain why we have chosen SLG-Resolution for the evaluation of Xcerpt programs, introduce the algorithm computing SLG-Resolution which we intend to implement and give same examples of the evaluation of the algorithm. SLG-Resolution does not always terminate. If there are infinitely many solutions for a user's query, SLG-Resolution does not terminate. We give two examples of the evaluation of SLG-Resolution. The first one terminates, the second one does not terminate.
See appendix A for the delailed comments and the code of the algorithm.

4.1 Advantages of SLG-Resolution

SLG means **L**inear resolution with **S**election function for **G**eneral logic programs. As an evaluation strategy it has the following advantages:

1. Negative terms in the body of a rule are allowed.
2. Answer sharing between subgoals that are variants of each other, a subgoal is guaranteed to be evaluated only once.
3. Top-down computation, only relevant answers are computed.
4. Maximum freedom in control strategies. SLG resolution allows a programmer or an implementer to choose an arbitrary computation rule for selecting a literal from the rule body and to choose an arbitrary strategy for selecting which transformation to apply. [4]
   In this paper we choose the first literal of a query and apply the rules whose heads unify with the new query in the order they appear in the program.

4.2 Description of the SLG-Algorithm

4.2.1 SLG-Resolution Tree

The algorithm computing the SLG-Resolution builds an *SLG-Resolution Tree*. The root of the tree is the user's query. Each node of the tree can have zero, one or many children, but it has exactly one parent. The parent of the root is nil. Each *node* consists of nested lists of goals (see below). Each goal is either a solution or a look-up goal.

All *solution goals* and their solutions are saved in a *table* (see Subsection 4.2.3). A goal, which unifies with no solution goal in the table, is a solution goal.

A goal, which unifies with a solution goal in the table, is a *look-up goal*. It uses the solutions of the solution goal.

The children of a solution goal are the bodies of all rule instances, the head of which unifies with this solution goal. The query terms of the body of such a rule instance are saved in *list_n*. We need the number n in order to distinguish between the lists. `n` is set to 0 at the beginning of the algorithm.

A child of a node beginning with a solution goal g_1 is the copy of the node but with list_n for the goal g_1 instead of g_1. g_1 can also appear within a list. That is why the lists within a node can be nested.

The children of a look-up goal are all solutions for the respective solution goal.

A child of a node beginning with a look-up goal is the copy of the node without the look-up goal.

The goals of the node are modified in the child of a node, because there are some *variable bindings* yield by the simulation unification of the first goal of the node into a solution (if the first goal of the node is a look-up goal) or the head of a rule instance (if the first goal of the node is a solution goal).

In order to distinguish between the children of a goal, which have already been processed, and other children a *pointer* is saved for each goal. The pointer points to the last finished child. If a goal is new, its pointer is set to -1. Every time a look-up goal uses a solution or a solution goal uses a rule instance, its pointer increases. Therefore each solution is computed only once.

A goal which unifies with a solution goal in the table is a look-up goal. If there are some solutions for this solution goal in the table, the look-up goal uses them. If the solution goal is not completed yet, the node beginning with the look-up goal is *suspended*. It waits, till the solution goal is completed and uses its new solutions.

If the child of a node is not empty and its goals do not contradict to the solutions saved in the table, the child of the node is saved in the list_of_nodes and the algorithm is called recursively with this new node. That is why the algorithm executes depth first-search.

4.2.2 Auxiliary SLG-Resolution Tree

If the child of a node begins with a negative goal the negation of which is a solution goal, a new SLG resolution tree is initialized. The root of the new tree is the solution goal. The negation of the solution goal in the old tree is a look-up goal. If the auxiliary resolution tree does not find any solutions and at least one of its nodes is suspended, the node in the old tree beginning with the negative look-up goal is also suspended.

It is enough to find at least one solution for the root of the auxiliary tree, to say that the negative look-up goal in the main tree is false. But there may be a positive look-up goal waiting for the root of the auxiliary tree. It needs all the solutions of the solution goal. That is why all solutions for all solution goals are computed.

4.2.3 Data Structures of the Algorithm

All the nodes of all trees with pointer to their parents are saved in the *list_of_nodes*. At the beginning the list_of_nodes consists of the root of the tree only. The root of the tree is the user's query.

All solution goals and their solutions are saved in a *table*. The table is a list of triples. Each triple consists of the following fields:

1. A solution goal.
   Note that the user's query is always the first solution goal in the table, because the table is empty at the beginning of the algorithm.

2. The list of solutions for this solution goal.
   If all the goals a solution goal g_1 depends on are true, list_n for the goal g_1 is empty. The algorithm deletes list_n and all the empty lists list_n appears within.
   If all the goals g_1 depends on are true, a solution for g_1 is found. The algorithm propagates variable bindings upwards beginning from the node containing an empty list until the node beginning with g_1 and puts the solution for g_1 in the table. If there is no variable binding, that is g_1 is a fact, true is saved as a solution for g_1. Variable bindings are proragated every time an empty list is deleted.

3. 'true' if the solution goal is completed or 'false' otherwise.

4.2.4 The Two Phases of the Evaluation of the Algorithm

In the first phase of the evaluation all paths are computed and all suspended nodes are detected.

If a path ends with an empty node, the path is completed. But no node in the path is deleted because it can contain nodes beginning with a look-up goal which has not used all the solutions of the respective solution goal yet.

If a path is completed or ends with a suspended node, the algorithm walks bottom-up through the tree searching for the nodes which have children which have not been processed yet and calls itself recursively with a child of such a node. If the current node has no children and begins with a solution goal, the solution goal is completed. If the parent of the root of the tree (nil) is reached, the algorithm tests whether the root of the tree is the user's query. In this case the second phase of the algorithm begins.

Otherwise there is a negative look-up goal in the other tree waiting for the solutions of the root of the actual tree. The algorithm processes this negative look-up goal.

In the second phase of the evaluation all suspended nodes in the list_of_nodes are computed. If there are many nodes beginning with a look-up goal waiting for the same solution goal, the node beginning with the more special look-up goal is evaluated first.

A positive look-up goal is always more special than a negative look-up goal.

If both look-up goals are positive or both are negative, then the goal which is an instance of another one is the more special.

Listing 4.1: Xcerpt Program *P"*

```
CONSTRUCT Ac[ anna, 0 ] END
CONSTRUCT
  Ac[ var Aq, var Depth + 1 ]
FROM
  and (
     knows[ var Person, var Aq ],
     Ac[ var Person, var Depth ],
     not ( Ac[ var Aq, var D ] where var D <= var Depth )
  )
END
CONSTRUCT knows[ anna,   bob ] END
CONSTRUCT knows[ anna, chuck ] END
CONSTRUCT knows[ bob,  chuck ] END
```

```
CONSTRUCT knows[ chuck , anna ] END
```

Consider the Xcerpt program *P”* in Listing 4.1 which is just a variant of the program *P* in Listing 1.4. There can be two nodes in the list_of_nodes such that $node_1$ begins with `not(Ac[chuck,var D1] where var D1<1)` and $node_2$ begins with `not(Ac[chuck,var D2] where var D2<2)`. Both look-up goals wait for the same solution goal `Ac[chuck,var D]`. If the solution goal is completed, there is no solution for it and we evaluate $node_2$ first and then $node_1$, there will be two solutions for the solution goal: `Ac[chuck,2]` and `Ac[chuck,1]`. But the first solution is wrong. If we evaluate $node_1$ first and then $node_2$, there will be only one solution `Ac[chuck,1]`, because `not(Ac[chuck,var D2] where var D2<2)` will be false.

If there is a look-up goal in the list_of_nodes which has not used all the solutions of its solution goal, it uses them. That is why no node will be deleted before the second phase of the algorithm is completed. Infinite loops are possible but without this part of the algorithm the set of solutions for the user’s query can be incomplete.

At the end all the solutions for the user’s query are printed out and the list_of_nodes and the table are deleted.

4.3 Examples of the Evaluation of the Algorithm

The numbers at the beginning of every node of the tree denote the order in which the nodes have been evaluated.

Each node begins either with a solution or with a look-up goal. If it is a look-up goal, the number of the respective solution goal is specified. For example look-up goal 1: `Ac[chuck,var D1]` uses the solutions of solution goal 1: `Ac[chuck,var D]`.

4.3.1 SLG-Resolution for the Xcerpt Program *P*

Let us compute the SLG-Resolution for *P* in Listing 1.4 with the algorithm in Listing A.1. The program *P* computes the shortest distance between the node `anna` and all other nodes in the directed graph in Figure 1.1.

The result of the first phase are the trees in Figures 4.1, 4.2 and 4.3.

The result of the second phase are the trees in Figures 4.4, 4.5, 4.6, 4.7 and 4.8.

The algorithm saves all found solutions in Table 4.1.

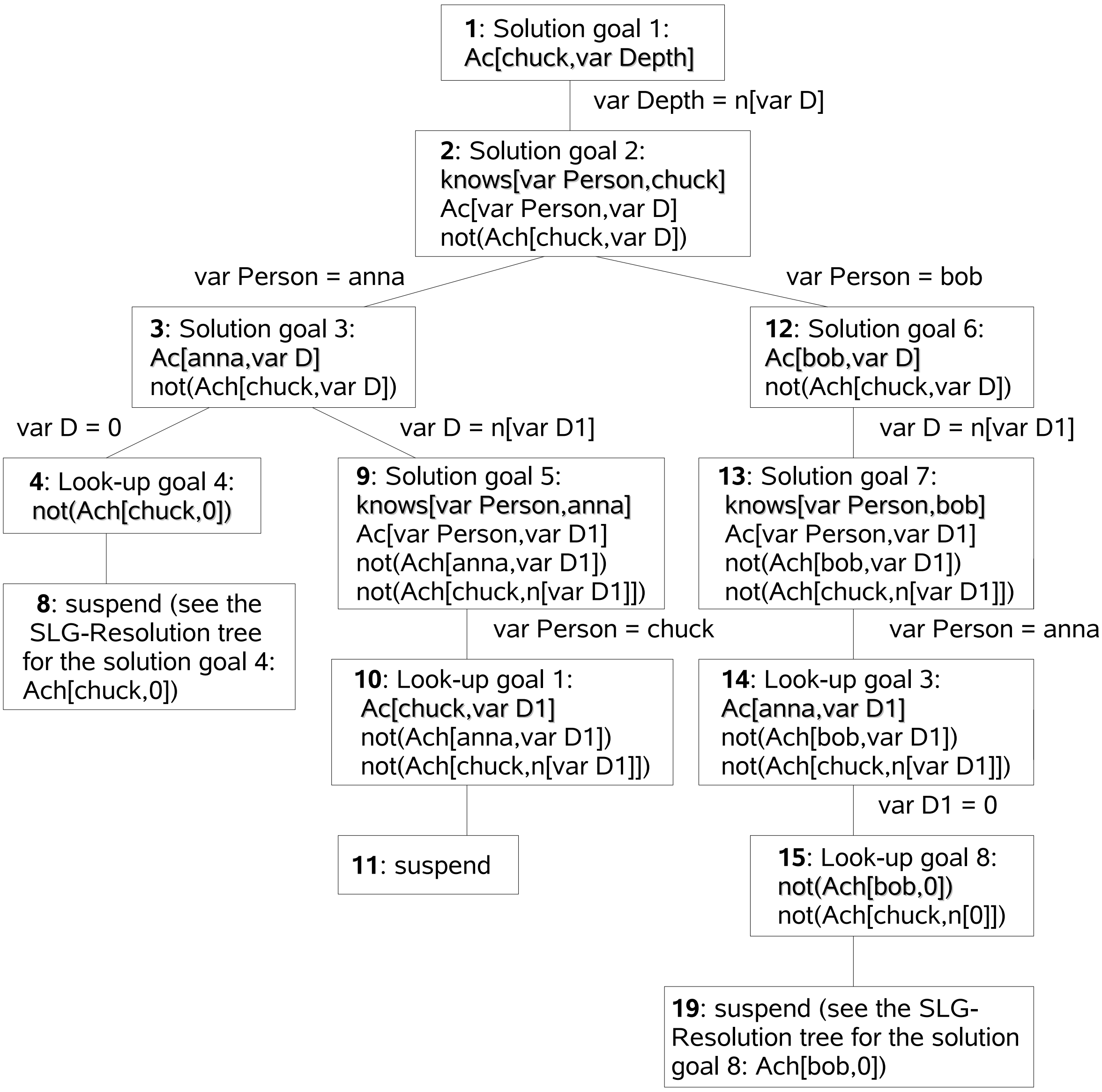

Figure 4.1: The SLG-Resolution tree for the solution goal `Ac[chuck,var Depth]` after the first phase of the algorithm

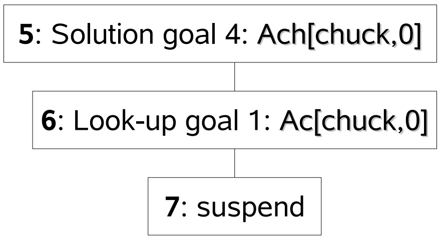

Figure 4.2: The auxiliary SLG-Resolution tree for the solution goal `Ach[chuck,0]` after the first phase of the algorithm

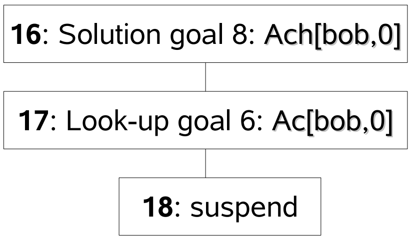

Figure 4.3: The auxiliary SLG-Resolution tree for the solution goal `Ach[bob,0]` after the first phase of the algorithm

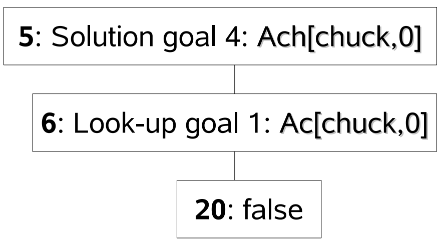

Figure 4.4: The auxiliary SLG-Resolution tree for the solution goal `Ach[chuck,0]` after the second phase of the algorithm

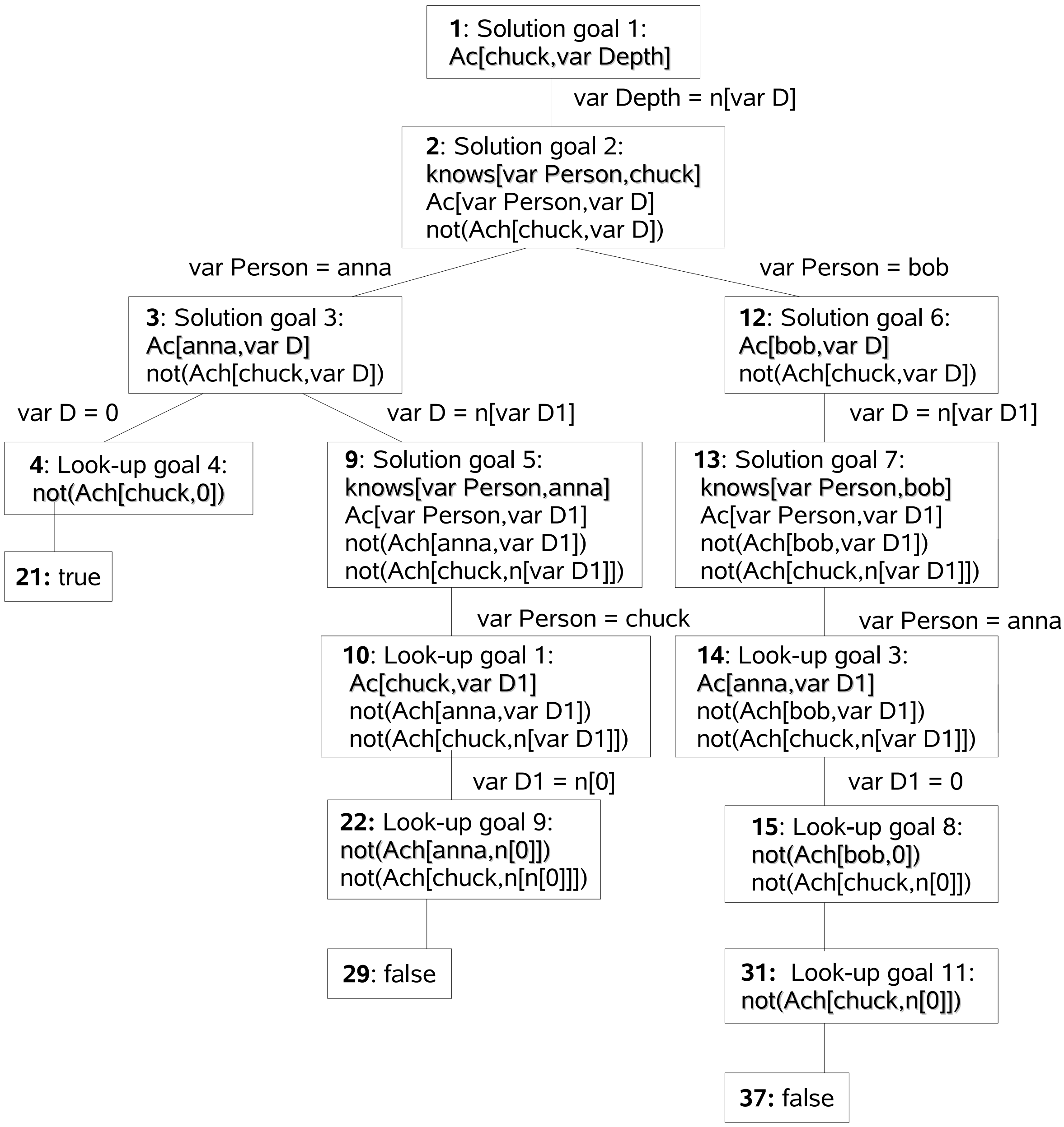

Figure 4.5: The SLG-Resolution tree for the solution goal `Ac[chuck,var Depth]` after the second phase of the algorithm

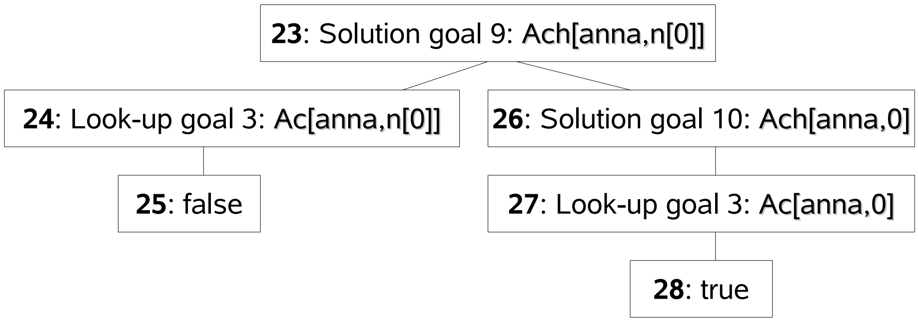

Figure 4.6: The auxiliary SLG-Resolution tree for the solution goal `Ach[anna,n[0]]` after the second phase of the algorithm

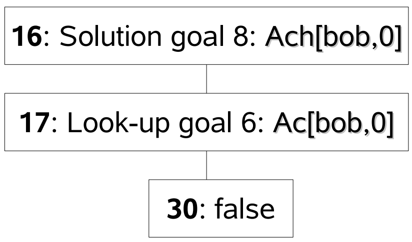

Figure 4.7: The auxiliary SLG-Resolution tree for the solution goal `Ach[bob,0]` after the second phase of the algorithm

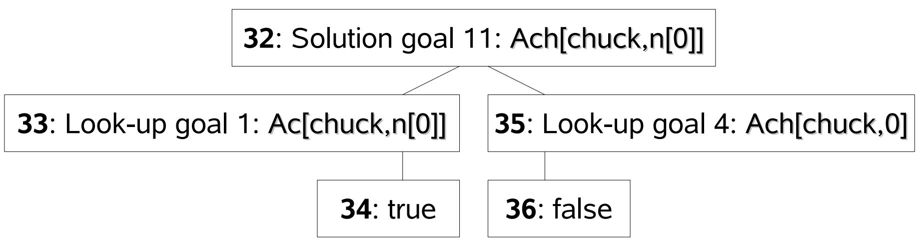

Figure 4.8: The auxiliary SLG-Resolution tree for the solution goal `Ach[chuck,n[0]]` after the second phase of the algorithm

Nr	Solution goals	Answers
1	Ac[chuck,var Depth]	Ac[chuck,n[0]]
2	knows[var Person,chuck]	knows[anna,chuck], knows[bob,chuck]
3	Ac[anna,var D]	Ac[anna,0]
4	Ach[chuck,0]	
5	knows[var Person,anna]	knows[chuck,anna]
6	Ac[bob,var D]	Ac[bob,n[0]]
7	knows[var Person,bob]	knows[anna,bob]
8	Ach[bob,0]	
9	Ach[anna,n[0]]	true
10	Ach[anna,0]	true
11	Ach[chuck,n[0]]	true

Table 4.1: Table of the solution goals and their solutions for the Xcerpt program P

4.3.2 SLG-Resolution for the Xcerpt Program G

Let us compute the SLG-Resolution for G in Listing 1.6 with the algorithm in Listing A.1. The program G computes all possible paths from the node `anna` to each other node in the directed graph in Figure 1.1.

The result of the first phase is the tree in Figure 4.9.

The result of the second phase is the tree in Figure 4.10.

The algorithm saves all found solutions in Table 4.2.

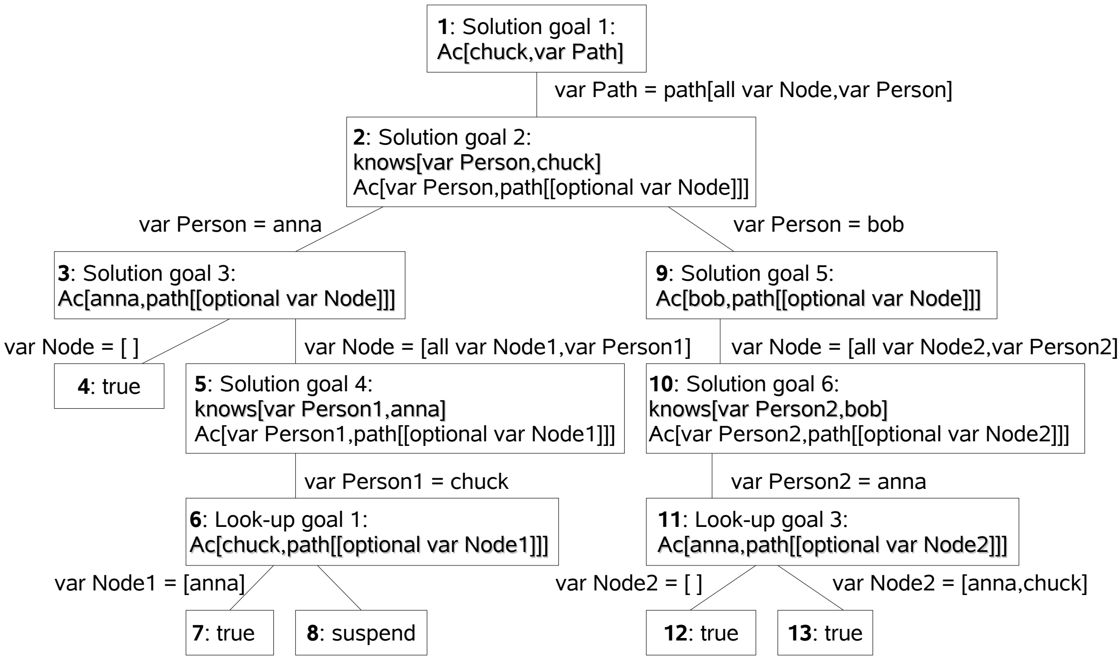

Figure 4.9: The SLG-Resolution tree for the solution goal `Ac[chuck,var Path]` after the first phase of the algorithm

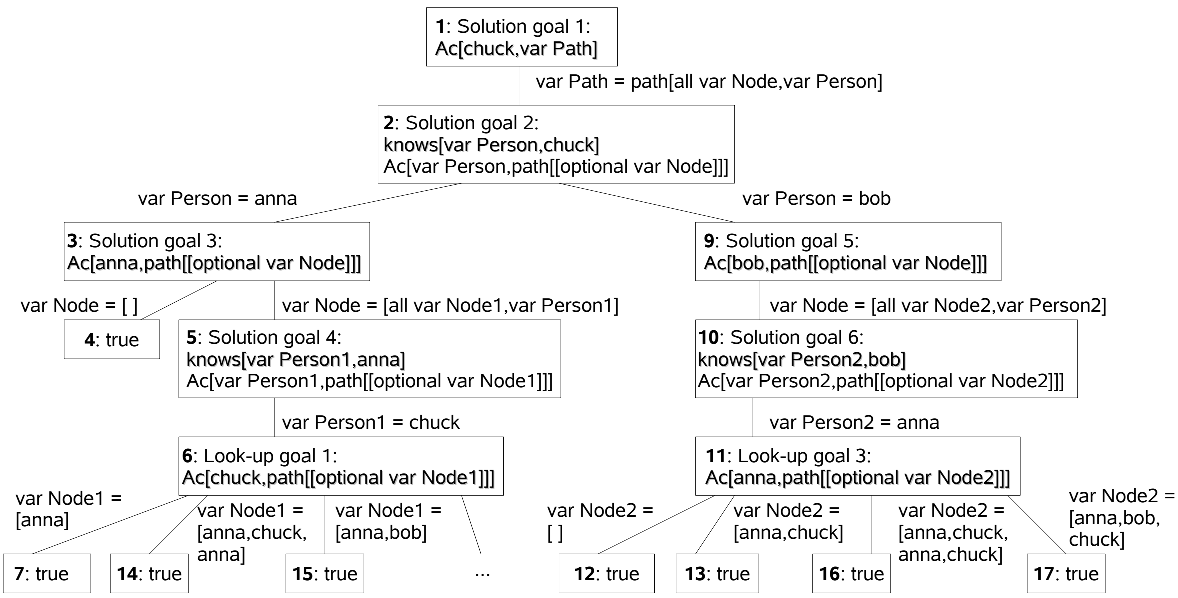

Figure 4.10: The SLG-Resolution tree for the solution goal `Ac[chuck,var Path]` during the second phase of the algorithm

Nr	Solution goals	Answers
1	Ac[chuck,var Path]	Ac[chuck,path[anna]], Ac[chuck,path[anna,chuck,anna]], Ac[chuck,path[anna,bob]], Ac[chuck,path[anna,chuck,anna,bob]], Ac[anna,path[anna,chuck,anna,chuck,anna]], Ac[anna,path[anna,bob,chuck,anna]], Ac[anna,path[anna,chuck,anna,bob,chuck,anna]], ...
2	knows[var Person,chuck]	knows[anna,chuck], knows[bob,chuck]
3	Ac[anna,var Path]	Ac[anna,path[ ]], Ac[anna,path[anna,chuck]], Ac[anna,path[anna,chuck,anna,chuck]], Ac[anna,path[anna,bob,chuck]], Ac[anna,path[anna,chuck,anna,bob,chuck]], ...
4	knows[var Person,anna]	knows[chuck,anna]
5	Ac[bob,var Path]	Ac[bob,path[anna]], Ac[bob,path[anna,chuck,anna]], ...
6	knows[var Person,bob]	knows[anna,bob]

Table 4.2: Table of the solution goals and some of their solutions for the Xcerpt program G

Chapter 5

Local Stratification for Xcerpt Programs

In this chapter local stratification for Xcerpt programs is defined and its connection to the dependency graph and the well-founded semantics of Xcerpt programs is explained. If an Xcerpt program is locally stratifiable there is no negative cycle in its dependency graph and its well-founded model is always total. The inversion of this statement does not hold. If an Xcerpt program has a total well-founded model it can still be locally unstratifiable.

Let P be an Xcerpt Program, P_H its Herbrand instantiation and $\mathcal{HB}$ its associated Herbrand base.

Definition 42 (**Local stratification for Xcerpt programs**). Local stratification of P is a partitioning of the ground terms of $\mathcal{HB}$ into a set of strata $\{P_1,...,P_k\}$ with $k \in \mathbb{N}$ such that for each ground instance R' in P_H the following holds:

1. if a query term $\mathtt{q} \in P_i$ with $i \in \mathbb{N}$, $i \leq k$ appears positively within the body of R', then the head of R' is in P_i or in a higher stratum

2. if a query term $\mathtt{q} \in P_i$ 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 P_i

3. if a ground query term `q` simulation unifies into a data term `d`, then `q` is in a higher, or the same stratum as `d`

Definition 43 (**Locally stratifiable Xcerpt program**). An Xcerpt program is locally stratifiable iff there exists a valid local stratification for it.

Consider the following Xcerpt program W.

Listing 5.1: Xcerpt program W

```
CONSTRUCT
   b{ d{ c } }
FROM
   a[ ]
END
CONSTRUCT
   a[ ]
FROM
   not( b{{ desc c }} )
END
```

Because of the second condition of Definition 42, `a[ ]` is in a higher stratum then `b{{desc c}}`. If there was no third condition in Definition 42, one could put the terms `a[ ]` and `b{d{c}}` in the same stratum:

W_2: a[ ], b{d{c}}

W_1: b{{desc c}}

But `b{{desc c}}`$\preceq$`b{d{c}}`. Thus `a[ ]` depends negatively on both `b{{desc c}}` and `b{d{c}}`. That is why `b{{desc c}}` must be in a higher, or the same stratum as `b{d{c}}`:

W_2: a[ ]

W_1: b{{desc c}}, b{d{c}}

But because of the first condition of Dedinition 42 the term `a[ ]` must be in the same or lower stratum as `b{d{c}}`. It is a contradiction to the second condition of the definition. That is why the program W is locally unstratifiable.

Note that every stratifiable Xcerpt program is also locally stratifiable. But the inversion of this statement does not hold. The program P in Listing 1.4 is an example for an Xcerpt program which is not stratifiable. We show in this chapter that P is locally stratifiable. Thus the set of all stratifiable Xcerpt programs is a real subset of the set of all locally startifiable Xcerpt programs. See Figure 5.1.

Note also that a program is always locally stratifiable if there is no negative cycle in the dependecy graph of the program. And vica versa: if the dependency graph of an Xcerpt program contains no negative cycle it is locally stratifiable.
See Figure 5.2 for the dependency graph of W.

Consider the Xcerpt program P in Listing 1.4. P computes the shortest distance between the node `anna` and all other nodes in the directed graph in Figure 1.1.

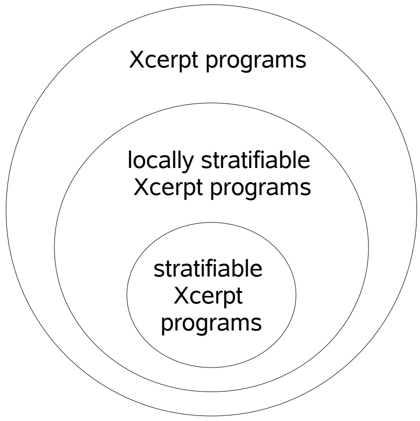

Figure 5.1: Stratification vs. Local Stratification

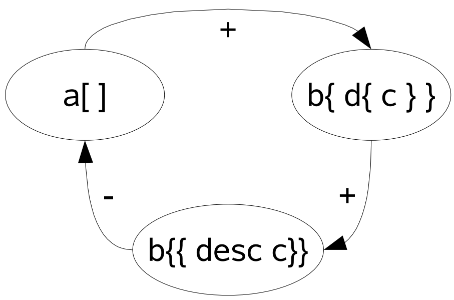

Figure 5.2: Dependency graph for W

The dependency graph of some rule instances of P has no negative cycle. See Figure 2.2. The whole dependency graph of P is infinite, but it does not contain a negative cycle because each term of a higher level of the dependency graph depends negatively on the terms in the lower levels of the graph and positively on the terms in the same or lower levels of the graph. See Figure 5.3. Thus P is locally stratifiable.

There is a valid local stratification for P. It is infinite.

$\mathcal{HB}$ contains also some 'illegal' terms, for instance `knows[0,0]` is illegal, because both parameters of a term with the outermost function symbol *knows* must be members of the set *Persons={anna,chuck,bob}*. *anna* is also an illegal term, bacause it appears only within some function symbol (*Ac*, *Ach* or *knows*) in P_H.
Illegal terms are always false in the well-founded model of a progam.

Each stratum contains both legal and illegal terms. The number of legal terms in each stratum is finite. The number of illegal terms in each stratum is infinite. That is why each stratum is infinite.

Condition 1: Because of the second rule of P the terms with the outermost function symbol *Ach* must be in a lower stratum as the terms with the outermost function symbol *Ac* if the number of nested function symbols n in the second term of the former terms is smaller as the number of nested function symbols n in the second term of the latter terms.
Condition 2: Because of the third rule of P the terms with the outermost function symbol *Ac* must be in the same or lower stratum as the terms with the outermost function symbol *Ach* if the number of nested function symbols n in the second term of the former terms is the same as the number of nested function symbols n in the second term of the latter terms.
Condition 3: Because of the second and the fourth rule of P the terms with the outermost function symbol *Ac* or *Ach* must be in the same or lower stratum as the terms with the same outermost function symbol if the number of nested function symbols n in the second term of the former terms is smaller as the number of nested function symbols n in the second term of the latter terms.

Conclusion: The terms with the outermost function symbol *Ac* or *Ach* and the same number of nested function symbols n in the second term are in the same stratum. See Figure 5.3. P_1 contains some other illegal terms which are not restricted by the conditions.

...

P_3:

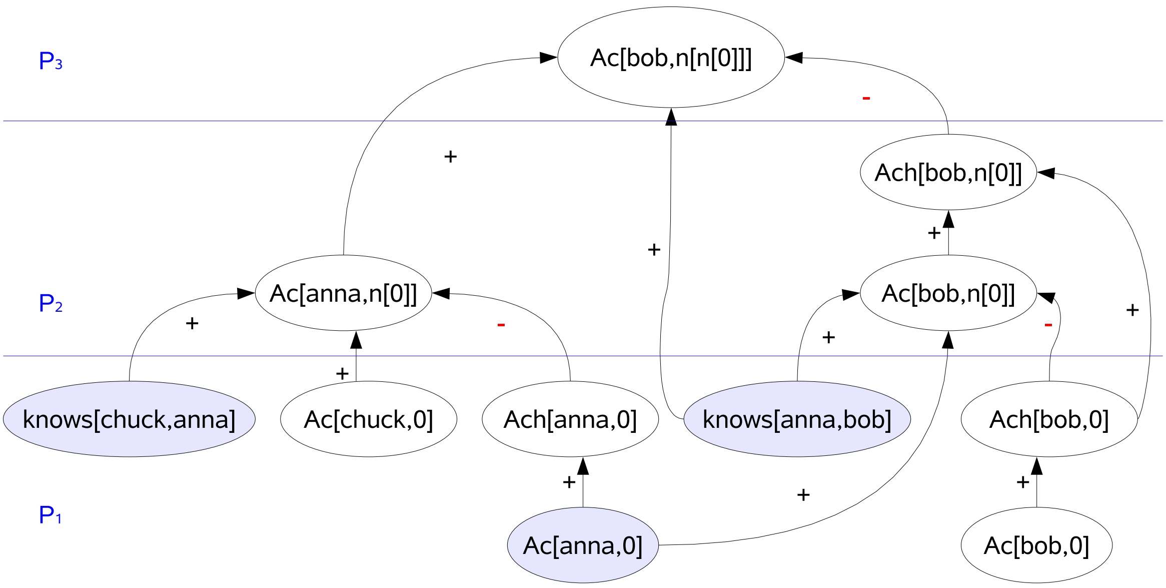

Figure 5.3: Dependency graph and local stratification for some rule instances of P

Legal terms:

Ac[anna,n[n[0]]], Ac[bob,n[n[0]]], Ac[chuck,n[n[0]]],

Ach[anna,n[n[0]]], Ach[bob,n[n[0]]], Ach[chuck,n[n[0]]]

Illegal terms:

Ac[anna,n[n[anna]]], ..., Ac[0,n[n[anna]]], ..., Ac[0,n[n[0]]], ..., Ac[n[0],n[n[anna]]], ..., Ac[n[n[0]],n[n[anna]]], ...

Ach[anna,n[n[anna]]], ..., Ach[0,n[n[anna]]], ..., Ach[0,n[n[0]]], ..., Ach[n[0],n[n[anna]]], ..., Ach[n[n[0]],n[n[anna]]], ...

P_2:

Legal terms:

Ac[anna,n[0]], Ac[bob,n[0]], Ac[chuck,n[0]],

Ach[anna,n[0]], Ach[bob,n[0]], Ach[chuck,n[0]]

Illegal terms:

Ac[anna,n[anna]], ..., Ac[0,n[anna]], ..., Ac[0,n[0]], ..., Ac[n[0],n[anna]], ..., Ac[n[n[0]],n[anna]], ...,

Ach[anna,n[anna]], ..., Ach[0,n[anna]], ..., Ach[0,n[0]], ..., Ach[n[0],n[anna]], ..., Ach[n[n[0]],n[anna]], ...

P_1:

Legal terms:

Ac[anna,0], Ac[bob,0], Ac[chuck,0],

Ach[anna,0], Ach[bob,0], Ach[chuck,0],

knows[anna,anna], knows[anna,bob], knows[anna,chuck],
knows[bob,bob], knows[bob,anna], knows[bob,chuck],
knows[chuck,chuck], knows[chuck,anna], knows[chuck,bob]
Illegal terms:
Ac[anna,anna], ..., Ac[0,anna], ..., Ac[0,0], ..., Ac[n[0],anna], ..., Ac[n[n[0]],anna], ...,
Ach[anna,anna], ..., Ach[0,anna], ..., Ach[0,0], ..., Ach[n[0],anna], ..., Ach[n[n[0]],anna], ...,

0, anna, ...,
n[anna], ...,
knows[anna,0], ..., knows[0,anna], ..., knows[0,0], ..., knows[anna,n[0]], ...

As we have already shown at page 6, P has a total well-founded model. Every locally stratifiable Xcerpt program has a total well-founded model. The statement will be proven in Chapter 7 of this thesis.

If an Xcerpt program is locally unstratifiable, it can have a total or a partial well-founded model.

For example the Xcerpt program A in Listing 5.2 has a total well-founded model `g[a]` and it is locally unstratifiable because there is a negative cycle in the dependency graph of the program. See Figure 5.4.

Listing 5.2: Xcerpt program A

```
CONSTRUCT g[ a ] END
CONSTRUCT
   g[ a ]
FROM
   not ( g[ a ] )
END
```

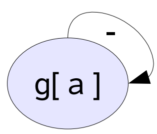

Figure 5.4: Dependency graph for A

The Xcerpt program B in Listing 5.3 is also locally unstartifiable (it has the same dependency graph as A but the node `g[a]` is white because it is not a fact) and it has a partial well-founded model $\{\oslash\}$.

	Total well-founded model	Partial well-founded model
Locally stratifiable Xcerpt programs	+, for example Xcerpt program P in Listing 1.4	-, see Chapter 7 of this thesis
Locally unstratifiable Xcerpt programs	+, for example Xcerpt program A in Listing 5.2	+, for example Xcerpt program B in Listing 5.3

Table 5.1: Local strtification and Well-founded model

Listing 5.3: Xcerpt program B

```
CONSTRUCT
   g[ a ]
FROM
   not ( g[ a ] )
END
```

Figure 5.1 can be extended. The set of all locally stratifiable Xcerpt programs is a real subset of the set of all Xcerpt programs which have a total well-founded model. See Figure 5.5.

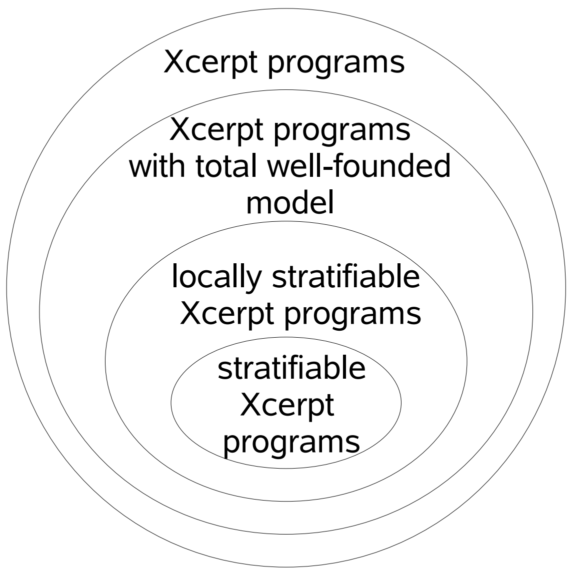

Figure 5.5: Local Stratification vs Stratification and Well-founded Semantics

Table 5.1 discribes the connection between the notions 'local stratification' and 'well-founded semantics'.

Chapter 6

Local Grouping Stratification for Xcerpt Programs

In this chapter the local grouping stratification for Xcerpt programs is defined, an example for an Xcerpt program which is not grouping stratifiable but is locally grouping stratifiable is considered and its intuitive meaning is analysed.

Let P be an Xcerpt program, P_H its Herbrand instantiation and $\mathcal{HB}$ its associated Herbrand base.

Definition 44 (**Local grouping stratification for Xcerpt programs**)**.** Local grouping stratification of P is a partitioning of the ground terms of $\mathcal{HB}$ into a set of strata $\{\mathrm{P}_1,...,\mathrm{P}_k\}$, k$\in \mathbb{N}$, such that for each rule R of P and every ground instance R' of R the following holds:

1. if the head of R contains no grouping constructs and the construct term of R' is in P_i with i$\in \mathbb{N}$, i$\leq$k, then the query terms of the body of R' are in P_i or in a lower stratum
2. if the head of R contains grouping constructs and the construct term of R' is in P_i with i$\in \mathbb{N}$, i$\leq$k, then the query terms of the body of R' are in a strictly lower stratum than P_i

Definition 45 (**Locally grouping stratifiable Xcerpt program**)**.** An Xcerpt program is locally grouping stratifiable iff there exists a valid local grouping stratification for it.

Note that every grouping stratifiable Xcerpt program is also locally grouping stratifiable. But the inversion of this statement does not hold. The program G in Listing 1.6 is an example for an Xcerpt program which is not grouping stratifiable. We show in this chapter that G is locally grouping stratifiable. Thus the set of all grouping stratifiable Xcerpt programs is a real subset of the set of all locally grouping startifiable Xcerpt programs. See Figure 6.1.

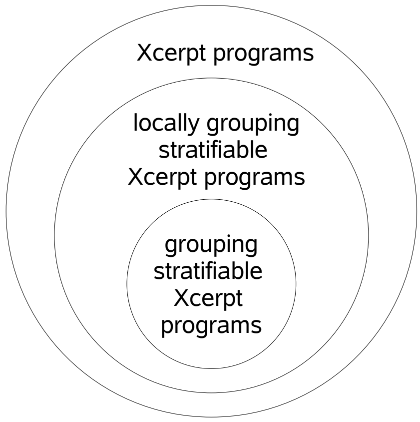

Figure 6.1: Grouping Stratification vs. Local Grouping Stratification

Consider the Xcerpt program G in Listing 1.6. G computes all possible paths from the node `anna` to each other node in the directed graph in Figure 1.1.

G is locally grouping stratifiable, because there is a valid local grouping stratification for G. It is infinite.

$\mathcal{HB}$ contains also some 'illegal' terms, for instance `Ac[anna,anna]` is illegal, because the second term must be have the outermost functional symbol *path*. *anna* is also an illegal term, bacause it appears only within some functional symbol (*Ac*, *path* or *knows*) in G_H. Illegal terms are always false in the intuitive meaning of a progam.

Each stratum contains both legal and illegal terms. The number of the legal terms in each stratum is finite. The number of the illegal terms in each stratum is infinite. That is why each stratum is infinite.

Condition 1: Because of the second rule of G the terms with the outermost functional symbol *Ac* must be in a lower stratum as the terms with the same outermost functional symbol if the arity of the functional symbol *path* in the second term of the former terms is smaller as the arity of the same functional symbol in the second term of the latter terms.
Condition 2: Because of the second rule of G the terms with the outermost functional symbol *knows* must be in a lower stratum as the terms with the outermost functional symbol *Ac*.
G_1 contains some other illegal terms which are not restricted by the conditions.

...

G_3:
Legal terms:
Ac[anna,path[anna,anna]], Ac[anna,path[bob,anna]], Ac[anna,path[anna,bob]], Ac[anna,path[bob,bob]], Ac[anna,path[anna,chuck]], Ac[anna,path[chuck,anna]], Ac[anna,path[chuck,chuck]], Ac[anna,path[chuck,bob]], Ac[anna,path[bob,chuck]], Ac[bob,path[anna,anna]], Ac[bob,path[bob,anna]], Ac[bob,path[anna,bob]], Ac[bob,path[bob,bob]], Ac[bob,path[anna,chuck]], Ac[bob,path[chuck,anna]], Ac[bob,path[chuck,chuck]], Ac[bob,path[chuck,bob]], Ac[bob,path[bob,chuck]], Ac[chuck,path[anna,anna]], Ac[chuck,path[bob,anna]], Ac[chuck,path[anna,bob]], Ac[chuck,path[bob,bob]], Ac[chuck,path[anna,chuck]], Ac[chuck,path[chuck,anna]], Ac[chuck,path[chuck,chuck]], Ac[chuck,path[chuck,bob]], Ac[chuck,path[bob,chuck]]
Illegal terms:
Ac[anna,path[path[anna],anna]], ..., Ac[anna,path[path[anna,anna],anna]], ...

G_2:
Legal terms:
Ac[anna,path[anna]], Ac[anna,path[bob]], Ac[anna,path[chuck]],
Ac[bob,path[anna]], Ac[bob,path[bob]], Ac[bob,path[chuck]],
Ac[chuck,path[anna]], Ac[chuck,path[bob]], Ac[chuck,path[chuck]]
Illegal terms:
Ac[anna,path[path[anna]]], ..., Ac[path[anna,anna],path[anna]], ...

G_1:
Legal terms:
Ac[anna,path[]], Ac[bob,path[]], Ac[chuck,path[]],
knows[anna,anna], knows[anna,bob], knows[anna,chuck],
knows[bob,bob], knows[bob,anna], knows[bob,chuck],
knows[chuck,chuck], knows[chuck,anna], knows[chuck,bob]
Illegal terms:
Ac[anna,anna], ..., Ac[path[anna],path[]], ...
knows[anna,path[bob]], ..., knows[path[anna],path[bob]], ...,

anna, ...,
path[], path[anna], ..., path[path[anna]], ..., path[Ac[anna,path[anna]],bob], ...

As we have already shown at page 10, G has an intuitive meaning. It computes all possible paths from the node `anna` to each other node in the directed graph in Figure 1.1.

Chapter 7

The Totality of the Well-founded Model of a Locally Stratifiable Xcerpt Program

In this chapter we prove that every locally stratifiable Xcerpt program has a total well-founded model and show that the inversion of this statement does not hold.

Theorem 1. *Every locally stratifiable Xcerpt program has a total well-founded model*

Proof through contradiction. Let P be an Xcerpt program, P_H its Herbrand instantiation, $\mathcal{HB}$ its associated Herbrand base and $D \subseteq \mathcal{HB}$ the set of all data terms of $\mathcal{HB}$.

Assumptions:

1. Let P be locally stratifiable.
2. Let $\mathrm{lfp}(\mathbf{W}_P)$ be a partial well-founded model of $P \Rightarrow$
   $\exists \mathtt{h} \in D.\ \mathtt{h} \notin \mathrm{pos}(\mathrm{lfp}(\mathbf{W}_P)) \cup \mathrm{neg}(\mathrm{lfp}(\mathbf{W}_P))$, that means $\mathtt{h}$ is undefined in $\mathrm{lfp}(\mathbf{W}_P) \Rightarrow$
   $\mathrm{pos}(\mathrm{lfp}(\mathbf{W}_P)) \cup \mathrm{neg}(\mathrm{lfp}(\mathbf{W}_P)) \subset D$.

Then there is at least one ground rule instance with head $\mathtt{h}$ in P_H
(otherwise $\mathtt{h} \in \mathrm{neg}(\mathrm{lfp}(\mathbf{W}_P))$)

Let $\mathrm{R}_1 = \mathtt{h} \leftarrow \mathrm{Q}_1$, ..., $\mathrm{R}_n = \mathtt{h} \leftarrow \mathrm{Q}_n$, $\mathrm{n} \in \mathbb{N}$, be the rule instances with head h in P_H
$\forall \mathtt{i}$, $1 \leq \mathrm{i} \leq \mathrm{n}$, Q_i is undefined in $\mathrm{lfp}(\mathbf{W}_P)$
(otherwise $\mathtt{h} \in \mathrm{pos}(\mathrm{lfp}(\mathbf{W}_P)) \cup \mathrm{neg}(\mathrm{lfp}(\mathbf{W}_P))$) $\Rightarrow$

$\exists \mathtt{q} \in \mathrm{Q}_i.\ \mathtt{q}$ is undefined in $\mathrm{lfp}(\mathbf{W}_P) \Rightarrow$

`q` depends negatively on itself, on `h` or it depends positively or negatively on another undefined atom $\Rightarrow$

There is a negative cycle in the dependency graph of $P \Rightarrow$

P is not locally stratifiable $\Rightarrow$

Contradiction to assumption 1.

Conclusion:
lfp($\mathbf{W}_P$) is a total well-founded model of $P \Rightarrow$
$\forall$h$\in D$. h$\in$pos(lfp($\mathbf{W}_P$))$\cup$neg(lfp($\mathbf{W}_P$)) $\Rightarrow$
pos(lfp($\mathbf{W}_P$))$\cup$neg(lfp($\mathbf{W}_P$))$=D$. $\square$

Let Q be a query containing only the function symbols of the signature $\mathcal{L}$ of an Xcerpt program P.

P returns 'true' for the query Q iff there is a ground query Q_{ground} such that $Q \preceq \mathrm{Q}_{ground}$ and lfp($\mathbf{W}_P$)$\models \mathrm{Q}_{ground}$ (see Definition 24).

P returns 'false' for the query Q iff for all ground queries Q_{ground} holds $Q \preceq \mathrm{Q}_{ground}$ and lfp($\mathbf{W}_P$)$\not\models \mathrm{Q}_{ground}$.

We have shown above that for a locally stratifiable Xcerpt program pos(lfp($\mathbf{W}_P$))$\cup$neg(lfp($\mathbf{W}_P$))$=D$ holds. With other words there is no Q_{ground} such that Q_{ground} is undefined in lfp($\mathbf{W}_P$) because each data term Q_{ground} depends on, is defined in lfp($\mathbf{W}_P$). That means that each query Q can be answered after the evaluation of a locally stratifiable Xcerpt program. That is the practical meaning of Theorem 1.

The inversion of Theorem 1 does not hold. There are Xcerpt programs which have a total well-founded model but are locally unstratifiable. Consider for example the Xcerpt program A in Listing 5.2.

Chapter 8

Conclusion

Results of this work:

1. Notions for Xcerpt, such as Xcerpt signature, Herbrand universe, model relationship, dependency graph and so on
2. Well-founded semantics for Xcerpt programs without grouping constructs
3. Local stratification for Xcerpt programs
4. Proof that every locally stratifiable Xcerpt program has a total well-founded model
5. Local grouping stratification for Xcerpt programs
6. SLG-Algorithm for Xcerpt programs

Open questions and future work:

1. Implementation of the SLG-Algorithm in Haskell
2. Proof that the SLG-Algorithm returns a complete answer for each query
3. Is there a syntactic condition for Xcerpt programs which is broader as local stratification, such that the following holds:
   An Xcerpt program satisfies this condition iff it has a total well-founded model
4. Dependency graph for Xcerpt programs with grouping constructs
5. Complexity of locally stratifiable and locally grouping stratifiable Xcerpt programs

Appendix A

SLG-Algorithm

In this chapter we present the code of the algorithm computing the SLG-Resolution for Xcerpt programs. The informal description of the algorithm in section 4.2 and some examples of its evaluation in Section 4.3 help to understand the following detailed description of the algorithm. In Section A.3 time and space complexity of the algorithm are analysed.

A.1 Data Structures and Auxilary Funktions of the Algorithm

The algorithm in Listing A.1 uses the data structures and auxiliary functions described in Tables A.1 and A.2 respectively.

Data Structures	Discription
list_of_nodes[ ]	All the nodes of the SLG-Resolution trees with pointer to their parents are saved in the list_of_nodes[ ]. At the beginning the list_of_nodes[ ] consists of the root of the tree only. The root of the tree is the user's query. (see lines 2,3).
table[ ]	All solution goals and their solutions are saved in the table. In line 4 of the algorithm the table is initialized. See section 4.2 for the detailed description of the data structure.
children(goal)[ ]	is a list containing all the children of the goal. The children of a solution goal are the bodies of all rule instances, the head of which unifies with this solution goal (see lines 41,53). The children of a look-up goal are all solutions for the respective solution goal (see lines 18 and 30). This function is used in order to build new children of the node the goal appears withing. For each child of the goal a new child of the node is built.
variable_bindings (goal)[ ]	is a list containing all the variable bindings yield by the simulation unification of the goal into a solution (if the goal is a look-up goal) or the head of a rule instance (if the goal is a solution goal)(lines 51,61). This function is used in order to propagate a solution for a solution goal.

Table A.1: Data structures of the algorithm in Listing A.1

Auxiliary Functions	Discription
isMain_query(goal)	returns true if the goal is the user's query. It returns false otherwise. There is only one goal in all SLG Resolution trees with isMain_query(goal):=true (line 1 of the algorithm). This function is used to avoid needless infinite loops (lines 146-157).
isLook-upGoal(goal)	returns true if the goal is a look-up goal, it returns false otherwise
isSolutionGoal(goal)	returns true if the goal is a solution goal, it returns false otherwise
list(goal)	returns the list the goal appears within
node(goal)	returns the node the goal appears within
pointer(goal)	In order to distinguish between the children of a goal, which have already been processed, and other children a pointer is saved for each goal. The pointer points to the last finished child. If a goal is new, its pointer is -1 (see line 64). Every time a look-up goal uses a solution or a solution goal uses a rule instance, its pointer increases (see lines 35,43,106 and 138). The pointer is used as an index in the list children(goal)[ ].
isNegated(goal)	returns true if the goal is negative. It returns false otherwise. See lines 17, 65.
not(goal)	returns the negation of the goal. The algorithm uses this function in order to build an auxiliary SLG resolution tree. See lines 65-70
solution_goal(look-upGoal)	returns the solution goal the look-upGoal is associated with
most_special(look-upGoal1, look-upGoal2)	returns the most special look-up goal of look-upGoal1 and look-upGoal2. If there are many suspended nodes in the list_of_nodes[ ] beginning with a look-up goal waiting for the same solution goal, the node beginning with the most special look-up goal is evaluated first. See Section 4.2 for delails and examples.
isCompleted(solutionGoal)	returns true if the solutionGoal is completed, that is there is no more solution for it (see line 110). It returns false otherwise.

solution_goal(list[ ])	returns the solution goal which depends on the goals saved in the list. If the list is empty a solution for the solution goal is found.
delete(list[ ])	deletes the list
parent(list[ ])	returns the list the list appears within. If the list is empty it is deleted. But probably the list is the only list of the parent(list[ ]). Threrefore parent(list[ ]) must be also deleted.
parent(node)	returns the node which is parent of the node. We need this link to navigate in the tree.
first_goal(node)	returns the first goal in the first list of the node
isSuspended(node)	returns true if the node is suspended, that is the node begins with a look-up goal waiting for a solution goal which is not completed yet (see lines 21,23 and 27-29). It returns false otherwise.
put(goal, table[ ])	puts the goal in the table[ ]
put(node, list_of_nodes[ ])	puts the node in the list_of_nodes[ ]
update(solutionGoal,solution, table[ ])	puts the solution for this solutionGoal in the table
put(variable_bindings(goal1)[ ], variable_bindings(goal2)[ ])	adds the list of variable bindings of the goal1 to the list of variable bindings of the goal2. If there are variables with the same name variable_bindings(goal2)[ ] win. The algorithm uses this function in order to propagate variable bindings.
delete(list_of_nodes[ ])	deletes the list_of_nodes[ ]
delete(table[ ])	deletes the table[ ]

Table A.2: Auxiliary functions of the algorithm in Listing A.1

A.2 The Code of the Algorithm

Listing A.1: The SLG-Algorithm

```
1    isMain_query(user's_query):=true
2    node:=user's_query
3  list_of_nodes[]:=[node]
```

```
table:=[]
n:=0
compute_solutions(node){
  list(user's_query)[]:=[]
  parent(node):=nil
  slg_resolution(node)
  delete(list_of_nodes[])
  print all solutions for the first goal in the table[]
  delete(table[])
}
slg_resolution(node) {
  goal:=first_goal(node)
  if isLook-upGoal(goal)
  then if isNegated(goal)
       then children(goal)[]:=solutions of the solution_goal(not(goal))
            if children(goal)[]!=[]
            then back(parent(node))
            else if isCompleted(solution_goal(not(goal)))
                 then new_child(node,goal,true)
                 else isSuspended(node):=true
                      back(parent(node))
                 end if
            end if
       else if !isCompleted(solution_goal(goal))
            then isSuspended(node):=true
            end if
            children(goal)[]:=solutions of the solution_goal(goal)
            if children(goal)[]=[]
            then back(parent(node))
            else if children(goal)[pointer(goal)+1]=nil
                 then back(parent(node))
                 else pointer(goal):=pointer(goal)+1
                      solution:=children(goal)[pointer(goal)]
                      new_child(node,goal,solution)
                 end if
            end if
  else put(goal,table[])
```

```
        children(goal)[]:= rule instances producing the goal
        if children(goal)[pointer(goal)+1]!= nil
        then pointer(goal):= pointer(goal)+1
             rule_instance:= children(goal)[pointer(goal)]
             new_child(node, goal, rule_instance)
        else back(parent(node))
        end if
  end if
}
new_child(node, goal, rule_instance) {
  variable_bindings(goal)[]:= goal<=the head of the rule_instance
  if isSolutionGoal(goal)
  then list_n[]:= all terms of the body of the rule_instance
        parent(list_n[]):= list(goal)
        child(node):= the node with the list_n[] instead of the goal
        delete_empty_lists(list_n[], child(node))
        n:=n+1
  else child(node):= the node without the goal
        delete_empty_lists(list(goal), child(node))
  end if
  propagate the variable_bindings(goal)[] in the child(node)
  parent(child(node)):= node
  goal_of_child:= first_goal(child(node))
  pointer(goal_of_child):= -1
  if isNegated(goal_of_child) and isSolutionGoal(not(goal_of_child))
  then list(not(goal_of_child)):= []
        node:= not(goal_of_child)
        parent(node):= nil
        list_of_nodes[]:= [node]
        slg_resolution(node)
  end if
  if child(node)= nil
  then delete_empty_lists(list(goal), node)
        back(node)
  else if the child(node) and the table[] are consistent
        then put(child(node), list_of_nodes[])
             slg_resolution(child(node))
```

```
            else back(node)
            end if
    end if
}
delete_empty_lists(list_n[],node){
    if list_n[]=[]
    then solution_goal:=solution_goal(list_n[])
            propagate_solution(solution_goal,node)
            delete(list_n[])
            delete_empty_lists(parent(list_n[]),node)
    end if
}
propagate_solution(solution_goal,node){
    goal:=first_goal(node)
    if solution_goal=goal
    then if variable_bindings(goal)[]=[]
            then update(goal,true,table[])
            else update(goal,variable_bindings(goal)[],table[])
            end if
    else parent_goal:=first_goal(parent(node))
            put(variable_bindings(goal)[],variable_bindings(parent_goal)[])
            propagate_solution(solution_goal,parent(node))
    end if
}
back(node) {
    if node!=nil
    then goal:=first_goal(node)
            if children(goal)[pointer(goal)+1]!=nil
            then pointer(goal):=pointer(goal)+1
                    rule_instance:=children(goal)[pointer(goal)]
                    new_child(node,goal,rule_instance)
            else if isSolutionGoal(goal) and !isCompleted(goal)
                    then isCompleted(goal):=true
                    end if
                    back(parent(node))
            end if
    else for each node1 in the list_of_nodes[]
```

```
            if isSuspended(node1) and
               isCompleted(solution_goal(first_goal(node1)))
            then for each node2 in the list_of_nodes[]
                     if isSuspended(node2) and node1!=node2 and
                        solution_goal(first_goal(node1)))=
                        solution_goal(first_goal(node2)))
                     then look-up_goal:=
                          most_special(first_goal(node1),first_goal(node2))
                          node1:=node(look-up_goal)
                     end if
                 end for each
                 if the node1 and the table[] are consistent
                 then slg_resolution(node1)
                 else back(parent(node1))
                 end if
            end if
        end for each
        for each solution_goal in the table[]
            for each node in the list_of_nodes[]
                if isLook-upGoal(first_goal(node)) and
                   solution_goal=solution_goal(first_goal(node)) and
                   length(children(first_goal(node)))>=
                   pointer(first_goal(node))
                then pointer(first_goal(node)):=pointer(first_goal(node))+1
                     solution:=
                     children(first_goal(node))[pointer(first_goal(node))]
                     new_child(node,first_goal(node),solution)
                end if
            end for each
        end for each
        goal:=first_goal(child(node))
        if !isMain_query(goal)
        then for each node in the list_of_nodes[]
                 if not(goal)=firts_goal(node)
                 then if children(not(goal))=[] and there is at least one
                         suspended node in the tree with root goal
                      then isSuspended(node(not(goal))):=true
```

```
                              back(parent(node(not(goal))))
                        else slg_resolution(node(not(goal)))
                        end if
                  end if
             end for each
       end if
 end if
}
```

A.3 Time and Space Complexity of the Algorithm

Let `s` be the number of rule instances of an Xcerpt Program ⇒ `s` is the greatest number of solution goals which will be produced by the program and therefore must be saved. Let `l` be the greatest number of look-up goals.

Time Complexity of the Algorithm

The algorithm decides for each node, whether it begins with a solution or with a look-up goal. That is why it searches the first goal of each node in the the table.
Each goal of each node must be resolved.

$T = O(\#nodes * length(table[\ ]) * \#goals) = O((s+l) * s * (s+l)) =$
$O(s^3 + s^2 * l + s * l^2)$

Space Complexity of the Algorithm

$S = O(length(list\_of\_nodes[\ ]) + length(table[\ ])) =$
O(#nodes*(#goals+#children of the first goal+parent of the node+pointer to the last used solution for a look-up goal or the number of the list of goals a solution goal depends on) + #solution goals*(1+#solutions+1)) =
$O(\ (s+l)*((s+l)+s+1+1) + s*(1+s+1)\ ) = O(s^2 + s*l + l^2)$

Space complexity of the table is #solution goals*(1+#solutions+1), bacause a triple is saved for each solution goal. Each triple consists of a solution goal, its solutions and 'true' if the solution goal is completed or 'false' otherwise. See Subsection 4.2.3 for details.

Bibliography

[1] F.Bry, N.Eisinger, T.Eiter, T.Furche, G.Gottlob, C.Ley, B.Linse, R.Pichler, F.Wei: Foundations of Rule-Based Query Answering

[2] F.Bry, S.Schaffert: XML and Databases

[3] A.V.Gelder, K.A.Ross, J.S.Schlipf: The Well-Founded Semantics for General Logic Programs

[4] W.Chen, D.S.Warren: Tabled Evaluation with Delaying for General Logic Progams

[5] S.Schaffert: Xcerpt: A Rule-Based Query and Transformation Language for the Web

[6] François Bry and Tim Furche and Benedikt Linse: Simulation Subsumption or Déjà vu on the Web