REWERSE©
reasoning on the web

I4-D4

Initial Draft of a Possible Declarative Semantics for the Language

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D4/D/PU/a1
Responsible editors:	Sebastian Schaffert
Reviewers:	Claude Kirchner and Wolfgang May
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	28 February 2005
Actual submission date:	15 April 2005

Abstract

This article introduces a preliminary declarative semantics for a subset of the language Xcerpt (so-called *grouping-stratifiable programs*) in form of a classical (Tarski style) model theory, adapted to the specific requirements of Xcerpt's constructs (e.g. the various aspects of incompleteness in query terms, grouping constructs in rule heads, etc.). Most importantly, the model theory uses *term simulation* as a replacement for term equality to handle incomplete term specifications, and an extended notion of substitutions in order to properly convey the semantics of grouping constructs. Based upon this model theory, a fixpoint semantics is also described, leading to a first notion of forward chaining evaluation of Xcerpt programs.

Keyword List

reasoning, query language, Semantic Web, model theory, semantics, declarative semantics

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

Initial Draft of a Possible Declarative Semantics for the Language

Sebastian Schaffert, François Bry, Tim Furche

Institute for Informatics, University of Munich, Germany
Email: {Sebastian.Schaffert,Francois.Bry,Tim.Furche}@ifi.lmu.de

15 April 2005

Abstract
This article introduces a preliminary declarative semantics for a subset of the language Xcerpt (so-called *grouping-stratifiable programs*) in form of a classical (Tarski style) model theory, adapted to the specific requirements of Xcerpt's constructs (e.g. the various aspects of incompleteness in query terms, grouping constructs in rule heads, etc.). Most importantly, the model theory uses *term simulation* as a replacement for term equality to handle incomplete term specifications, and an extended notion of substitutions in order to properly convey the semantics of grouping constructs. Based upon this model theory, a fixpoint semantics is also described, leading to a first notion of forward chaining evaluation of Xcerpt programs.

Keyword List
reasoning, query language, Semantic Web, model theory, semantics, declarative semantics

Contents

1 Introduction

This article introduces a declarative semantics for a restricted form of Xcerpt programs (so-called *grouping-stratifiable programs* without negation). Although a short introduction to Xcerpt is given in Section 2, this article does not cover the language in much detail; interested readers can find a more thorough description of Xcerpt in e.g. [SB04] and [Sch04]. The aim of the declarative semantics introduced here is to describe the semantics of Xcerpt programs in a precise and formal, yet intuitive and straightforward, manner without referring to a concrete implementation of the language. This description should serve as a reference for verifying the correctness and completeness of language implementations and as a formal specification for users seeking to get a precise understanding of the language.

The declarative semantics is given as a model theory in the style of *Tarski* (i.e. recursively defined over the formula structure). It follows the semantics for first order logic rather closely but needs to take into account the particularities of Xcerpt terms and programs (e.g. the various aspects of incompleteness in query terms, grouping constructs in rule heads, etc.). Intuitively, the definition of interpretations and models is straightforward: an interpretation is a set of data terms and specifies what data terms exist; a model is then simply an interpretation that consists of the terms that are “produced” by the rules in a program.

Section 2 briefly recapitulates the language Xcerpt and introduces several formalisms and denotations used in the remainder of this article. Section 3 introduces so-called *term formulas* that can be composed of Xcerpt terms and logical connectives like $\wedge$ or $\vee$. Term formulas depart form first order logic in that they do not distinguish between predicate and term symbols, because the Web consists of “data”, not “statements”. Next, a notion of *substitution sets* is described in Section 4. Substitution sets take the role of substitutions in first order logic and logic programming and are required to properly convey the meaning of Xcerpt’s grouping constructs `all` and `some`. Section 5 defines *ground query term simulation* as a relation between terms that properly conveys the meaning of incomplete term specifications (e.g. unordered or partial). This definition is further used in Section 6, where interpretations and the satisfaction of term formulas is defined. In Section 7, a fixpoint semantics for stratifiable Xcerpt programs is suggested, first for programs without negation, and then for arbitrary Xcerpt programs. Finally, Section 8 contains some concluding remarks and perspectives for further refinement of the semantics. Note that this article mostly follows the semantics described in [Sch04].

2 Preliminaries

2.1 Xcerpt: A versatile Web Query Language

An Xcerpt [SB04, Sch04] program consists of at least one *goal* and some (possibly zero) *rules*. Rules and goals contain query and construction patterns, called *terms*. Terms represent tree-like (or graph-like) structures. The children of a node may either be *ordered*, i.e. the order of occurrence is relevant (e.g. in an XML document representing a book), or *unordered*, i.e. the order of occurrence is irrelevant and may be chosen by the storage system (as is common in database systems). In the term syntax, an *ordered term specification* is denoted by square brackets `[ ]`, an *unordered term specification* by curly braces `{ }`.

Likewise, terms may use *partial term specifications* for representing incomplete query patterns and *total term specifications* for representing complete query patterns (or data items). A term t using a partial term specification for its subterms matches with all such terms that (1) contain matching subterms for all subterms of t and that (2) might contain further subterms without corresponding subterms in t. Partial term specification is denoted by *double* square brackets `[[ ]]` or curly braces `{{ }}`. In contrast, a term *t* using a total term specification does not match with terms that contain additional subterms without corresponding subterms in *t*. Total term specification is expressed using *single* square brackets `[ ]` or curly braces `{ }`. Matching is formally defined later in this article using so-called *term simulation*.

Furthermore, terms may contain the *reference constructs* `^id` (*referring occurrence of the identifier* `id`) and `id @ t` (*defining occurrence of the identifier* `id`). Using reference constructs, terms can form cyclic (but rooted) graph structures.

2.1.1 Data Terms

Data terms represent XML documents and the data items of a semistructured database, and may thus only contain total term specifications (i.e. single square brackets or curly braces). They are similar to *ground* functional programming expressions and logical atoms. A *database* is a (multi-)set of data terms (e.g. the Web). A non-XML syntax has been chosen for Xcerpt to improve readability, but there is a one-to-one correspondence between an XML document and a data term. Example 1 on the facing page gives an impression of the Xcerpt term syntax.

2.1.2 Query Terms

Query terms are (possibly incomplete) patterns matched against Web resources represented by data terms. They are similar to the latter, but may contain *partial* as well as *total* term specifications, are augmented by *variables* for selecting data items, possibly with *variable restrictions* using the → construct (read *as*), which restricts the admissible bindings to those subterms that are matched by the restriction pattern, and may contain additional query constructs like *position matching* (keyword `position`), *subterm negation* (keyword `without`), *optional subterm specification* (keyword `optional`), and *descendant* (keyword `desc`).

Query terms are "matched" with data or construct terms by a non-standard unification method called *simulation unification* that is based on a relation called *simulation* (cf. Section 5). In contrast to Robinson's unification (as e.g. used in Prolog), simulation unification is capable of determining substitutions also for incomplete and unordered query terms. Since incompleteness usually allows many different alternative bindings for the variables, the result of simulation unification is not only a single substitution, but a (finite) *set of substitutions*, each of which yielding ground instances of the unified terms such that the one ground term matches with the other. Whenever a term t_1 simulates into another term t_2, this shall be denoted by $t_1 \preceq t_2$.

2.1.3 Construct Terms

Construct terms serve to reassemble variables (the bindings of which are specified in query terms) so as to construct new data terms. Again, they are similar to the latter, but augmented by *variables* (acting as place holders for data selected in a query) and the *grouping construct* `all` (which serves to collect all instances that result from different variable bindings). Occurrences of `all` may be accompanied by an optional sorting specification.

Example 2
Left: A query term retrieving departure and arrival stations for a train in the train document. Partial term specifications (partial curly braces) are used since the train document might contain additional information irrelevant to the query. *Right:* A construct term creating a summarised representation of trains grouped inside a `trains` term. Note the use of the `all` construct to collect all instances of the

Example 1
The following two data terms represent a train timetable (from `http://railways.com`) and a hotel reservation offer (from `http://hotels.net`).

At site `http://railways.com`:

```
travel {
  last-changes-on { "2004-04-30" },
  currency { "EUR" },
  train {
    departure {
      station { "Munich" },
      date { "2004-05-03" },
      time { "15:25" }
    },
    arrival {
      station { "Vienna" },
      date { "2004-05-03" },
      time { "19:50" }
    },
    price { "75" }
  },
  train {
    departure {
      station { "Munich" },
      date { "2004-05-03" },
      time { "13:20" }
    },
    arrival {
      station { "Salzburg" },
      date { "2004-05-03" },
      time { "14:50" }
    },
    price { "25" }
  },
  train {
    departure {
      station { "Salzburg" },
      date { "2004-05-03" },
      time { "15:20" }
    },
    arrival {
      station { "Vienna" },
      date { "2004-05-03" },
      time { "18:10" }
    }
  }
  ...
}
```

At site `http://hotels.net`:

```
voyage {
  currency { "EUR" },
  hotels {
    city { "Vienna" },
    country { "Austria" },
    hotel {
      name { "Comfort␣Blautal" },
      category { "3␣stars" },
      price-per-room { "55" },
      phone { "+43␣1␣88␣8219␣213" },
      no-pets {}
     },
    hotel {
      name { "InterCity" },
      category { "3␣stars" },
      price-per-room { "57" },
      phone { "+43␣1␣82␣8156␣135" }
     },
    hotel {
      name { "Opera" },
      category { "4␣stars" },
      price-per-room { "106" },
      phone { "+43␣1␣77␣8123␣414" }
     },
  ...
  },
...
}
```

train subterm that can be created from substitutions in the substitution set resulting from the query on the left.

```
travel {{                            trains {
  train {{                             all train {
    departure {{                         from { var From },
      station { var From } }},           to   { var To }
    arrival {{                         }
      station { var To }   }}        }
  }}
}}
```

2.1.4 Construct-Query Rules

Construct-query rules (short: rules) relate a construct term to a query consisting of AND and/or OR connected query terms. They have the form

```
CONSTRUCT Construct Term FROM Query END
```

Rules can be seen as "views" specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g. an XML document or a database). Queries or parts of a query may be further restricted by arithmetic constraints in a so-called condition box, beginning with the keyword where.

Example 3
The following Xcerpt rule is used to gather information about the hotels in Vienna where a single room costs less than 70 Euro per night and where pets are allowed (specified using the without construct).

```
CONSTRUCT
  answer [ all var H ordered by [ P ] ascending ]
FROM
  in {
    resource { "http://hotels.net" },
    voyage {{
      hotels {{
        city { "Vienna" },
        desc var H    hotel {{
          price-per-room { var P },
          without no-pets {}
        }}
      }}
    }}
   } where var P < 70
END
```

An Xcerpt query may contain one or several references to *resources*. Xcerpt rules may furthermore be *chained* like active or deductive database rules to form complex query programs, i.e. rules may query the results of other rules. Recursive chaining of rules is possible (but note that the declarative semantics described here requires certain restrictions on recursion, cf. Section 2.2). In contrast to the inherent structural recursion used e.g. in XSLT, which is essentially limited to the tree structure of the input document, recursion in Xcerpt is always explicit and free in the sense that any kind of recursion can be implemented. Applications of recursion on the Web are manifold:

- structural recursion over the input tree (like in XSLT) is necessary to perform transformations that preserve the overall document structure and change only certain things in arbitrary documents (e.g. replacing all `em` elements in HTML documents by `strong` elements).
- recursion over the conceptual structure of the input data (e.g. over a sequence of elements) is used to iteratively compute data (e.g. create a hierarchical representation from flat structures with references).
- recursion over references to external resources (hyperlinks) is desirable in applications like Web crawlers that recursively visit Web pages.

Example 4
The following scenario illustrates the usage of a “conceptual” recursion to find train connections, including train changes, from Munich to Vienna.

The `train` relation (more precisely the XML element representing this relation) is defined as a “view” on the train database (more precisely on the XML document seen as a database on trains):

```
CONSTRUCT
  train [ from [ var From ], to [ var To ] ]
FROM
  in {
    resource { "file:travel.xml" },
    travel {{
      train {{
        departure {{ station { var From } }},
        arrival   {{ station { var To }   }}
      }}
    }}
  }
END
```

A recursive rule implements the transitive closure `train-connection` of the relation `train`. If the connection is not direct (recursive case), then all intermediate stations are collected in the subterm `via` of the result. Otherwise, `via` is empty (base case).

```
CONSTRUCT
  train-connection [
    from [ var From ],
    to   [ var To ],
    via  [ var Via, all optional var OtherVia ]
  ]
FROM
  and {
    train [ from [ var From ], to [ var Via ] ],
    train-connection [
      from [ var Via ],
      to   [ var To  ],
      via  [[ optional var OtherVia ]]
    ]
  }
END

CONSTRUCT
  train-connection [
    from [ var From ],
    to   [ var To ],
    via  [ ]
  ]
FROM
  train [ from [ var From ], to [ var To ] ]
END
```

Based on the "generic" transitive closure defined above, the following rule retrieves only connections between Munich and Vienna.

```
GOAL
  connections {
    all var Conn
  }
FROM
  var Conn    train-connection [[ from { "Munich" } , to { "Vienna" } ]]
END
```

2.2 Range Restrictedness and Stratification

The declarative semantics described in this article assumes certain restrictions on Xcerpt programs: *range restrictedness*, *negation stratification*, and *grouping stratification*. Range restrictedness restricts the occurrences of variables in rules and grouping and negation stratification restricts the way recursion is used in Xcerpt programs. Note that for all three kinds of restrictions, there exist examples where a relaxation might be desirable.

2.2.1 Range Restrictedness

Range restrictedness (often referred to as *safe-ness*) means that a variable occurring in a rule head also must occur at least once in every disjunctive part in the rule body. This requirement simplifies the definition of the declarative semantics of Xcerpt, as it allows to assume that all query terms are unified with data terms instead of construct terms (i.e. variable-free and grouping-free terms). Without this restriction, it is necessary to consider undefined or infinite sets of variable bindings, which would be a difficult obstacle for a forward chaining evaluation. Besides this technical reason, range restricted programs are also usually more intuitive, as they disallow variables in the head that are not justified somewhere in the body.

Range restrictedness can be verified by assigning "polarities" to every term and all its subterms in a rule such that all terms in the query part initially have negative polarity while the construct term initially has positive polarity (cf. [Sch04]). A variable occurrence with *positive* polarity represents a *consuming* occurrence of that variable, a variable occurrence with *negative* polarity represents a *defining* occurrence of that variable. Polarities may switch if the query contains negation constructs like `not` or `without`.

Range restrictedness requires that every variable occurring positively (i.e. as a consuming occurrence) also must occur negatively (i.e. as a defining occurrence) in each disjunctive part of a rule.

Example 5
Consider the following Xcerpt program:

```
CONSTRUCT
  f{var X, var Y}
FROM
  or {
    g{var X, var Y, var Z},
    and {
      h{var X, var Y},
      not k{var X, var Z}
    }
  }
END
```

Because of the or-construct in the rule body, this rule contains two disjuncts. In the first disjunct, the variables X, Y, and Z occur with negative polarity (because they are part of the query), and the variables X and Y also occur with positive polarity (because they occur in the rule head). This part of the rule would thus be range restricted. However, in the second disjunct, only the variables X and Y occur positively, while X, Y, and Z occur negatively (note that Z is contained within a not-negation). Thus, this part is not range restricted.

2.2.2 Stratification

Stratification is a technique to define a class of logic programs where non-monotonic features like Xcerpt's grouping constructs or negation can be defined in a declarative manner. The principal idea of stratification is to disallow programs with a recursion over negated queries ("*negation stratification*") or grouping constructs ("*grouping stratification*") and thereby preclude undesirable programs that have a non-intuitive semantics. While this requirement is very strict, its advantages are that it is straightforward to understand and can be verified by purely syntactical means without considering terms that are not part of the program (as is required by more elaborate techniques like *stable models*).

Several refinements over stratification have been proposed, e.g. *local stratification* [Prz88] that allow certain kinds of recursion, but these usually require more "knowledge" of the program or the queried resources. This section only gives an intuition over grouping and negation stratification; stratification of Xcerpt programs is described in detail in [Sch04].

Grouping Stratification The grouping constructs all and some are powerful constructs that are justified by many practical applications. However, using them in recursive rules allows to define programs with no useful meaning. Consider for example the program

$$\begin{array}{l} f\{all\ var\ X\} \leftarrow f\{\{var\ X\}\} \\ f\{a\} \end{array}$$

The meaning of such programs is unclear and probably unintended by the program author. The solution is to disallow recursion of rules with grouping constructs, and to require that all rules on which a rule with grouping constructs depends can be evaluated first. Programs that fulfill this propertiy are called *grouping stratifiable*.

Negation Stratification Xcerpt's `not`-construct is evaluated as *negation as failure (NaF)*, i.e. a negated query succeeds if the query itself fails finitely (i.e. can be proven to be not provable). NaF is desirable for a Web query language, because it is close to the intuitive understanding of negation: for instance, it is natural to assume that a train not listed in a train timetable does not exist, instead of requiring that every non-existent train is explicitly listed in the timetable.

Although NaF has a purely operational meaning, it is desirable to provide a declarative semantics as well, because the latter is usually easier to understand than the evaluation algorithm. Unfortunately, like recursion over grouping constructs, negation as failure allows for programs whose meaning is unclear. Consider for instance the following Xcerpt program:

$$f\{a\} \leftarrow \mathit{not}\ \ f\{a\}$$

Backward chaining evaluation of this rule does not terminate: for proving $f\{a\}$, it is necessary to show (in an auxiliary computation) that $f\{a\}$ does not hold, which again requires to evaluate the rule, and so on.

Declaratively, the meaning of this rule is problematic. When representing rules by implication as in traditional logic programming, this rule is simply equivalent to $f\{a\} \vee \neg\neg f\{a\}$, which simplifies to $f\{a\}$. This interpretation does not reflect the operational behaviour (which is the definition for negation as failure) described in the previous paragraph. Other approaches have been considered (like Clarke's completion or default negation) that interpret the symbol $\leftarrow$ differently, but all of these have similar problems.

Xcerpt programs are therefore assumed to be also *negation stratifiable*, a syntactic restriction that excludes such programs that involve problematic use of negation as in the example above. Negation stratification in Xcerpt programs is defined in the usual manner (as e.g. in [ABW88]). In stratifiable programs, both recursion and negation are allowed, but a recursion "through negation" is disallowed.

2.3 Ground Query Terms and Ground Query Term Graphs

Let T^q be the set of all query terms.

Definition 6 (Ground Query Term)

1. A query term is called *ground*, if it does not contain (subterm, label, namespace, or positional) variables.

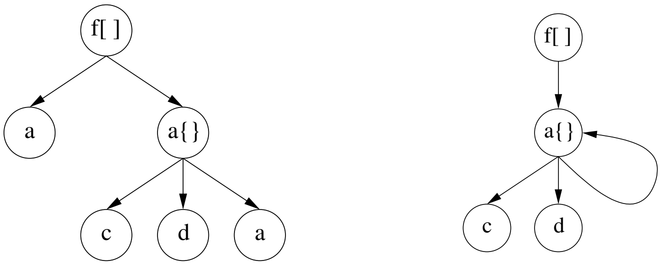
2. $T^g \subset T^q$ denotes the set of all ground query terms, and $T^d \subset T^g$ denotes the set of all data terms.

In the following, we differentiate between the ground query term itself and the graphs induced by a ground query term. Whereas the term itself contains subterms of the form `^id` and `id@t`, all references are dereferenced in the graph induced by the ground query term. By the *position* of a subterm in a ground query term, we mean the position in the list of children of that term. For example, in `f{a,b,c}`, *c* is the subterm at position 3. Likewise, in `f{id@a,^id}`, `id@a` is the subterm at position 1, and `^id` is the subterm at position 2. The position of subterms in the graph induced by a ground query term is defined differently: in the last example, the subterm *a* has both the position 1 and the position 2. For this reason, we will usually speak about *successors* when referring to the graph induced by a ground query term, and about *subterms*, when referring to the syntactical representation of a ground query term.

The *graph induced by a ground query term* (or short: *ground query term graph*) is defined in a straightforward manner as follows.

Definition 7 (Graph Induced by a Ground Query Term)
Given a ground query term t. The *graph induced by t* is a tuple $G_t = (V, E, r)$, with:

Figure 1 Graphs induced by $f[a,a[c,d,a]]$ and $f[[\&1 \ @ \ a\{\{c,d,\uparrow \&1\}\}]]$



1. a set of *vertices* (or *nodes*) V defined as the set of all (immediate and indirect) subterms of t (including t itself).
2. a set of *edges* $E \subseteq V \times V \times \mathbb{N}$ characterised as follows:
   - for all terms $t_1, t_2, t_3 \in V$: if t_2 is the subexpression of t_1 at position i and of the form `^oid` (a referring occurrence), and t_3 is of the form `oid @ t'` (a defining occurrence), with `oid` an identifier and `t'` a term ($\in V$), then $(t_1, t_3, i) \in E$.
   - for all terms $t_1, t_2 \in V$: if t_2 is the subexpression of t_1 at position i and *not* of the form `^oid`, then $(t_1, t_2, i) \in E$.
3. a distinguished vertex $r \in V$ called the *root node* with $r = t$.

The *label* of a vertex is either the label, the string value, or the regular expression of the subterm it represents.

Representing vertices as complete subterms and edges with positions is necessary for the definition of the simulation relation as it conveys information about ordered/unordered and partial/total term specifications and the respective positions of subterms in a term. Figure 1 illustrates this definition on two ground query terms. Note that for space reasons, the vertices in both graphs do not contain the subterms, but only the term labels and specifications.

The following additional terminology from graph theory is used below. Let $G = (V, E, r)$ be the graph induced by a ground query term. For any two nodes $v_1 \in V$ and $v_2 \in V$, if $(v_1, v_2, i) \in E$ for some integer i (i.e. there is an edge from v_1 to v_2), v_1 and v_2 are called *adjacent*, v_2 is the i^{th} *successor* of v_1, and v_1 is a *predecessor* of v_2.

2.4 Term Sequences and Successors

The following sections use the notion of (finite) *term sequences* to represent the (immediate) successors of a term. Note that sequences of subterms are used regardless of the kind of subterm specification: in case of unordered term specifications, there is still a sequence of subterms given by the syntactical representation of the term.

Recall in the following that a function $f : N \rightarrow M$ can be seen as a (binary) relation $f \subseteq N \times M$ such that for every two different pairs $(n_1, m_1) \in f$ and $(n_2, m_2) \in f$ holds that $n_1 \neq n_2$. Considering a function as a relation is more convenient for the representation of sequences. A function $f : N \rightarrow M$ is furthermore called *total*, if f is defined for every element of N.

Definition 8 (Term Sequence)

1. Let X be a set of terms and let $N = \{1, \ldots, n\}$ ($n \geq 0$) be a set of non-negative integers. A *term sequence* is a total function $S \subseteq N \times X$ mapping integers to terms.

   Instead of writing $S = \{(1,a),(2,b),\ldots\}$, term sequences are often denoted by $S = \langle a,b,\ldots\rangle$.

2. Let S be a term sequence, and let $s = (i,t)$ be an element in S.

   - the *index* of s is defined as $index(s) = i$ (projection on the first element)
   - the *term* of s is defined as $term(s) = x$ (projection on the second element)

If $S = \langle \ldots, a, \ldots \rangle$ is a term sequence, i.e. $S = \{\ldots,(a,i),\ldots\}$, then $term((a,i)) = a$. Since using $term((a,i))$ is very inconvenient, we shall often write a instead of (a,i) and e.g. use $a \in S$ instead of $(a,i) \in S$. Accordingly, we use the notion $index(a)$ to represent the position of the subterm a in the term sequence, unless we have to distinguish multiple occurrences of a in S.

Note that empty term sequences are not precluded by the definition, and term sequences are always finite, because they serve to represent the (immediate) successors of a term. Instead of *term sequence*, we shall often simply write *sequence* as other sequences are not considered in this work. The *index* of an element can also be called the *position* of that element. However, the notion *index* is preferred to better distinguish between the `position` construct in a query term and the position in the sequence.

Sequences allow for multiple occurrences of the same term. For example, both $S = \langle a,b,a\rangle = \{(1,a),(2,b),(3,a)\}$ and $T = \langle a,a,b\rangle = \{(1,a),(2,a),(3,b)\}$ are term sequences of a and b.

Based on the graph induced by a ground query term, the definition of the sequence of successors is as expected:

Definition 9 (Sequence of Successors)
Let t be a ground query term, let $G_t = (V,E,t)$ be the graph induced by t, and let $v \in V$ be a node in G_t (i.e. subterm of t). The *sequence of successors* of v, denoted $Succ(v)$, is defined as

$$Succ(v) = \{(i,v') \mid (v,v',i) \in E\}$$

Note that $Succ(v)$ may be the empty sequence $\langle\ \rangle$, if v does not have successors.

Consider the term $t_1 = f\{a,a,b\}$. The sequence of successors of t_1 is $Succ(t_1) = \langle a,a,b\rangle = \{(1,a),(2,a),(3,b)\}$. Consider furthermore $t_2 = o1@f[a, \uparrow o1, b]$. The sequence of successors of t_2 is $Succ(t_2) = \langle a, o1@f[a,\uparrow o1,b], b\rangle = \{(1,a),(2,o1@f[a,\uparrow o1,b]),(3,b)\}$. Note that the reference in t_2 is dereferenced (one level).

Mostly, the sequence of successors and the sequence of (immediate) subterms of a term coincide. The most significant difference is that the sequence of successors is already dereferenced, i.e. all references are "replaced" by the subterms they refer to. For this reason, the remainder of this Section uses the term *successors* instead of *subterms*. Although it is somewhat imprecise, the notion *subterm* is often added in parentheses to emphasise the coincidence of the two sequences in most cases.

In Section 4, the following additional notions of subsequences and concatenation of sequences are needed. Both definitions are straightforward. In order to distinguish subsequences from subsets, we usually write $S' \sqsubseteq S$.

Definition 10 (Subsequences, Concatenation of Sequences)
Let $S = \langle s_1,\ldots,s_m\rangle$ and $T = \langle t_1,\ldots,t_n\rangle$ be term sequences.

1. T is called a *subsequence* of S, denoted $T \sqsubseteq S$, if there exists a strictly monotonic mapping π such that for each $(i,x) \in T$ there exists $(\pi(i),x) \in S$.

2. The *concatenation* of S and T, denoted $S \circ T$, is defined as

$$S \circ T = \langle s_1, \ldots, s_m, t_1, \ldots, t_n \rangle$$

Consider for example the sequences $S_1 = \langle a,b \rangle = \{(1,a),(2,b)\}$ and $S_2 = \langle a,a,b \rangle = \{(1,a),(2,a),(3,b)\}$. S_1 is a subsequence of S_2 with $\pi(1) = 1, \pi(2) = 3$ or with $\pi(1) = 2, \pi(2) = 3$. The concatenation of S_1 and S_2 yields

$$S_1 \circ S_2 = \langle a,b,a,a,b \rangle = \{(1,a),(2,b),(3,a),(4,a),(5,b)\}$$

2.5 Substitutions and Substitution Sets

In principle, the usual notion of substitutions is also used for Xcerpt terms. However, variable restrictions occurring in query terms have to be taken into account. As a variable might be restricted, not every substitution is applicable to every query term.

Also, Xcerpt construct terms extend the usual terms by grouping constructs that group several substitutions within a single ground instance by using the constructs `all` and `some`. For instance, given a construct term $f\{all\ var\ X\}$ and three alternative substitutions $\{X \mapsto a\}$, $\{X \mapsto b\}$ and $\{X \mapsto c\}$, the resulting data term is $f\{a,b,c\}$.

In order to define such groupings, it is therefore necessary to provide a construct that represents all possible alternatives and can be applied to a construct term. This is called a *substitution set* below. Since the application of substitution sets to query and construct terms involves some complexity, it is described separately in Section 4. Substitution sets are then used in Section 6 which defines satisfaction for Xcerpt term formulas. In the following, substitutions are denoted by lowercase greek letters (like σ or π), while substitution sets are denoted by uppercase greek letters (like Σ or Π).

2.5.1 Substitutions

A *substitution* is a mapping from the set of (all) variables to the set of (all) construct terms. In the following, lower case greek letters (like σ or τ) are usually used to denote substitutions. As usual in mathematics, a substitution is a mapping of infinite sets. Of course, finite representations are usually used, as the number of variables occurring in a term is finite. Substitutions are often conveniently denoted as sets of variable assignments instead of as functions. For example, we write $\left\{X \mapsto a, Y \mapsto b\right\}$ to denote a substitution that maps the variable X to a and the variable Y to b, and any other variable to arbitrary values. In general, a substitution provides assignments for all variables, but "irrelevant" variables are not given in the description of substitutions.

If a substitution is *applied* to a query term t^q, all occurrences of variables for which the substitution provides assignments are replaced by the respective assignments (see Section 4.1 below). The resulting term is called an *instance* of t^q and the substitution. Not every substitution can be applied to every query term: variable assignments in the substitution have to respect variable restrictions occurring in the pattern for a substitution to be applicable (see also 4.1). If a substitution σ respects the variable restrictions in a query term t^q, it is said to be *a substitution for t^q*. For example, the substitution $\left\{X \mapsto f\{a\}\right\}$ is a substitution for $var\ X \leadsto f\{\{\}\}$, but not for $var\ X \leadsto g\{\{\}\}$. Note that a substitution cannot be applied to a construct term, because construct terms may contain grouping constructs that group several instances of subterms together. Instead, substitution sets are used for this purpose (see below).

A substitution σ is called a *grounding substitution* for a term t, if $\sigma(t)$ is a ground query term. Consequently, a grounding substitution is always a mapping from the set of variable names to the set of data terms (i.e. ground construct terms). A substitution σ is called an *all-grounding substitution*, if it maps every variable to a data term. Naturally, every all-grounding substitution is a grounding substitution for every query term to which it is applicable. Note that the reverse does not hold: a grounding substitution is grounding wrt. some term t and does not necessarily assign ground terms to variables not occurring in t.

A substitution σ_1 is a *subset* of a substitution σ_2 (i.e. $\sigma_1 \subseteq \sigma_2$), if $\sigma_1(X) \cong \sigma_2(X)$ for every variable name X with $\sigma_1(X) \neq X$ (i.e. σ_1 does not map X to itself), where $\cong$ denotes simulation equivalence (i.e. mutual simulation, cf. Section 5.3). Correspondingly, two substitutions σ_1 and σ_2 are considered to be *equal* (i.e. $\sigma_1 = \sigma_2$), if $\sigma_1 \subseteq \sigma_2$ and $\sigma_2 \subseteq \sigma_1$. For example, $\{X \mapsto f\{a,b\}\}$ and $\{X \mapsto f\{b,a\}\}$ are equal. This definition is reasonable because the data terms resulting from applying two such substitutions are treated equally in the model theory described below.

The *composition* of two substitutions σ_1 and σ_2, denoted by $\sigma_1 \circ \sigma_2$ is defined as $(\sigma_1 \circ \sigma_2)(t) = \sigma_1(\sigma_2(t))$ for every query term t. Note that the assignments in σ_2 take precedence, because σ_2 is applied first. Consider for example $\sigma_1 = \{X \mapsto a, Y \mapsto b\}$ and $\sigma_2 = \{X \mapsto c\}$, and a term $t = f\{var\, X, var\, Y\}$. Applying the composition $\sigma_1 \circ \sigma_2$ to t yields $(\sigma_1 \circ \sigma_2)(t) = f\{c,b\}$.

The *restriction* of a substitution σ to a set of variable names V, denoted by $\sigma_{|V}$, is the mapping that agrees with σ on V and with the identical mapping on the other variables.

2.5.2 Substitution Sets

A *substitution set* is simply a set containing substitutions. In the following, upper case greek letters (like Σ and Φ) are usually used to denote substitution sets.

Substitution sets can be *applied* to a query *or* construct term (cf. Sections 4.1 and 4.2). The result of this application is in general a set of terms called the *instances* of the substitution set and the term. A substitution set Σ is only applicable to a query term t^q, if all substitutions in Σ are applicable to t^q. In this case, Σ is called *a substitution set for t^q*. Since construct terms do not contain variable restrictions, every substitution set except for the empty set is a substitution set for a construct term. There exists no query or construct term t such that the empty substitution set $\{\}$ is a substitution set for t.

A substitution set Σ for a term t is called a *grounding substitution set*, if all instances of t and Σ are ground query terms or data terms. A substitution set Σ is called an *all-grounding substitution set*, if all $\sigma \in \Sigma$ are all-grounding substitutions.

The *composition* of two substitution sets Σ_1 and Σ_2, denoted as $\Sigma_1 \circ \Sigma_2$, is defined as

$$\Sigma_1 \circ \Sigma_2 = \left\{\sigma_1 \circ \sigma_2 \mid \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2\right\}$$

Consider for example the substitution sets $\Sigma_1 = \left\{\{X \mapsto a\}\right\}$ and $\Sigma_2 = \left\{\{Y \mapsto b\}, \{Y \mapsto c\}\right\}$. Then $\Sigma_1 \circ \Sigma_2 = \left\{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto a, Y \mapsto c\}\right\}$.

The *restriction* of a substitution set Σ to a set of variables V, denoted by $\Sigma_{|V}$, is the set of substitutions in Σ restricted to V.

Similarly, the *extension* of a substitution set Σ restricted to a set of variables V to a set of variables V' with $V \subseteq V'$, extends every substitution σ in Σ to substitutions σ' by adding all possible assignments of variables in $V' \setminus V$ to data terms. For example, the extension of the restricted substitution set $\left\{\{X \mapsto a\}\right\}$ to the set of variables $\{X,Y\}$ is the (infinite) set $\left\{\{X \mapsto a, Y \mapsto a\}, \{X \mapsto a, Y \mapsto b\}, \dots\right\}$

Note that in practice, it would be desirable to define substitution sets as *multi-sets* that may contain duplicate elements: if an XML document contains two persons named “Donald Duck”, then it should be assumed that these are different persons with the same name. Providing a proper formalisation with multi-sets is, however, not in the scope of this article, as subsequent definitions and proofs would be much more complicated without adding an interesting aspect to the formalisation.

2.5.3 Maximal Substitution Sets

So as to properly convey the meaning of `all`, it is not sufficient to consider arbitrary substitution sets. The interesting substitution sets are those that are *maximal* for the satisfaction of the query part Q of a rule. As satisfaction is not yet formally defined, this property shall for now simply be called P.

Intuitively, the definition of maximal substitution sets is straightforward: a substitution set Σ satisfying P is a maximal substitution set, if there exists no substitution set Φ satisfying P such that Σ is a proper subset of Φ. However, this informal definition does not take into account that there might be substitution sets that differ only in that some substitutions contain bindings that are irrelevant because they do not occur in the considered term formula Q. Maximal substitution sets are therefore formally defined as follows:

Definition 11 (Maximal Substitution Set)
Let Q be a quantifier free query term formula with set of variables V, let P be a property, and let Σ be a set of substitutions such that P holds for Σ. Σ is called a *maximal substitution set wrt. P and Q*, if there exists no substitution set Φ such that P holds for Φ and $\Sigma_{|V}$ is a proper subset of $\Phi_{|V}$ (i.e. $\Sigma_{|V} \subset \Phi_{|V}$).

3 Terms as Formulas

Classical logic distinguishes between

- terms, which are composed of function symbols and serve as data structures representing objects of the application domain at hand, and
- atomic formulas, which are composed of relation symbols and terms and represent statements about objects of the application domain.

Statements represented by formulas have truth values, objects represented by terms have no truth value. In contrast, XML and Web data does not need this distinction, because it has no (formal) semantics and merely holds semistructured data. Therefore, Xcerpt terms (corresponding to Web data) are considered as being atomic formulas representing the statement that the respective terms “exist”. A salient aspect of this representation is the possibility to specify integrity constraints for data terms. These are, however, not covered in depth in this article.

3.1 Term Formulas

Atomic formulas are composed of Xcerpt query, construct, and data terms, and of the two special terms $\bot$ and $\top$ (denoting falsity and truth). As an intuition, such atomic formulas are statements about the existence or satisfiability of a term. Compound formulas can be constructed in the usual manner using the binary connectives $\vee$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$, the unary connective $\neg$, the zero-ary connectives $\top$ and $\bot$, and the quantifiers $\forall$ and $\exists$. Instead of quantifying each variable separately, the construct $\forall^*$ may be used to universally quantify all free variables in a formula. Also, instead of writing $F_1 \vee \cdots \vee F_n$, we sometimes write $\bigvee_{1 \leq i \leq n} F_i$, and instead of writing $F_1 \wedge \cdots \wedge F_n$, we sometimes write $\bigwedge_{1 \leq i \leq n} F_i$.

In the following, formulas built in this manner shall be called *Xcerpt term formulas*, or simply *term formulas*. If a term formula consists only of query terms, it is also called *query term formula*, if it consists only of construct terms, it is called *construct term formula*.

Example 12
The following example shows a term formula built up from query terms, implications and quantifiers. It represents an integrity constraint that requires all books in the `bib.xml` document to have at least one author:

```
∀ B . bib{{ var B → book{{ }} }} ⇒
    ∃ A . bib{{ var B → book{{ authors{{ var A }} }} }}
```

3.2 Xcerpt Programs as Formulas

Like in traditional logic programming, rules in Xcerpt are implications. However, Xcerpt rules with grouping constructs have a particular semantics that cannot be represented as implications in the usual manner. We therefore keep the denotation $t^c \leftarrow Q$ to represent rules.

In addition to the usual quantifiers $\forall$ and $\exists$, the grouping constructs `all` and `some` that may be part of a construct term may bind variables in a formula within a specific scope, usually the head and body of a rule. As these constructs are contained within the term structure, their scope is not immediately apparent. It is thus useful to introduce new symbols $\ll \cdot \gg$ that are used to indicate the scope of *all* the grouping constructs contained in them. In practice, it is neither desirable nor useful to have scopes extending over different subformulas for the grouping constructs contained in a single construct term, thus a single scope for all grouping constructs suffices. The grouping constructs of a construct term always refer to the variables of a single rule and thus all have the same scope.

Example 13
Consider for example the program (in formula notation)

```
g{a,b,c}
f{all var X} ← g{{var X}}
```

The scope of the `all` construct in the rule head is made explicit using $\ll \cdot \gg$ in the following manner:

```
g{a,b,c} ∧ ≪f{all var X} ← g{{var X}}  ≫
```

As usual, formulas representing programs are always considered to be universally closed, even if quantifiers are not explicitly given.

Example 14
Consider the following Xcerpt program (in the notation introduced in Section 2 and with internalised resources):

```
f{all var X, var Y} ← and{ g{{var X}}, h{{ e{var X,var Y} }} }
g[ var X ]           ← h{{ e[var X] }}
h[ e[a,1], e[b,1], e[c,1], e[d,2] ]
```

The formula representation of this program is as follows:

```
∀ Y ≪f{all var X, var Y} ← g{{var X}} ∧ h{{ e{var X,var Y} }} ≫ ∧
∀ X ≪g[ var X ]  ← h{{ e[var X] }} ≫ ∧
h[ e[a,1], e[b,1], e[c,1], e[d,2] ]
```

The variable `X` in the first rule is in the scope of the `all` construct in the rule head, while the variable `Y` is in the scope of the universal quantification represented by $\forall Y$. Note that the scope of the `all` is restricted to the first rule and the occurrences of `X` in the second rule are not affected (thus $\forall X$ in the second rule).

4 Application of Substitutions to Xcerpt Terms

4.1 Application to Query Terms

Since query terms do not contain the grouping constructs *all* and *some*, applying substitutions and substitution sets is straightforward. Application of a single substitution yields a *single* term where some variable occurrences are substituted, while application of a substitution set yields a *set* of terms where some variables are substituted.

Definition 15 (Substitutions: Application to Query Terms)
Let t^q be a query term.

1. The application of a *substitution* σ to t^q, written $\sigma(t^q)$ is recursively defined as follows:
   - $\sigma(var\ X) = t'$ if $(X \mapsto t') \in \sigma$
   - $\sigma(var\ X \leadsto s) = t'$ if $(X \mapsto t') \in \sigma$ and $\sigma(s) \preceq t'$
   - $\sigma(f\{t_1,\ldots,t_n\}) = \sigma(f)\{\sigma(t_1),\ldots,\sigma(t_n)\}$
   - $\sigma(f[t_1,\ldots,t_n]) = \sigma(f)[\sigma(t_1),\ldots,\sigma(t_n)]$
   - $\sigma(f\{\{t_1,\ldots,t_n\}\}) = \sigma(f)\{\{\sigma(t_1),\ldots,\sigma(t_n)\}\}$
   - $\sigma(f[[t_1,\ldots,t_n]]) = \sigma(f)[[\sigma(t_1),\ldots,\sigma(t_n)]]$
   - $\sigma(without\ t) = without\ \sigma(t)$
   - $\sigma(optional\ t) = optional\ \sigma(t)$

   for some $n \geq 0$.

2. The application of a *substitution set* Σ to t^q is defined as follows:

$$\Sigma(t^q) = \left\{\sigma(t^q)\ \mid\ \sigma \in \Sigma\right\}$$

Note that not every substitution can be applied to a query term t^q. If a variable in t^q is restricted as in $var\ X \leadsto s$, then a substitution can only be applied if it provides bindings for X that are compatible to this restriction. Likewise, a substitution set is only applicable to a query term t^q, if all its substitutions are applicable to t^q.

Since query terms never contain grouping constructs, the cardinality of $\Sigma(t)$ always equals the cardinality of Σ. In particular, if $\Sigma = \emptyset$, then $\Sigma(t) = \emptyset$, even if t is a ground query term. Since an interpretation with an empty substitution set would be a model for any formula, substitution sets in the following are considered to be non-empty. In case no variables are bound, substitution sets are usually defined as $\Sigma = \{\emptyset\}$.

4.2 Application to Construct Terms

Applying a single substitution to a construct term is not reasonable as the meaning of the grouping constructs *all* and *some* is unclear in such cases. In the following, the application is thus only defined for substitution sets. On substitution sets, the grouping constructs group such substitutions that have the same assignment on the *free variables* of a construct term. For each such group, the application of the substitution Σ yields a different construct term. A variable is considered *free* in a construct term if it is not in the scope of a grouping construct. The set of free variables of a construct term t^c is denoted by $FV(t^c)$. The relation $\cong$ denotes simulation equivalence between two ground terms and is defined later in this article.

Definition 16 (Grouping of a Substitution Set)
Given a substitution set Σ and a set of variables $V = \{X_1, \ldots, X_n\}$ such that all $\sigma \in \Sigma$ have bindings for all $X_i, 1 \leq i \leq n$.

- The equivalence relation $\simeq_V \subseteq \Sigma \times \Sigma$ is defined as: $\sigma_1 \simeq_V \sigma_2$ iff $\sigma_1(X) \cong \sigma_2(X)$ for all $X \in V$.
- The set of equivalence classes $\Sigma/_{\simeq_V}$ with respect to $\simeq_V$ is called the *grouping of* Σ *on* V.
- Each of the equivalence classes $[\![\sigma]\!] \in \Sigma/_{\simeq_V}$ is accordingly defined as $[\![\sigma]\!] = \left\{ \tau \in \Sigma \mid \tau \simeq_V \sigma \right\}$.

Informally, each equivalence class $[\![\sigma]\!] \in \Sigma/_{\simeq_V}$ contains such substitutions that have the same assignment for each of the variables in V.

Example 17
Given the substitution set $\Sigma = \left\{\sigma_1, \sigma_2, \sigma_3\right\}$ with

$$\sigma_1 = \{X_1 \mapsto a, X_2 \mapsto b\}, \sigma_2 = \{X_1 \mapsto a, X_2 \mapsto c\}, \text{ and } \sigma_3 = \{X_1 \mapsto c, X_2 \mapsto b\}$$

The grouping of Σ on $V = \{X_1\}$ is

- $[\![\sigma_1]\!] = [\![\sigma_2]\!] = \left\{\{X_1 \mapsto a, X_2 \mapsto b\}, \{X_1 \mapsto a, X_2 \mapsto c\}\right\}$
- $[\![\sigma_3]\!] = \left\{\{X_1 \mapsto c, X_2 \mapsto b\}\right\}$

The application of a substitution set to a construct term (possibly containing grouping constructs) is defined in terms of this grouping. Given a substitution set Σ, the application $\Sigma(t^c)$ to a construct term t^c with free variables $FV(t^c)$ yields exactly $|\Sigma/_{\simeq_{FV(t^c)}}|$ results, one for each different binding of the free variables in t^c.

Example 18
Given a term $t = f\{X_1, g\{all\ X_2\}\}$, i.e. $FV(t) = \{X_1\}$. Consider again

$$\Sigma = \left\{\{X_1 \mapsto a, X_2 \mapsto b\}, \{X_1 \mapsto a, X_2 \mapsto c\}, \{X_1 \mapsto c, X_2 \mapsto b\}\right\}$$

from Example 17. The result of applying Σ to t is

$$\Sigma(t) = \left\{ f\{a, g\{b, c\}\}, f\{c, g\{b\}\}\right\}$$

The following definition specifies how a substitution set is applied to a construct term t^c. The definition is divided into two parts: In the first part, it is assumed that all substitutions in the substitution set Σ contain the same assignments for the free variables of t^c (variables occurring within the scope of grouping constructs are unrestricted). As the quotient $\Sigma/_{\simeq_{FV(t^c)}}$ in this case obviously only contains a single equivalence class, the application of this restricted Σ to t^c yields only a single term, which simplifies the recursive definition. In the second part of Definition 19, this restriction is lifted.

Since the construction of data terms requires to construct new lists of subterms, the following definition(s) use the notion of *term sequences* introduced in Section 2.4. Recall that a sequence is a binary relation between a set of integers and a set of terms, and usually denoted by $S = \langle x_1, \ldots, x_n \rangle$ for some n and terms x_i. Recall furthermore the definitions of *subsequences* and *concatenation* (Definition 10 on page 10).

Defining the semantics of `order by` furthermore requires a function $sort_{f(V)}(\cdot, \cdot)$, where V is a sequence of variables, that takes as arguments a grouping of a substitution set on V and returns a sequence of substitution sets ordered according to $f(V)$ and the variables in V. $f(V)$ is a total ordering on the set of substitution sets that assign ground terms to the variables in V comparing variable bindings for the variables in V. [1]

Definition 19 (Substitutions: Application to Construct Terms)

1. Let Σ be a substitution set and let t^c be a construct term such that all free variables of t^c have the same assignment in all substitutions of Σ, i.e. $\Sigma/_{\simeq_{FV(t^c)}} = \{[\![\sigma]\!]\}$. The restricted application of Σ to t^c, written $[\![\sigma]\!](t^c)$, is recursively defined as follows:

   - $[\![\sigma]\!](var\ V) = \langle \sigma(V) \rangle^2$
   - $[\![\sigma]\!](f\{t_1, \dots, t_n\}) = \langle [\![\sigma]\!](f)\{[\![\sigma]\!](t_1) \circ \cdots \circ [\![\sigma]\!](t_n)\}\rangle$ for some $n \geq 0$
   - $[\![\sigma]\!](f[t_1, \dots, t_n]) = \langle [\![\sigma]\!](f)[[\![\sigma]\!](t_1) \circ \cdots \circ [\![\sigma]\!](t_n)]\rangle$ for some $n \geq 0$
   - $[\![\sigma]\!](all\ t) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!], \dots, [\![\tau_k]\!]\} = [\![\sigma]\!]/_{\simeq_{FV(t)}}$
   - $[\![\sigma]\!](all\ t\ group\ by\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!], \dots, [\![\tau_k]\!]\} = [\![\sigma]\!]/_{\simeq_{FV(t) \cup V}}$
   - $[\![\sigma]\!](all\ t\ order\ by\ f\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$
     where $\langle [\![\tau_1]\!], \dots, [\![\tau_k]\!]\rangle = sort(f(V), [\![\sigma]\!]/_{\simeq_{FV(t) \cup V}})$
   - $[\![\sigma]\!](some\ k\ t) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!], \dots, [\![\tau_k]\!]\} \subseteq [\![\sigma]\!]/_{\simeq_{FV(t)}}$
   - $[\![\sigma]\!](some\ k\ t\ group\ by\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!], \dots, [\![\tau_k]\!]\} \subseteq [\![\sigma]\!]/_{\simeq_{FV(t) \cup V}}$
   - $[\![\sigma]\!](some\ k\ t\ order\ by\ f\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$
     where $\langle [\![\tau_1]\!], \dots, [\![\tau_k]\!]\rangle \sqsubseteq sort(f(V), [\![\sigma]\!]/_{\simeq_{FV(t) \cup V}})$

   where $[\![\tau]\!]1, \dots, [\![\tau]\!]k$ are pairwise different substitution sets.

2. Let t^c be a term, and let $FV(t^c)$ be the free variables in t^c. The application of a *substitution set* Σ to t^c is defined as follows:

$$\Sigma(t) = \left\{ t^{c'} \mid [\![\sigma]\!] \in \Sigma/_{\simeq_{FV(t^c)}} \ \wedge\ \langle t^{c'} \rangle = [\![\sigma]\!](t^c) \right\}$$

Although not explicitly defined above, integrating aggregations and functions in this definition is straightforward.

Example 20

Consider the substitution set

$$\Sigma = \big\{ \{X \mapsto f\{a\}, Y \mapsto g\{a\}\},\ \{X \mapsto f\{a\}, Y \mapsto g\{b\}\},\ \{X \mapsto f\{b\}, Y \mapsto g\{a\}\} \big\}$$

and the construct terms $t_1 = h\{all\ var\ X, var\ Y\}$ and $t_2 = h\{var\ X, all\ var\ Y\}$. Grouping Σ according to the free variables $FV(t_1) = \{Y\}$ in t_1 and $FV(t_2) = \{X\}$ in t_2 yields

$$\begin{array}{lcl} \Sigma/_{\simeq_{FV(t_1)}} & = & \Big\{ \big\{ \{X \mapsto f\{a\}, Y \mapsto g\{a\}\}, \{X \mapsto f\{b\}, Y \mapsto g\{a\}\} \big\},\ \big\{ \{X \mapsto f\{a\}, Y \mapsto g\{b\}\} \big\} \Big\} \\ \Sigma/_{\simeq_{FV(t_2)}} & = & \Big\{ \big\{ \{X \mapsto f\{a\}, Y \mapsto g\{a\}\}, \{X \mapsto f\{a\}, Y \mapsto g\{b\}\} \big\},\ \big\{ \{X \mapsto f\{b\}, Y \mapsto g\{a\}\} \big\} \Big\} \end{array}$$

The ground instances of t_1 and t_2 by Σ are thus

$$\begin{array}{lcl} \Sigma(t_1) & = & \{\ h\{f\{a\}, f\{b\}, g\{a\}\},\ h\{f\{a\}, g\{b\}\}\ \} \\ \Sigma(t_2) & = & \{\ h\{f\{a\}, g\{a\}, g\{b\}\},\ h\{f\{a\}, g\{b\}\}\ \} \end{array}$$

[1] As the substitution set is grouped on V, all substitutions in $[\![\sigma]\!]$ (respectively $[\![\tau]\!]$) provide identical bindings for variables in V.

[2] Note that σ is the representative of the equivalence class $[\![\sigma]\!]$

4.3 Application to Query Term Formulas

In the following, it is often interesting to study ground instances not only of terms but also of compound formulas. The following definition defines the application of substitution sets to formulas consisting only of query terms (so-called *query term formulas*); construct terms are problematic, as they group several substitutions and thus do not behave “synchronously” with query terms in the same formula. Fortunately, the formalisation of Xcerpt programs does not need to consider formulas containing construct terms. The only exception are program rules, which are treated separately anyway.

Applying a substitution set to a query term formula is straightforward: as each substitution in a substitution set represents a different alternative, the application of the substitution set to a query term formula simply yields a conjunction of all different instances.

Definition 21 (Substitutions: Application to Query Term Formulas)
Let F be a quantifier-free term formula where all atoms are query terms (a *query term formula*).

1. The application of a *substitution* σ to F, written $\sigma(F)$, is recursively defined as follows:
   - $\sigma(F_1 \wedge F_2) = \sigma(F_1) \wedge \sigma(F_2)$
   - $\sigma(F_1 \vee F_2) = \sigma(F_1) \vee \sigma(F_2)$
   - $\sigma(\neg F') = \neg\sigma(F')$
   - $\sigma(\neg F') = \neg\sigma(F')$
2. The application of a *substitution set* Σ to F, written $\Sigma(F)$, is defined as follows:

$$\Sigma(F) = \bigwedge_{\sigma\in\Sigma} \sigma(F)$$

5 Simulation and Simulation Unifiers

Matching query terms with data terms is based on the notion of *rooted graph simulations* [HHK96, Mil71]. Intuitively, a query term matches with a data term, if there exists at least one substitution for the variables in the query term (called *answer substitution* of the query term) such that the corresponding graph induced by the resulting *ground* query term simulates in the graph induced by the data term. Of course, graph simulation needs to be modified to take into account the different term specifications, descendant construct, optional subterms, subterm negation, and regular expressions.

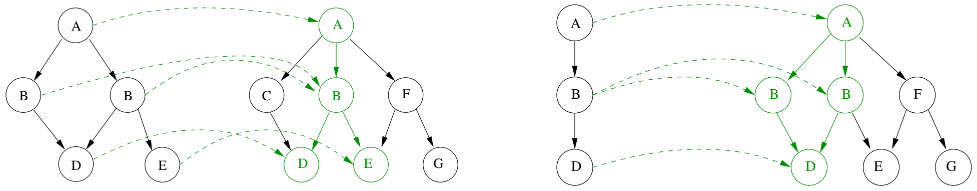
To simplify the formalisation below, it is assumed that strings and regular expressions are represented as compound terms with the string or regular expression as label, no subterms, and a total term specification. For example, the string `"Hello, World"` is represented as the term `"Hello, World"{}`.

5.1 Rooted Graph Simulation

Pattern matching in Xcerpt (and UnQL, for that matter) is based on a similarity relation between the graphs induced by two semistructured expressions, which is called *graph simulation* [HHK96, Mil71]. Graph simulation is a relation very similar to graph homomorphisms, but more general in the sense that it allows to match two nodes in one graph with a single node in the other graph and vice versa.

The following definition is inspired by [HHK96, Mil71] and refines the simulation considered in [BS02]. Recall that a (directed) rooted graph $G = (V, E, r)$ consists in a set V of vertices, a set E of edges (i.e. ordered pairs of vertices), and a vertex r called the root of G such that G contains a path from

Figure 2 Rooted Graph Simulations (with respect to vertex adornment equality)



r to each vertex of G. Note that the initial definition of a rooted graph simulation does not take into account the edge labels of graphs induced by a semistructured expression, it is defined on generic, node labelled and rooted graphs. Note furthermore, that in general, there might be more than one simulation between two graphs, which leads to the notion of *minimal* simulations also defined below.

Definition 22 (Rooted Graph Simulation)
Let $G_1 = (V_1, E_1, r_1)$ and $G_2 = (V_2, E_2, r_2)$ be two rooted graphs and let $\sim \subseteq V_1 \times V_2$ be an order or equivalence relation. A relation $S \subseteq V_1 \times V_2$ is a *rooted simulation* of G_1 in G_2 with respect to if:

1. $r_1 \; S \; r_2$.
2. If $v_1 \; S \; v_2$, then $v_1 \sim v_2$.
3. If $v_1 \; S \; v_2$ and $(v_1, v_1', i) \in E_1$, then there exists $v_2' \in V_2$ such that $v_1' \; S \; v_2'$ and $(v_2, v_2', j) \in E_2$

A rooted simulation S of G_1 in G_2 with respect to $\sim$ is *minimal* if there are no rooted simulations S' of G_1 in G_2 with respect to $\sim$ such that $S' \subset S$ (and $S \neq S'$).

Definition 22 does not preclude that two distinct vertices v_1 and v_1' of G_1 are simulated by the same vertex v_2 of G_2, i.e. $v_1 \; S \; v_2$ and $v_1' \; S \; v_2$. Figure 2 gives examples of simulations with respect to the equality of vertex adornments. The simulation of the right example is not minimal.

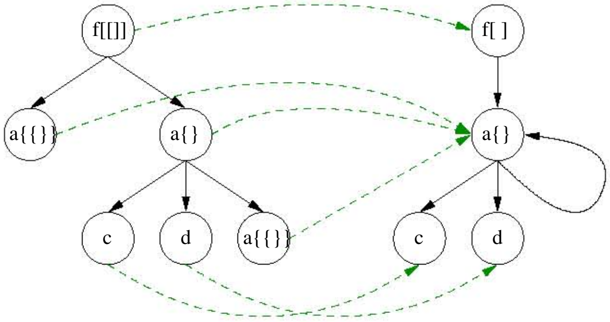
The *existance* of a simulation relation between two graphs (without variables) can be computed efficiently: results presented in [Kil92] give rise to the assumption that such problems can generally be solved in polynomial time and space. However, computation of pattern matching usually requires to compute not only one, but all minimal simulations between two graphs, in which case the complexity increases with the size of the "answer".

5.2 Ground Query Term Simulation

Using the graphs induced by ground query terms (cf. Definition 7), the notion of rooted simulation almost immediately extends to all ground query terms: intuitively, there exists a simulation of a ground query term t_1 in a ground query term t_2 if the labels and the structure of (the graph induced by) t_1 can be found in (the graph induced by) t_2 (see Figure 3). So as to define an ordering on the set of all ground query terms, ground query term simulation is designed to be transitive and reflexive.

Naturally, the simulation on ground query terms has to respect the different kinds of term specification: if t_1 has a *total* specification, it is not allowed that there exist successors (i.e. subterms) of t_2 that do not simulate successors of t_1; if t_1 has an *ordered* specification, then the successors of t_2 have to appear in the same order as their partners in t_1 (but there might be additional successors between them if the specification is also partial).

Figure 3 Minimal simulation of $f[[\,a\{\{\ \}\}, a\{\{c,d,a\{\{\ \}\}\ \}\}\ ]]$ in $f[\&1\ @\ a\{c,d,\uparrow \&1\}]$



The definition of *ground query term simulation* is characterised using a mapping between the sequences of successors (i.e. subterms) of two ground terms with one or more of the following properties, depending on the kinds of subterm specifications and occurrences of the constructs `without` and `optional`. Recall that a mapping is called total if it is defined on all elements of a set and partial if it is defined on some elements of a set.

Definition 23
Given two term sequences $M = \langle s_1, \dots, s_m \rangle$ and $N = \langle t_1, \dots, t_n \rangle$.

A partial or total mapping $\pi : M \rightarrow N$ is called

- *index injective*, if for all $s_i, s_j \in M$ with $index(s_i) \neq index(s_j)$ holds that $index(\pi(s_i)) \neq index(\pi(s_j))$
- *index monotonic*, if for all $s_i, s_j \in M$ with $index(s_i) < index(s_j)$ holds that $index(\pi(s_i)) < index(\pi(s_j))$
- *index bijective*, if it is index injective and for all $t_k \in N$ exists an $s_i \in M$ such that $\pi(s_i) = t_k$.
- *position respecting*, if for all $s_i \in M$ such that s_i is of the form `position` $j\ \ s'_i$ holds that $index(\pi(s_i)) = j$
- *position preserving*, if for all $s_i \in M$ such that s_i is of the form `position` $j\ \ s'_i$ holds that $\pi(s_i)$ is of the form `position` $l\ \ t'_k$ and $j = l$.

Index monotonic mappings preserve the order of terms in the two sequences and are used for matching terms with ordered term specifications. *Index bijective* mappings are used for total term specifications.

A *position respecting* mapping maps a term with position specification to a term with the specified position and is required (and only applicable) if the term with the sequence of successors (subterms) N uses total and ordered term specification. E.g. given two terms $f\{\{position\ 2\ b\}\}$ and $f[a,b,b]$, a position respecting mapping maps the subterm *position* 2 b only to the first b, because its position is 2, but not to the second b, because its position is 3.

A *position preserving* mapping maps a term with position specification to a term with the same position specification; it is applicable in case the sequence of successors of the second term N is incomplete with respect to order or breadth, as the exact position cannot be determined otherwise in these cases. In particular, this ensures the reflexivity and transitivity of the ground query term simulation (see Theorem 28 below). E.g. given the terms $f\{\{position\ 2\ b\}\}$ and $f\{a,b,position\ 2\ b\}$, the subterm *position* 2 b of the first term needs to be mapped to the subterm *position* 2 b of the second term, but cannot be mapped to the first b because its position is not “guaranteed”.

To summarise, a *position respecting* mapping *respects* the specified position by mapping the subterm only to a subterm at this position. On the other hand, a *position preserving* mapping *preserves* the position by mapping the subterm only to a subterm with the same position specification.

Besides these properties, ground query term simulation needs a notion of *label matches* to allow matching of string labels, regular expressions, or both:

Definition 24 (Label Match)
A term label l_1 matches with a term label l_2, if

- if l_1 and l_2 both are character sequences or both are regular expressions, then $l_1 = l_2$ or
- if l_1 is a regular expression and l_2 is a character sequence, then $l_2 \in L(l_1)$ where $L(l_1)$ is the language induced by the regular expression l_1

l_1 does not match with l_2 in all other cases.

Example 25

1. the labels of the terms $f\{a,b\}$ and $f\{b,a\}$ match
2. the labels of the terms $f\{a,b\}$ and $g\{b,a\}$ do not match
3. the labels of the terms `/.*/` and `"Hello World"` match
4. the labels of the terms `"Hello World"` and `/.*/` do not match

Let $G = (V,E,t)$ be the graph induced by a ground query term t. In the following, $Succ(t')$ denotes the sequence of all successors (i.e. immediate subterms) of t' in G, $Succ^+(t') \subseteq Succ(t')$ denotes the sequence of all successors of a term t' in G that are not of the form `without` t'', and $Succ^-(t)$ denotes the sequence of all successors of a term t' in G that are of the form `without` t'' (i.e. $Succ^+(t') \uplus Succ^-(t') \equiv Succ(t')$). Furthermore, $Succ^!(t') \subseteq Succ(t')$ denotes the sequence of all successors of a term t' in G that are not of the form `optional` t'', and $Succ^?(t') \subseteq Succ(t')$ denotes the sequence of all successors of a term t' that are of the form `optional` t'' (i.e. $Succ^!(t') \uplus Succ^?(t') \equiv Succ(t')$). Note that $Succ^- \subseteq Succ^!$, because a combination of `without` and `optional` is not reasonable.[3]

Definition 26 (Ground Query Term Simulaton)
Let r_1 and r_2 be ground (query) terms, and let $G_1 = (V_1,E_1,r_1)$ and $G_2 = (V_2,E_2,r_2)$ be the graphs induced by r_1 and r_2. A relation $\preceq \subseteq V_1 \times V_2$ on the sets V_1 and V_2 of immediate and indirect subterms of r_1 and r_2 is called a *ground query term simulation*, if and only if:

1. $r_1 \preceq r_2$ (i.e. the roots are in $\preceq$)
2. if $v_1 \preceq v_2$ and neither v_1 nor v_2 are of the form *desc t* nor have successors of the forms `without` t or `optional` t, then the labels l_1 and l_2 of v_1 and v_2 match and there exists a *total, index injective mapping* $\pi : Succ(v_1) \rightarrow Succ(v_2)$ such that for all $s \in Succ(v_1)$ holds that $s \preceq \pi(s)$. Depending on the kinds of subterm specifications of v_1 and v_2, π in addition satisfies the following requirements:

[3] `optional` only has effect on the variable bindings, and `without` may never yield variable bindings

v_1	v_2	it holds that
$l_1[s_1,\dots,s_m]$	$l_2[t_1,\dots,t_n]$	π is *index bijective* and *index monotonic*
$l_1\{s_1,\dots,s_m\}$	$l_2[t_1,\dots,t_n]$	π is *index bijective* and *position respecting*
	$l_2\{t_1,\dots,t_n\}$	π is *index bijective* and *position preserving*
$l_1[[s_1,\dots s_m]]$	$l_2[t_1,\dots,t_n]$	π is *index monotonic* and *position respecting*
	$l_2[[t_1,\dots,t_n]]$	π is *index monotonic* and *position preserving*
$l_1\{\{s_1,\dots s_m\}\}$	$l_2\{t_1,\dots,t_n\}$	π is *position preserving*
	$l_2[t_1,\dots,t_n]$	π is *position respecting*
	$l_2\{\{t_1,\dots,t_n\}\}$	π is *position preserving*
	$l_2[[t_1,\dots,t_n]]$	π is *position preserving*

3. if $v_1 \preceq v_2$ and v_1 is of the form *desc* t_1, then

   - v_2 is of the form *desc* t_2 and $t_1 \preceq t_2$ (*descendant preserving*, or
   - $t_1 \preceq v_2$ (*descendant shallow*), or
   - there exists a $v_2' \in SubT(v_2)$ such that $v_1 \preceq v_2'$ (*descendant deep*)

In all other cases (e.g. combinations of subterm specifications not listed above), $\preceq$ is no ground query term simulation. In subsequent parts of this article, the symbol $\preceq$ always refers to relations that are ground query term simulations.

Note that although graph simulation allows to relate two nodes of the one graph with a single node of the other graph, it is desirable to restrict simulations between two ground query terms to *injective* cases, i.e. such cases where no two subterms of t_1 are simulated by the same subterm of t_2. While it makes certain queries more difficult, this restriction turned out to be much easier to comprehend for authors of Xcerpt programs and reflected the intuitive understanding of query patterns.

Example 27
The following comprehensive list of examples illustrates the different requirements for a ground query term simulation. They are grouped in categories, each referring to the relevant requirement in Definition 26.

For illustration purposes, subterms are annotated with their index as subscript. This subscript is not considered to be part of the label. Also, `position` is abbreviated as `pos`, `optional` is abbreviated as `opt`, and `without` is abbreviated as $\neg$ for space reasons.

1. **total ordered term specification (cf. requirement 2)**
   Let $t_1 = f[a_1,b_2,c_3]$, $t_2 = f[a_1,b_2,c_3,d_4]$, $t_3 = f[a_1,c_2,b_3]$, $t_4 = f\{a_1,b_2,c_3\}$, and $t_5 = g[a_1,b_2,c_3]$

   - $t_1 \preceq t_1$: there exists a total, index bijective, and index monotonic mapping π from $\langle a_1,b_2,c_3\rangle$ to $\langle a_1,b_2,c_3\rangle$ with $s \preceq \pi(s)$, mapping each subterm to itself.
   - $t_1 \not\preceq t_2$: there exists no index bijective mapping from $\langle a_1,b_2,c_3\rangle$ to $\langle a_1,b_2,c_3,d_4\rangle$, as the two sets have different cardinality.
   - $t_1 \not\preceq t_3$: there exists no index monotonic mapping from $\langle a_1,b_2,c_3\rangle$ to $\langle a_1,c_2,b_3\rangle$ with $s \preceq \pi(s)$; the only mapping that would satisfy $s \preceq \pi(s)$, i.e. $\{a_1 \mapsto a_1, b_2 \mapsto b_3, c_3 \mapsto c_2\}$, is not index monotonic.
   - $t_1 \not\preceq t_4$: the braces of t_1 and t_4 are incompatible.
   - $t_1 \not\preceq t_5$: the labels of t_1 and t_5 do not match.

2. **total unordered term specification (cf. requirement 2)**
   Let $t_1 = f\{a_1,b_2,c_3\}$, $t_2 = f[a_1,b_2,c_3,d_4]$, $t_3 = f[a_1,c_2,b_3]$, $t_4 = f\{a_1,b_2,c_3\}$, and $t_5 = g[a_1,b_2,c_3]$

- $t_1 \preceq t_1$: there exists a total and index bijective mapping π from $\langle a_1, b_2, c_3 \rangle$ to $\langle a_1, b_2, c_3 \rangle$ with $s \preceq \pi(s)$, mapping each subterm to itself, thus being position preserving.
- $t_1 \not\preceq t_2$: there exists no index bijective mapping from $\langle a_1, b_2, c_3 \rangle$ to $\langle a_1, b_2, c_3, d_4 \rangle$, as the two sets have different cardinality.
- $t_1 \preceq t_3$: there exists a total and index bijective mapping π from $\langle a_1, b_2, c_3 \rangle$ to $\langle a_1, c_2, b_3 \rangle$ with $s \preceq \pi(s)$, the mapping $\{a_1 \mapsto a_1, b_2 \mapsto b_3, c_3 \mapsto c_2\}$ (it does not need to be index monotonic) and it is trivially position respecting, because t_1 does not contain position subterms.
- $t_1 \preceq t_4$: there exists a total and index bijective mapping π from $\langle a_1, b_2, c_3 \rangle$ to $\langle a_1, b_2, c_3 \rangle$ with $s \preceq \pi(s)$, mapping each subterm to itself, thus being position preserving.
- $t_1 \not\preceq t_5$: the labels of t_1 and t_5 do not match

3. **partial ordered term specification (cf. requirement 2)**

   Let $t_1 = f[[b_1, c_2]]$, $t_2 = f[a_1, b_2, c_3, d_4]$, $t_3 = f[a_1, c_2, b_3]$, $t_4 = f\{a_1, b_2, c_3\}$, and $t_5 = f[b_1, a_2, c_3]$

   - $t_1 \preceq t_1$
   - $t_1 \preceq t_2$: there exists a total, index injective, and index monotonic mapping $\pi = \{b_1 \mapsto b_2, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$. It is trivially position respecting.
   - $t_1 \not\preceq t_3$: there exists no mapping π with $s \preceq \pi(s)$ that is also index monotonic, because t_3 does not contain b and c in the right order.
   - $t_1 \not\preceq t_4$: the braces of t_1 and t_4 are incompatible.
   - $t_1 \preceq t_5$: there exists a total, index injective, and index monotonic mapping $\pi = \{b_1 \mapsto b_1, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$. It is trivially position respecting.

4. **partial unordered term specification (cf. requirement 2)**

   Let $t_1 = f\{\{b_1, c_2\}\}$, $t_2 = f[a_1, b_2, c_3, d_4]$, $t_3 = f[a_1, c_2, b_3]$, $t_4 = f\{a_1, b_2, c_3\}$, $t_5 = f[b_1, a_2, c_3]$, and $t_6 = f[a_1, b_2, d_3]$. All mappings π on $Succ(t_1)$ are trivially position respecting and position preserving.

   - $t_1 \preceq t_1$
   - $t_1 \preceq t_2$: there exists a total, index injective mapping $\pi = \{b_1 \mapsto b_2, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$
   - $t_1 \preceq t_3$: there exists a total, index injective mapping $\pi = \{b_1 \mapsto b_3, c_2 \mapsto c_2\}$ with $s \preceq \pi(s)$
   - $t_1 \preceq t_4$: there exists a total, index injective mapping $\pi = \{b_1 \mapsto b_2, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$
   - $t_1 \preceq t_5$: there exists a total, index injective mapping $\pi = \{b_1 \mapsto b_1, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$
   - $t_1 \not\preceq t_6$: there exists no total mapping π such that $s \preceq \pi(s)$ holds for all s, as t_6 does not contain a subterm matching with c_2.

5. **position specification (cf. requirement 2)**

   Let $t_1 = f\{\{c_1, \text{pos } 2\ b_2\}\}$, $t_2 = f[a_1, b_2, c_3]$, $t_3 = f[b_1, c_2, a_3]$, $t_4 = f[[a_1, b_2, c_3]]$ and $t_5 = f[[a_1, \text{pos } 2\ b_2, c_3]]$

   - $t_1 \preceq t_1$: there exists a total, index injective, position preserving mapping $\pi = \{c_1 \mapsto c_1, \text{pos } 2\ b_2 \mapsto \text{pos } 2\ b_2\}$ with $s \preceq \pi(s)$
   - $t_1 \preceq t_2$: there exists a total, index injective, position respecting mapping $\pi = \{c_1 \mapsto c_3), \text{pos } 2\ b_2 \mapsto b_2\}$ with $s \preceq \pi(s)$
   - $t_1 \not\preceq t_3$: there exists no position respecting mapping π with $s \preceq \pi(s)$; the only mapping with $s \preceq \pi(s)$ is not position respecting, as it contains $\text{pos } 2\ b_2 \mapsto b_1$.
   - $t_1 \not\preceq t_4$: there exists no position preserving mapping π with $s \preceq \pi(s)$, because t_4 contains no subterm of the form `pos 2` t'; position *respecting* is not sufficient, as t_4 is incomplete and might match further terms with b at a different position than 2, e.g. the term $f[a_1, d_2, b_3, c_4]$, in which case $\preceq$ would not be transitive.

- $t_1 \preceq t_5$: there exists a total, index injective, position preserving mapping $\pi = \{c_1 \mapsto c_3), \text{pos } 2\ b_2 \mapsto \text{pos } 2\ b_)\}$ with $s \preceq \pi(s)$; in contrast to t_4, the term t_5 “preserves transitivity” of $\preceq$.

6. **descendant (cf. requirement 3)**
   Let $t_1 = \mathit{desc}\ f\{a\}$, $t_2 = \mathit{desc}\ f\{a\}$, $t_3 = \mathit{desc}\ f\{\{a,b\}\}$, and $t_4 = g\{f\{a\},h\{b\}\}$
   - $t_1 \preceq t_2$, because $f\{a\} \preceq f\{a\}$
   - $t_1 \not\preceq t_3$, because $f\{a\} \not\preceq f\{\{a,b\}\}$
   - $t_1 \preceq t_4$, because t_4 contains a subterm t'_4 such that $f\{a\} \preceq t'_4$.

5.3 Simulation Order and Simulation Equivalence

Ground query term simulation has been designed carefully to be transitive and reflexive, because it is desirable that ground query term simulation is an ordering over the set T^g of ground query terms. This is necessary e.g. for the definition of *Grouping of a Substitution Set* (cf. Definition 16).

Theorem 28 (Transitivity and Reflexivity of $\preceq$ [Sch04])
$\preceq$ is reflexive and transitive.

With this result, the following corollary follows trivially:

Corollary and Definition 29
$\preceq$ defines a preorder[4] on the set of all ground query terms called the *simulation order*.

Note that the simulation order is not antisymmetric (e.g. $f\{a,b\} \preceq f\{b,a\}$ and $f\{b,a\} \preceq f\{a,b\}$, but $f\{a,b\} \neq f\{b,a\}$) and thus does not immediately provide a partial ordering. We therefore define an equivalence relation as follows:

Definition 30 (Simulation Equivalence)
Two ground query terms t_1 and t_2 are said to be *simulation equivalent*, denoted $t_1 \cong t_2$, if $t_1 \preceq t_2$ and $t_2 \preceq t_1$.

The meaning of simulation equivalence is rather intuitive: two terms are considered to be equivalent, if they differ only “insignificantly”, e.g. in a different order in the sequence of subterms in unordered term specifications (e.g. $f\{a,b\}$ and $f\{b,a\}$). This is consistent with the intuitive notion of unordered term specifications given above. Note, however, that $f\{a,a\} \ncong f\{a\}$, because the first term contains two a subterms, whereas the second contains only one a subterm, i.e. there cannot exist an index bijective mapping of the successors of the first into the successors of the second term (and vice versa). Simulation equivalence plays an important role later, because it allows to consider terms as “equal” that behave equally.

Simulation equivalence extends to non-ground terms in a straightforward manner: two non-ground query terms t_1 and t_2 are simulation equivalent, if for every grounding substitution σ holds that $\sigma(t_1) \cong \sigma(t_2)$. Note that for any two data terms t_1 and t_2 it holds that if $t_1 \preceq t_2$ then $t_1 \cong t_2$, because data terms do not contain partial term specifications.

Note that simulation equivalence is similar, but not equal to, bisimulation, because bisimulation requires the *same* relation to be a simulation in both directions, whereas simulation equivalence allows two different relations.

$\cong$ partitions T^g into a set of equivalence classes $T^g/_{\cong}$. On this set, $\preceq$ is a partial ordering. Given two equivalence classes $\tilde{t}_1 \in T^g/_{\cong}$ and $\tilde{t}_2 \in T^g/_{\cong}$, we shall write $\tilde{t}_1 \preceq \tilde{t}_2$ iff $t_1 \preceq t_2$.

[4] a preorder is defined as a transitive, reflexive relation

Corollary 31
$\preceq$ is a partial ordering on $T^g/_{\cong}$.

In this partial ordering, it even holds that given two terms t_1 and t_2 such that there exists a least upper bound t_3, then t_3 is unique except for terms t_3' that are equivalent wrt. $\cong$.

5.4 Simulation Unifiers

In Classical Logic, a unifier is a substitution for two terms t_1 and t_2 that, applied to t_1 and t_2, makes the two terms identical. The *simulation unifiers* introduced here follow this basic scheme, with two extensions: instead of equality, simulation unifiers are based on the (asymmetric) simulation relation of Section 5 and instead of a single substitution, substitution sets are considered. Both extensions are necessary for handling the special Xcerpt constructs *all* and *some* and incomplete term specifications.

Informally, a *simulation unifier* for a query term t^q and a construct term t^c is a set of substitutions Σ, such that each ground instance $t^{q'}$ of t^q in Σ simulates into a ground instance $t^{c'}$ of t^c in Σ. This restriction is too weak for fully describing the semantics of the evaluation algorithm. For example, consider a substitution set $\Sigma = \big\{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto b, Y \mapsto a\}$, a query term $t^q = f\{var\ X\}$ and a construct term $t^c = f\{var\ Y\}$. With the informal description above, Σ would be a simulation unifier of t^q in t^c, but this is not reasonable. We therefore also require that the substitution $\sigma \in \Sigma$ that yields $t^{q'}$ also is "used" by $t^{c'}$. This can be expressed by grouping the substitutions according to the free variables in t^c (cf. Definition 16 on page 16).

Definition 32 (Simulation Unifier)
Let t^q be a query term, let t^c be a construct term with the set of free variables $FV(t^c)$, and let Σ be an all-grounding substitution set. Σ is called a *simulation unifier* of t^q in t^c, if for each $[\![\sigma]\!] \in \Sigma/_{\simeq_{FV(t^c)}}$ holds that

$$\forall t^{q'} \in [\![\sigma]\!](t^q) \quad t^{q'} \preceq [\![\sigma]\!](t^c)$$

Recall from Section 4 that all substitutions in an all-grounding substitution set assign data terms to each variable. Intuitively, it is sufficient to only consider grounding substitutions for t^q and t^c. However, all-grounding substitution sets simplify the formalisation of most general simulation unifiers below.

Example 33 (Simulation Unifiers)

1. Let $t^q = f\{\{var\ X, b\}\}$ and let $t^c = f\{a, var\ Y, c\}$. A simulation unifier of t^q in t^c is the (all-grounding) substitution set

$$\Sigma_1 = \big\{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto c, Y \mapsto b\}\big\}$$

2. Let $t^q = f\{\{var\ X\}\}$ and let $t^c = f\{all\ var\ Y\}$. A simulation unifier of t^q in t^c is the (all-grounding) substitution set

$$\Sigma_2 = \big\{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto a, Y \mapsto a\}\big\}$$

Assignments for variables not occurring in the terms t^q and t^c are not given in the substitutions above.

Simulation unifiers are required to be *grounding* substitution sets, because otherwise the simulation relation cannot be established. Also, only grounding substitution sets can be applied to construct terms containing grouping constructs, because a grouping is not possible otherwise. This restriction is less significant than it might appear: as rules in Xcerpt are range restricted, the evaluation algorithm always determines bindings for the variables in t^c, so that it is always possible to extend the solutions determined by the simulation unification algorithm to a grounding substitution set by merging with these bindings.

Usually, there are infinitely many unifiers for a query term and a construct term. Traditional logic programming therefore considers the most general unifier (mgu), i.e. the unifier that subsumes all other unifiers. Since simulation unifiers are always grounding substitution sets, such a definition is not possible for simulation unifiers. Instead, we define the *most general simulation unifier* (mgsu) as the smallest superset of all other simulation unifiers. Note that the notion *most general simulation unifier* is – although different in presentation – indeed similar to the traditional notion of most general unifiers, because a most general simulation unifier subsumes all other simulation unifiers.

Definition 34 (Most General Simulation Unifier)
Let t^q be a query term and let t^c be a construct term without grouping constructs such that there exists at least one simulation unifier of t^q in t^c. The *most general simulation unifier* (mgsu) of t^q in t^c is defined as the union of all simulation unifiers of t^q in t^c.

Note that the most general simulation unifier is indeed always a simulation unifier if t^c does not contain grouping constructs. This is easy to see because the union of two simulation unifiers simply adds ground instances of t^q and t^c where for every ground instance $t^{q'}$ of t^q there exists a ground instance $t^{c'}$ of t^c such that $t^{q'} \preceq t^{c'}$. This does in general not hold for construct terms with grouping.

6 Interpretations and Entailment

The definition of satisfaction of Xcerpt term formulas, and in particular of Xcerpt programs, is similar to the approach taken in classical first order logic, but differs in several important aspects: term formulas do not differentiate between relations and terms, and the incompleteness of query terms and the grouping constructs in construct terms have to be taken into account. Section 6.1 gives an intuitive meaning of interpretations for Xcerpt term formulas. Satisfaction is then defined in Section 6.2 in terms of the *simulation relation* introduced earlier in Section 5. Based on this definition of satisfaction, entailment between formulas can be defined in the classical manner.

6.1 Interpretations

As terms are considered to be formulas themselves, interpretations – informally – convey whether "a term exists" or "a term does not exist". Thus, a first approximation defines an interpretation as a set of data terms (which are also ground query terms). A ground atom (i.e. a ground query term) is then satisfied if it is contained in the set, or it simulates into a term that is contained in the set. Since Xcerpt data terms represent Web pages, this definition is natural and close to the application, and thus well suited for reasoning on the Web. Such a definition may be unusual from a Classical Logic perspective, but is rather common in logic programming for it is close to Herbrand interpretations.

Furthermore, an interpretation provides a grounding substitution set which provides assignments to all free variables in the formulas considered. Interpretations are thus formally defined as follows:

Definition 35 (Interpretation)
An *interpretation* M is a tuple $M = (I, \Sigma)$ where I is a set of data terms and $\Sigma \neq \emptyset$ is a grounding substitution set.

The set of data terms I conveys what data terms (Web pages) are considered to exist. The substitution set Σ is necessary to properly treat formulas containing free variables, and allows to provide a recursive definition of satisfaction below. As formulas are usually always (explicitly or implicitly) universally closed, Σ can be seen as a mere technicality of the definition and is irrelevant for the general notion of satisfaction. For this reason, the following Sections often somewhat imprecisely equate interpretations with the set of data terms I.

Note that $\Sigma \neq \emptyset$. Otherwise, $\Sigma(t)$ would yield an empty set of terms even in case t is a ground query term. As the application of a substitution set to a query term formula yields a conjunction over all substitutions, application of $\emptyset$ would yield an empty conjunction, i.e. $\top$. To define a substitution set that merely maps each term to itself it has to be specified as $\Sigma = \left\{ \emptyset \right\}$, where the empty substitution σ corresponds to the identity function.

It is important to note that the interpretations considered here are very specific in that they only consider *terms* as objects, instead of arbitrary objects. They are thus similar to Herbrand interpretations in traditional model theory. However, this restriction is reasonable, as term formulas do not intend to represent arbitrary objects.

6.2 Satisfaction and Models

Although similar to the definition of satisfaction in classical logic, satisfaction for Xcerpt term formulas differs in several important aspects, in particular the satisfaction of atoms (i.e. terms) and of program rules. A term (atomic formula) is considered to be satisfied if (and only if) its ground instance simulates in some term of the interpretation. Considering the Web as an interpretation, this means that a query term "succeeds" (is satisfied) if there exists a Web page (data term) such that the ground instance of the query term simulates into this data term.

Unlike in traditional logic programs, rules in Xcerpt are not treated as (classical) implications ($\Rightarrow$ below), because the grouping constructs `all` and `some` require that the query part of a rule is not only satisfied, but that it is also satisfied in the maximal manner, i.e. the substitution set yielding the ground instance of the construct term must include all possible substitutions for which the query part is satisfied. Otherwise, interpretations would include answer terms for a rule that differ from the intuitive understanding of the constructs `all` and `some` (see Example 38 below). The definition of satisfaction for Xcerpt rules uses the notion of maximal substitution sets defined above in Definition 11.

With the exception of term and rule satisfaction, the following definition follows the classical definition of satisfaction. Note in particular, that the negation used in this definition is *classical* negation and not negation as failure (as the query negation in Xcerpt programs).

Definition 36 (Satisfaction, Model)

1. Let $M = (I, \Sigma)$ be an interpretation (i.e. a set of data terms I and a substitution set Σ), and let t be a construct or query term.

   The satisfaction of a term formula F in M, denoted by $M \models F$, is defined recursively over the structure of F:

$$\begin{array}{lll}
M \models \top & & \text{holds} \\
M \models \bot & & \text{does not hold} \\
M \models t & \text{iff} & \text{for all } t' \in \Sigma(t) \text{ there exists a term } t^d \in I \text{ such that } t' \preceq t^d \\
M \models \neg F & \text{iff} & M \not\models F \\
M \models F_1 \wedge \cdots \wedge F_n & \text{iff} & M \models F_1 \text{ and} \ldots \text{and } M \models F_n \\
M \models F_1 \vee \cdots \vee F_n & \text{iff} & M \models F_1 \text{ or} \ldots \text{or } M \models F_n \\
M \models F \Rightarrow G & \text{iff} & M \models \neg F \vee G \\
M \models \forall x.F & \text{iff} & \text{for all } t \in I \text{ holds that } M' = (I,\Sigma') \models F, \\
 & & \text{where } \Sigma' = \big\{\sigma \circ \{x \mapsto t\} \mid \sigma \in \Sigma\big\} \\
M \models \exists x.F & \text{iff} & \text{there exists a } t \in I \text{ such that } M' = (I,\Sigma') \models F, \\
 & & \text{where } \Sigma' = \big\{\sigma \circ \{x \mapsto t\} \mid \sigma \in \Sigma\big\} \\
M \models \forall^* \ll t^c \leftarrow Q \gg & \text{iff} & M' = (I,\Sigma') \models t^c \text{ for a maximal grounding substitution set } \Sigma' \text{ for } Q \\
 & & \text{with } M' \models Q
\end{array}$$

2. If a formula F is satisfied in an interpretation $\mathscr{M}$, i.e. $\mathscr{M} \models F$, then $\mathscr{M}$ is called a *model* of F.

Note that the maximality requirement in the last part of (1) refers to the satisfaction of Q in M and ensures that grouping constructs in the head of the rule are substituted properly.

As instances of Xcerpt rules are variable disjoint (so-called *standardisation apart*), it is possible to replace Σ by Σ' in the model definition for $\forall^* \ll t \leftarrow Q \gg$. Otherwise, the substitutions in Σ and Σ' would have to be merged to $\Sigma \circ \Sigma'$.

Example 37 (Satisfaction of Term Formulas)
Let $M = (I,\Sigma)$ be an interpretation with

$$\begin{array}{lll}
I & := & \big\{f[a,b], f[a,c], b\big\} \\
\Sigma & := & \big\{\{X \mapsto a,\ Y \mapsto b\}, \{X \mapsto a,\ Y \mapsto c\}\big\}
\end{array}$$

The following statements hold for M:

1. $M \models f[a,b]$, because for each $t \in \Sigma(f[a,b]) = \big\{f[a,b]\big\}$ exists a $t' \in I$ with $t \preceq t'$

2. $M \not\models f[a,d]$, because for $t = f[a,d] \in \Sigma(f[a,d]) = \big\{f[a,d]\big\}$ does not exist a $t' \in I$ with $t \preceq t'$.

3. $M \models f\{a,b\}$, because for each $t \in \Sigma(f\{a,b\}) = \big\{f\{a,b\}\big\}$ exists a $t' \in I$ with $t \preceq t'$

4. $M \models f\{\{var\ X, var\ Y\}\}$, because

   - $\sigma_1 = \{X \mapsto a,\ Y \mapsto b\}$ and $\sigma_1(f\{\{var\ X, var\ Y\}\}) \preceq f[a,b]$, and
   - $\sigma_2 = \{X \mapsto a,\ Y \mapsto c\}$ and $\sigma_2(f\{\{var\ X, var\ Y\}\}) \preceq f[a,c]$

5. $M \models \exists Z. f\{\{var\ Z\}\}$, because $M' = (I,\Sigma')$ with
   $\Sigma' = \big\{\{X \mapsto a,\ Y \mapsto b,\ Z \mapsto a\}, \{X \mapsto a,\ Y \mapsto c,\ Z \mapsto a\}\big\}$
   is a model for $f\{\{var\ Z\}\}$

6. $M \not\models \forall Z. f\{\{var\ Z\}\}$, because there exists a term $f[a,b]$ as substitution for Z such that $M \not\models f\{\{f[a,b]\}\}$

7. $M \models \forall Z.var\ Z$, because for all $t \in I$ holds that $M' = (I,\Sigma')$ with $\Sigma' = \big\{\{X \mapsto a,\ Y \mapsto b,\ Z \mapsto t\}, \{X \mapsto a,\ Y \mapsto c,\ Z \mapsto t\}\big\}$
   is a model for $var\ Z$[5]

[5] This result might be surprising from a classical perspective, but it is self-evident when considering terms as formulas: universal quantification quantifies over all existing terms, and obviously all these are satisfied in any interpretation.

For a program P, a model is intuitively an interpretation that contains all the data terms that are “produced” by P (and possibly also further data terms unrelated to P).

Example 38 (Satisfaction of Xcerpt Programs)
Let P be the following Xcerpt program (in compact notation):

```
p{all var X} ← q{{var X}}
q{a,b,c}
```

- the interpretation $M_1 = (I_1, \{\emptyset\})$ with $I_1 = \big\{q\{a,b,c\}, p\{a,b,c\}\big\}$ is a model for P, i.e. $M_1 \models P$.
- the interpretation $M_2 = (I_2, \{\emptyset\})$ with $I_1 = \big\{q\{a,b,c\}, p\{a,b\}\big\}$ is no model for P, i.e. $M_1 \not\models P$, because $p\{a,b\}$ is not the ground instance of $p\{all\ var\ X\}$ by the *maximal* substitution set for which $q\{\{var\ X\}\}$ is satisfied
- the interpretation $M_3 = (I_3, \{\emptyset\})$ with $I_3 = \big\{q\{a,b,c\}, p\{a,b,c\}, p\{a,b\}\big\}$ is a model for P, i.e. $M_3 \models P$, because $p\{a,b,c\} \in I$; the additional $p\{a,b\}$ is not produced by P, but irrelevant for the satisfaction of P in M_3.

Note that “terms” with infinite breadth are precluded by the definition of terms and can thus never appear in an interpretation. Programs where a rule “defines” such terms do not have a model. For example, the program

$$\begin{array}{l} f\{all\ var\ X\} \leftarrow g\{var\ X\} \\ g\{g\{var\ Y\}\} \leftarrow g\{var\ Y\} \\ g\{a\} \end{array}$$

does not have a model, because the first rule defines a “term” of the form $f\{a, g\{a\}, g\{g\{a\}\}, \ldots\}$. To avoid non-terminating evaluation of such programs, it is desirable to find sufficient requirements to preclude such programs syntactically. This is however out of the scope of this article.

7 Fixpoint Semantics

A classical approach to describing the semantics of logic programs is the so-called *fixpoint semantics*, first proposed by Van Emden and Kowalski [vEK76]. In the fixpoint semantics, a model is constructed by iteratively trying to apply program rules (using an operator called T_P) to a set of data terms and adding their results until a fixpoint is reached, i.e. no new data terms can be added. This smallest fixpoint is then a model of the program (assuming that programs do not contain negation).

Example 39
Consider again the program

$$\begin{array}{l} f\{all\ var\ X\} \leftarrow g\{\{var\ X\}\} \\ g\{a\} \end{array}$$

By definition, the starting point is always $I_0 = \emptyset$. In the first iteration, no rules are applicable, but the data terms are added to the set. Thus,

$$I_1 = T_P(\emptyset) = \big\{g\{a\}\big\}$$

The next iteration allows to apply the program rule. Thus,

$$I_2 = T_P(I_1) = \big\{g\{a\}, f\{a\}\big\}$$

Further application of rules does not add new terms, thus I_2 is the smallest fixpoint. It is easy to see that I_2 is also a "reasonable" model of the program. Note that there are other fixpoints besides I_2, e.g. $\{g\{a\}, f\{a\}, f\{b\}\}$, all of them supersets of I_2.

The following section proposes a fixpoint semantics for Xcerpt programs with grouping constructs but without negation, and shows that the fixpoint of the program is also a model of a program. Since the fixpoint semantics is the most precise characterisation of Xcerpt programs available, it is also used as the reference for the verification of the backward chaining algorithm. Programs with negation are not considered in this article, but their treatment should be very similar to the treatment of negation in other logic programming languages. Since Xcerpt programs are negation stratifiable, a similar approach to the approach taken by Apt, Blair, and Walker [ABW88] appears promising.

This article slightly diverges from the traditional definition of the fixpoint operator T_P in that it defines T_P as a function whose result contains not only the new terms but also those given as argument. Thus, it is sufficient to simply let T_P saturate in iterative applications instead of using a complex notion of powers of the form $T_P \uparrow n$. Arguably, this approach is more straightforward, because it reflects the intuitive understanding of program evaluation.

Recall that ω denotes the first ordinal number, i.e. the smallest number that is larger than any natural number. Thus, T_P^ω denotes the application of T_P "until a fixpoint is reached" (whether it be finite or infinite). The fixpoint operator is defined as follows:

Definition 40 (Fixpoint Operator T_P, Fixpoint Interpretation)
Let P be an Xcerpt program.

1. The fixpoint operator T_P is defined as follows:

$$T_P(I) = I \cup \Big\{ t^d \mid \begin{array}{l} \text{there exists a rule } t^c \leftarrow Q \text{ in } P \text{ and substitution set } \Sigma \\ \text{such that } \Sigma \text{ is the maximal set with } (I,\Sigma) \models Q \text{ and } t^d \in \Sigma(t^c), \\ \text{or } t^d \text{ is a data term in } P \end{array} \Big\}$$

2. The fixpoint of T_P is denoted by $M_P = T_P^\omega(\emptyset)$ and called the fixpoint interpretation of P.

A problem with this first definition is that it can yield interpretations that contain unjustified terms in case the program contains grouping constructs, because rules with grouping constructs require the rule body to be satisfied maximally, but not all required information might be available in the iteration of T_P where the rule is applied.

Example 41
Consider the following Xcerpt program:

$$\begin{array}{l} f\{all\ var\ X\} \leftarrow g\{\{var\ X\}\} \\ g\{var\ Y\} \leftarrow h\{\{var\ Y\}\} \\ g\{a\} \\ h\{b\} \end{array}$$

Applying the fixpoint operator T_P yields the following results:

$$\begin{array}{rcrcl} & & T_P^1(\emptyset) & = & \{g\{a\}, h\{b\}\} \\ & & T_P^2(\emptyset) & = & \{g\{a\}, h\{b\}, g\{b\}, f\{a\}\} \\ M_P & = & T_P^3(\emptyset) & = & \{g\{a\}, h\{b\}, g\{b\}, f\{a\}, f\{a,b\}\} \end{array}$$

However, $f\{a\}$ should not occur, because it is not the result of the maximal substitution for $g\{\{var\ X\}\}$. Obviously, applying the first rule already in T_P^2 is too early.

Therefore, we refine the notion of fixpoint interpretations to fixpoint interpretations for stratifiable programs. Constructing fixpoints for Xcerpt programs containing grouping constructs is based on the grouping stratification of such programs and simply applies the fixpoint operator stratum by stratum, beginning with the lowest stratum and ending with the highest. The following definition follows closely a definition by Apt, Blair, and Walker [ABW88]:

Definition 42 (Fixpoint Interpretation for Stratifiable Programs)
Let P be a program with grouping stratification $P = P_1 \uplus \cdots \uplus P_n$ ($n \geq 1$). The fixpoint interpretation M_P is defined by

$$\begin{array}{rcl} M_1 & = & T^{\omega}_{P_1}(\emptyset) \\ M_2 & = & T^{\omega}_{P_2}(M_1) \\ \vdots & & \\ M_n & = & T^{\omega}_{P_n}(M_{n-1}) \end{array}$$

with $M_P = M_n$.

Note that this definition of M_P is in principle applicable to all kinds of stratification, i.e. grouping stratification, negation stratification, and full stratification.

Example 43
Consider the following Xcerpt program stratifiable into two strata P_1 and P_2:

P_2	$f\{all\ var\ X\} \leftarrow g\{\{var\ X\}\}$
P_1	$g\{var\ Y\} \leftarrow h\{\{var\ Y\}\}$
	$g\{a\}$
	$h\{b\}$

Applying the fixpoint operator T_{P_1} for the stratum P_1 yields the following sets:

$$\begin{array}{rcrcl} & & T^1_{P_1}(\emptyset) & = & \{g\{a\}, h\{b\}\} \\ M_1 & = & T^2_{P_1}(\emptyset) & = & \{g\{a\}, h\{b\}, g\{b\}\} \end{array}$$

$M_1 = T^2_{P_1}$ is a fixpoint for this stratum. Further application of the fixpoint operator T_{P_2} for the stratum P_2 to this set then results in:

$$M_2 \quad = \quad T^1_{P_2}(M_1) \quad = \quad \{g\{a\}, h\{b\}, g\{b\}, f\{a,b\}\}$$

it is easy to see that $M_2 = T^1_{P_2}(M_1)$ is a model of P, and that M_2 does not contain unjustified terms.

We now show that the fixpoint of a program is also a model. Note, however, that the inverse statement does not hold:

Theorem 44
Let P be a grouping stratified program without negation. Then the fixpoint M_P of P is a model of P.

Proof. Suppose M_P is not a model of P. Then there exists a term t not in M_P that is required by M_P and P. There are two cases for this:

- t is a data term in P. By definition of T_P, t is then in M_P. $\lightning$
- t is a ground instance of a rule in P, i.e. there exists a rule $t^c \leftarrow Q$ in P and a substitution set Σ that is a maximal substitution with $M_P \models \Sigma(Q)$ such that $t \in \Sigma(t^c)$. By definition of T_P, it holds that $\Sigma(t^c) \subseteq M_P$. $\lightning$

8 Outlook and Future Work

The semantics described in this article is unsatisfactory in that it only covers a limited set of Xcerpt programs (namely those that are grouping stratifiable), does not cover negation (as failure), and does not provide a theory of minimal model as is usually done in traditional logic programming. The following sections briefly suggest refinements that might be addressed in future work.

8.1 Semantics of Advanced Xcerpt Constructs

Some more advanced Xcerpt constructs are not covered by the model-theoretic semantics described in this article. This section gives a brief outline over possible approaches for these constructs. More elaborated proposals can be found in [Sch04].

Arithmetic Expressions and Aggregation Functions. Xcerpt construct terms may contain arithmetic expressions (like `+`, `-`, string concatenation, etc.) and aggregation functions (like `count`, `sum`, etc., usually in conjunction with grouping constructs). In general, both arithmetic expressions and aggregation functions are applied to a number of data terms (i.e. ground construct terms) and yield a new data term (for example, `sum` can be applied to the three data terms `3`, `4`, and `5`, and yields the data term `12`). Extending the model-theoretic semantics to convey their meaning can be achieved by a simple modification of the *application of substitution sets to construct terms* (cf. Section 4.2). Expressions might e.g. be evaluated on the ground instances that are the result of applying a substitution set to a construct term.

Optional Subterms. Xcerpt query and construct terms may contain so-called *optional* subterms preceded by the keyword `optional`. Intuitively, optional subterms have the following meaning:

- *query terms* containing optional subterms may match with data terms even if there exists no corresponding subterm in the data term, i.e. matching does not fail in this case (but does not yield variable bindings). On the other hand, if the data term does contain at least one corresponding subterm, optional subterms are required to match (and possibly yield variable bindings). The semantics of optional subterms in a query term can be formalised by properly adapting the notion of *ground query term simulation* (cf. Section 5). To reflect that optional subterms are required to match if possible, it is furthermore necessary to allow only those substitutions as valid answers for a query term and a data term that provide bindings for a maximal subset of variables.

- optional subterms in *construct terms* may be omitted if a substitution or substitution set does not provide bindings for at least one of the variables contained in the optional subterm (such "incomplete" substitutions might be the result of optional subterms in the query part of a rule). The sematics of optional subterms in a construct term can be formalised by extending the definition of *application of substitution sets to construct terms* (cf. Section 4.2).

Subterm Negation. In query terms, subterm negation (using the keyword `without`) denotes that matching data terms may not contain corresponding subterms that are matched by the negated subterm. For example, `f{{without b}}` matches only with data terms that have a root with label `f` and arbitrary subterms except for such that are matched by *b*. Thus, the data term `f{a,c}` would match, whereas the data term `f{a,b}` would not.

The semantics of subterm negation is best integrated into the *ground query term simulation* defined in Section 5. A first approach following this idea is described in [Sch04].

8.2 (Non-)Monotonicity: Negation and Grouping Constructs

Requiring grouping/negation stratification as in this article is too strict for many applications. Therefore, it would be worthwhile to investigate relaxations of these requirements (like *local stratification* [Prz88]) or even entirely different approaches that have been proposed in the last 20 years (like *stable models* [GL88] or *paraconsistent interpretations* [Bry02]) to non-monotonic constructs in Xcerpt.

8.3 Minimal Models

In traditional logic programming, the fixpoint of a program coincides with its *minimal model*, which is simply the intersection of all models of the program. It is easy to see that this approach is not feasible in the presence of grouping constructs like in Xcerpt. Consider the following simple program P consisting of a single rule and a single data term:

```
CONSTRUCT
  f{all var X}
FROM
  g{var X}
END

CONSTRUCT
  g{a}
END
```

Models for this program are e.g.

- $I_1 = \{g\{a\}, f\{a\}\}$
- $I_2 = \{g\{a\}, g\{b\}, f\{a, b\}\}$
- $I_3 = \{g\{a\}, g\{b\}, g\{c\}, f\{a, b, c\}\}$

Obviously, I_1 is the only "desirable" model, and also the fixpoint of P, i.e. $I_1 = T_P^{\omega}(\emptyset)$. It is easy to see that the intersection of e.g. I_1 and I_2 is not a model of P, i.e. the minimal model cannot be determined by simple set intersections.

Approaches to this problem could redefine intersection to "look inside terms". In the above example, a solution could be to not only do set intersection but also "term intersection". Thus, the intersection of $f\{a, b\}$ and $f\{a\}$ would be $f\{a\}$. However, several further problems arise with this kind of definition: it is unclear which terms to intersect, one cannot rely on known properties of set operations (if intersection is redefined, how about union?), and the resulting minimal model semantics is no longer as "declarative" as would be desirable.

Regardless of the approach taken, the minimal model semantics needs to be simple, because it is intended to *describe* the meaning of a program without relying on its operational behaviour; if no reasonable, understandable minimal model semantics can be found, it would probably be preferrable to be stick to the operational description given in form of the fixpoint semantics.

Acknowledgements.

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

References

[ABW88] Krzysztof Apt, Howard Blair, and Adrian Walker. Towards a Theory of Deductive Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann, 1988.

[Bry02] Franois Bry. An Almost Classical Logic for Logic Programming and Nonmonotonic Reasoning. In *Proceedings of Workshop on Paraconsistent Computational Logic (PCL'2002)*, Copenhagen, Denmark, July 2002.

[BS02] François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of the International Conference on Logic Programming (ICLP'02)*, LNCS 2401, Copenhagen, Denmark, July 2002. Springer-Verlag.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[HHK96] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing Simulations on Finite and Infinite Graphs. Technical report, Computer Science Department, Cornell University, July 1996.

[Kil92] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, Dept. of Computer Sciences, University of Helsinki, November 1992.

[Mil71] Robin Milner. An Algebraic Definition of Simulation between Programs. Technical Report CS-205, Computer Science Department, Stanford University, 1971. Stanford Aritifical Intelligence Project, Memo AIM-142.

[Prz88] Teodor Przymusinsik. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann, 1988.

[SB04] Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Extreme Markup Languages 2004*, Montral, Canada, August 2004. IDEAlliance. `http://www.extrememarkup.com/extreme/2004/`.

[Sch04] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Institute for Informatics, University of Munich, October 2004.

[vEK76] M. H. van Emden and R. Kowalski. The Semantics of Logic as a Programming Language. *Journal of the ACM*, 3:733–742, 1976.