I4-D5

Simulation Unification

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-D5/D/PU/a1
Responsible editors:	Sebastian Schaffert
Reviewers:	Norbert Eisinger and Claude Kirchner
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	31 August 2005
Actual submission date:	30.12.2005

Abstract

Simulation unification is a novel kind of (non-standard) unification that allows to treat the particularities of Xcerpt terms properly and is based on the notions of ground query term simulation and answers (cf. Section 3). It has first been proposed in [BS02] and is further refined here. Simulation Unification is an algorithm that, given two terms t_1 and t_2, determines variable substitutions such that the ground instances of t_1 and t_2 simulate. Like standard unification (cf. [Rob65]), simulation unification is *symmetric* in the sense that it can determine (partial) bindings for variables in both terms. Unlike standard unification, it is however *asymmetric* in the sense that it does not make the two terms equal, but instead ensures a ground query term simulation, which is directed and asymmetric. The outcome of Simulation Unification is a set of substitutions called *simulation unifier*.

Keyword List

query language, Semantic Web, simulation, unification, backward chaining

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

Simulation Unification

Sebastian Schaffert, François Bry, Tim Furche

Institute for Informatics, University of Munich, Germany
Email: Sebastian.Schaffert@ifi.lmu.de

30.12.2005

Abstract
Simulation unification is a novel kind of (non-standard) unification that allows to treat the particularities of Xcerpt terms properly and is based on the notions of ground query term simulation and answers (cf. Section 3). It has first been proposed in [BS02] and is further refined here. Simulation Unification is an algorithm that, given two terms t_1 and t_2, determines variable substitutions such that the ground instances of t_1 and t_2 simulate. Like standard unification (cf. [Rob65]), simulation unification is *symmetric* in the sense that it can determine (partial) bindings for variables in both terms. Unlike standard unification, it is however *asymmetric* in the sense that it does not make the two terms equal, but instead ensures a ground query term simulation, which is directed and asymmetric. The outcome of Simulation Unification is a set of substitutions called *simulation unifier*.

Keyword List
query language, Semantic Web, simulation, unification, backward chaining

Contents

1 Introduction

Simulation unification is a novel kind of (non-standard) unification that allows to treat the particularities of Xcerpt terms properly and is based on the notions of ground query term simulation and answers (cf. Section 3). It has first been proposed in [BS02] and is further refined here. Simulation Unification is an algorithm that, given two terms t_1 and t_2, determines variable substitutions such that the ground instances of t_1 and t_2 simulate. Like standard unification (cf. [Rob65]), simulation unification is *symmetric* in the sense that it can determine (partial) bindings for variables in both terms. Unlike standard unification, it is however *asymmetric* in the sense that it does not make the two terms equal, but instead ensures a ground query term simulation, which is directed and asymmetric. The outcome of Simulation Unification is a set of substitutions called *simulation unifier*.

Section 4 discusses simulation unification in the context of the Xcerpt constraint solver. Section 2 briefly recapitulates the language Xcerpt and introduces several formalisms and denotations used in the remainder of this article.

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 following page gives an impression of the Xcerpt term syntax.

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" }
    },
  ...
  },
...
}
```

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 3). 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 `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. [SBF05]). 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 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 2.2. Substitution sets are then used in the declarative semantics (cf. [SBF05]) 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).

2.2.1 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 $\{X \mapsto a, Y \mapsto b\}$ 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 2.3 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 2.3). 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 $\{X \mapsto f\{a\}\}$ is a substitution for *var* $X \rightsquigarrow f\{\{\}\}$, but not for *var* $X \rightsquigarrow g\{\{\}\}$. 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). 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.

2.2.2 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 2.3 and 2.4). 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

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

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

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 $\{\{X \mapsto a\}\}$ to the set of variables $\{X,Y\}$ is the (infinite) set $\{\{X \mapsto a, Y \mapsto a\}, \{X \mapsto a, Y \mapsto b\}, \ldots\}$

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.

2.2.3 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 5 (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. $\Sigma_{|V} \subset \Phi_{|V}$).

2.3 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 6 (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 $(X \mapsto t') \in \sigma$
   - $\sigma(var\ X \rightsquigarrow s) = t'$ 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 $var\ X \rightsquigarrow s$, 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 $\Sigma = \{\emptyset\}$.

2.4 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 $FV(t^c)$. Recall also that $\cong$ denotes simulation equivalence between two ground terms.

Definition 7 (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 $/_{\backsim_V}$ with respect to $\simeq_V$ is called the *grouping of* Σ *on* V.
- Each of the equivalence classes $\in /_{\backsim_V}$ is accordingly defined as $= \{ \tau \in \Sigma \mid \tau \simeq_V \sigma \}$.

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

Example 8
Given the substitution set $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ 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

- $_1 = _2 = \{\{X_1 \mapsto a, X_2 \mapsto b\}, \{X_1 \mapsto a, X_2 \mapsto c\}\}$
- $_3 = \{\{X_1 \mapsto c, X_2 \mapsto b\}\}$

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 $|/_{\backsim_{FV(t^c)}}|$ results, one for each different binding of the free variables in t^c.

Example 9
Given a term $t = f\{X_1, g\{all\ X_2\}\}$, i.e. $FV(t) = \{X_1\}$. Consider again

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

from Example 8. The result of applying Σ to t is

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

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 $/_{\backsim_{FV(t^c)}}$ 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 10, 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 [SBF05]. 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.

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 10 (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. $/_{\lfloor FV(t^c)} = \{\}$. The restricted application of Σ to t^c, written (t^c), is recursively defined as follows:

   - $(var\ V) = \langle \sigma(V) \rangle$[2]
   - $(f\{t_1, \ldots, t_n\}) = \langle (f)\{(t_1) \circ \cdots \circ (t_n)\} \rangle$ for some $n \geq 0$
   - $(f[t_1, \ldots, t_n]) = \langle (f)[(t_1) \circ \cdots \circ (t_n)] \rangle$ for some $n \geq 0$
   - $(all\ t) = {}_1(t) \circ \cdots \circ {}_k(t)$ where $\{{}_1, \ldots, {}_k\} = /_{\simeq_{FV(t)}}$
   - $(all\ t\ group\ by\ V) = {}_1(t) \circ \cdots \circ {}_k(t)$ where $\{{}_1, \ldots, {}_k\} = /_{\simeq_{FV(t) \cup V}}$
   - $(all\ t\ order\ by\ f\ V) = {}_1(t) \circ \cdots \circ {}_k(t)$ where $\langle {}_1, \ldots, {}_k \rangle = sort(f(V), /_{\simeq_{FV(t) \cup V}})$
   - $(some\ k\ t) = {}_1(t) \circ \cdots \circ {}_k(t)$ where $\{{}_1, \ldots, {}_k\} \subseteq /_{\simeq_{FV(t)}}$
   - $(some\ k\ t\ group\ by\ V) = {}_1(t) \circ \cdots \circ {}_k(t)$ where $\{{}_1, \ldots, {}_k\} \subseteq /_{\simeq_{FV(t) \cup V}}$
   - $(some\ k\ t\ order\ by\ f\ V) = {}_1(t) \circ \cdots \circ {}_k(t)$ where $\langle {}_1, \ldots, {}_k \rangle \sqsubseteq sort(f(V), /_{\simeq_{FV(t) \cup V}})$
   - $(optional\ t) = \begin{cases} (t) & \text{if the ground instance } (t) \text{ exists} \\ \langle\rangle & \text{otherwise} \end{cases}$
   - $(optional\ t\ with\ default\ t') = \begin{cases} (t) & \text{if the ground instance } (t) \text{ exists} \\ (t') & \text{otherwise} \end{cases}$

   where $1, \ldots, k$ 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:

$$\Sigma(t) = \left\{ t^{c'} \mid \in /_{\lfloor FV(t^c)} \ \wedge\ \langle t^{c'} \rangle = (t^c) \right\}$$

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

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

[2] Note that σ is the representative of the equivalence class

Example 11
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

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

The ground instances of t_1 and t_2 by Σ are thus

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

2.5 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 12 (Substitutions: Application to Query Term Formulas)
Let F be a quantifier-free term formula where all atoms are query terms (a *query term formula*).

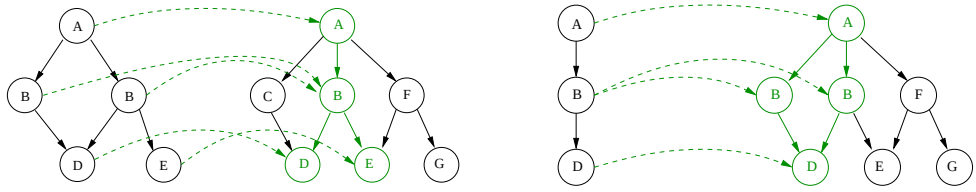
1. The application of a *substitution* σ to F, written $\sigma(F)$, is recursively defined as follows:
   - $\sigma(F_1 \wedge F_2) = \sigma(F_1) \wedge \sigma(F_2)$
   - $\sigma(F_1 \vee F_2) = \sigma(F_1) \vee \sigma(F_2)$
   - $\sigma(\neg F') = \neg\sigma(F')$
   - $\sigma(F') = \sigma(F')$
2. The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

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

3 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

Figure 1 Rooted Graph Simulations (with respect to vertex adornment equality)



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"{}`.

3.1 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 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 13 (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 $\sim$ if:

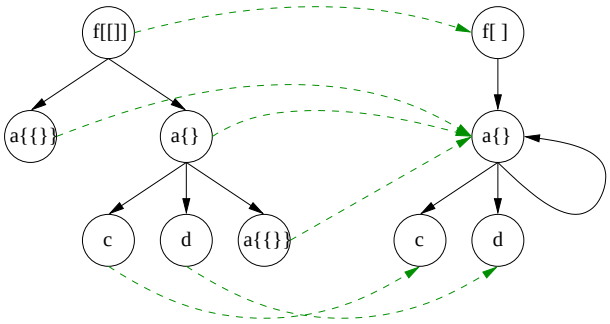
1. $r_1 \; S \; r_2$.
2. If $v_1 \; S \; v_2$, then $v_1 \sim v_2$.
3. If $v_1 \; S \; v_2$ and $(v_1, v_1', i) \in E_1$, then there exists $v_2' \in V_2$ such that $v_1' \; S \; v_2'$ and $(v_2, v_2', j) \in E_2$

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 13 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 1 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

Figure 2 Minimal simulation of $f[[\ a\{\{\ \}\}, a\{\{c, d, a\{\{\ \}\}\ \}\}\ ]]$ in $f[\&1\ @\ a\{c, d, \uparrow \&1\}]$



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".

3.2 Ground Query Term Simulation

Using the graphs induced by ground query terms, 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 2). 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).

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 14
Given two term sequences $M = \langle s_1, \ldots, s_m \rangle$ and $N = \langle t_1, \ldots, t_n \rangle$.

A partial or total mapping $\pi : M \to N$ 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. 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 15 (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 16

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]

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

Definition 17 (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:

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

A comprehensive list of examples is given in [Sch04].

3.3 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 3.2 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 $\Sigma = \{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto b, Y \mapsto a\}$, 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 7 on page 9).

Definition 18 (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 $\in /_{\downarrow_{FV(t^c)}}$ holds that

$$\forall t^{q'} \in (t^q) \quad t^{q'} \preceq (t^c)$$

Recall from Section 2.2 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 19 (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

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

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

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

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 20 (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.

4 A Constraint Solver for Language Evaluation

The evaluation of Xcerpt programs is described in terms of a constraint solver that applies so-called *simplification rules* to a constraint store consisting of conjunctions and disjunctions of constraints. The purpose of the constraint solver is to determine variable bindings for variables occurring in query and construct terms, which ultimately yield substitutions that can be used to create the answer terms of a program. A simplification rule in this thesis has the following form:

$$\frac{\begin{array}{c} C_1 \\ \vdots \\ C_n \end{array}}{D}$$

where $C_1, \ldots, C_n$ ($n \geq 1$) are atomic constraints (the condition) and D is either an atomic constraint, or a conjunction or disjunction of constraints (the consequence). If a simplification rule is applied, then the conjunction $C_1 \wedge \cdots \wedge C_n$ in the constraint store is replaced by the constraint D. Note that these simplification rules are similar to the simplification rules in the language *Constraint Handling Rules* [Frü95], albeit with a different notation.

The constraint solver is non-deterministic to a high degree in that the order in which simplification rules are applied is not significant. This approach might be advantageous, as it gives much freedom to the evaluation engine to e.g. perform optimisations.

This constraint solver differs from common approaches in that the result of a rule may contain disjunctions, whereas usually only conjunctions are admitted. Such constraint solvers have been studied in constraint programming research, e.g. in [WM96]. The approach taken in this thesis is rather simplistic, as it after each application of a simplification rule creates the disjunctive normal form (DNF) of the constraint store. Simplification rules are independently applied to the different conjuncts of the DNF. This approach is rather inefficient in implementations, and various optimisations can be considered. A straightforward optimisation would be to not create the DNF after *each* simplification step, but instead only if it is "necessary", because no other simplification rules apply. However, such optimisations are not further investigated in this thesis, as the focus is on Web query languages and not on constraint programming.

Furthermore, the constraint solver needs to be able to treat negation. As both negation constructs `not` and `without` describe negation as failure, the negation behaves differently to classic negation in some cases (cf. Example 26). The treatment of negation is described in the formula simplification rules in Section 4.3, and in the consistency verification rules 3, 4, and 5 in Section 4.4 below.

4.1 Data Structures and Functions

4.1.1 Constraints

The main data structure of the evaluation algorithm is the *constraint store* which may contain several types of constraints, including other (sub-)constraint stores. For the purpose of this thesis, constraints

are defined by the following grammar (defined in a variant of *Extended Backus-Naur Form*):

```
<constraint>       := <conjunction>    | <disjunction>
                    | 'True'           | 'False'
                    | '(' <constraint> ')'
                    | <sim-constraint>
                    | <dep-constraint>
                    | <query-constraint> .
<conjunction>      := <constraint> ('∧' <constraint>)+ .
<disjunction>      := <constraint> ('∨' <constraint>)+ .
<negation>         := '' <constraint> .
<sim-constraint>   := <query-term> '≼u' <construct-term> .
<dep-constraint>   := '(' <constraint> '|' <constraint> ')' .
<query-constraint> := '⟨' <query-term> '⟩''{' <data-term-list>? '}' .
<dbterm-list>      := <data-term> (',' <data-term>)* .
```

It is easy to observe that a constraint store usually consists of arbitrary conjunctions, disjunctions, and negations of constraints. As usual, conjunctions always take precedence over disjunctions unless explicitly specified by parentheses. A brief description of the other kinds of constraints is given below:

Truth Values. The truth values "True" and "False" have their expected meaning in a constraint store. Simplification of the constraint store can eliminate them in all cases except when they are the only remaining constraint.

Simulation Constraint. A simulation constraint – written $t_1 \preceq_u t_2$ for some construct, data, or query term t_1 and some construct or data term t_2 – is a binary constraint which requires that variables are only bound to data terms such that there is a ground query term simulation between the ground instances of t_1 and t_2. The term t_1 is called the left hand side of the simulation constraint and t_2 is called the right hand side of the simulation constraint in subsequent sections. So as to distinguish the simulation constraint from the ground query term simulation, but nonetheless emphasise the relationship between the two, the symbol $\preceq_u$ is used (with u for "to be unified"). Note that the right hand side of a simulation constraint is always necessarily a construct or data term, because the simplification rules in the simulation unification and backward chaining algorithms never put a query term to the right hand side.

Most simulation constraints can be further reduced by applying the simulation unification algorithm on them until at least one of the sides consists merely of a variable. If a simulation constraint is of the form $X \preceq_u t$ where X is a variable, t is also called an *upper bound* of X. Likewise, if a simulation constraint is of the form $t \preceq_u X$, t is called an *lower bound* of X.

Query Constraint. A query constraint is a constraint consisting of a valid Xcerpt query (i.e. either a query term, an and/or-connection of queries, a negated query, or an input resource specification containing a query). Query constraints are used to represent queries that are not yet evaluated and are unfolded during the evaluation (if necessary). For some query Q, the query constraint is denoted by $\langle Q \rangle$.

A query constraint may optionally have a set of associated data terms which results from resolving and parsing an external resource (elimination of the `in` construct). If a query constraint $\langle Q \rangle$ is associated with the data terms $\{t_1, \ldots, t_n\}$, this is denoted by $\langle Q \rangle_{\{t_1,\ldots,t_n\}}$.

Dependency Constraint. A meta-constraint stating a dependency between two constraints. If C and D are constraints, the dependency constraint $(C \mid D)$ requires that C may only be evaluated if the evaluation of D did not fail (otherwise, the complete constraint fails). Thus D usually needs to be completely evaluated before C can be processed. The substitutions resulting from the evaluation of D are applied to C if they exist (i.e. under the condition that D is neither *False* nor *True*).

The justification for the dependency constraint are the requirements of the grouping constructs `all` and `some`, which require to consider all alternative solutions for the query part of a rule. If `all` or `some` appears in the head of a rule which is evaluated, the unification of a query with the head cannot be completed before the rule is fully evaluated.

4.1.2 Functions

substitutions(CS): The ultimate step of the algorithm, after no more rules are applicable or necessary, is always to generate a set of substitutions from the constraint store. In this step, *CS* is put in DNF, all constraints of the form $X \preceq_u t$ (where X is a variable and t is a construct term[4]) are replaced by $X = t$ and for each conjunct of *CS* a separate substitution is generated from these replacements. Note that

- $substitutions(True)$ is the set of all all-grounding substitutions
- $substitutions(False) = \{\}$, i.e. there exists no substitution.

Thus, neither a result of *True* nor a result of *False* are desirable for a query containing variables. Fortunately, the evaluation algorithm never yields *True* in case a variable occurs in a query, and only yields *False* if the evaluation fails.

apply(Σ,t): Applying a set of substitutions Σ to a term is implemented recursively over the term structure. The implementation of this function can be derived from Definitions 6 and 10 in a straightforward manner.

retrieve(R): Given a resource description R, the function $retrieve(R)$ returns a set of those terms that are represented by this resource provided that the data can in some way be parsed into Xcerpt's term representation. A resource description may for example contain a URI for identifying the resource and a format specification to indicate which parser to use. The current prototype provides support for XML, HTML and Xcerpt syntax, but different formats are more or less straightforward to implement (e.g. Lisp S-expressions, RDF statements or relational databases).

restrict(V,C): restricts the constraint store C to only such (non-negated) simulation constraints where the lower bound is a variable occurring in V. This function is used for evaluating query negation below.

deref(id): Dereferences the term reference identified by *id* and returns the subterm associated with the identifier *id*.

vars(Q): Returns the set of all variables occurring non-negated in a query Q.

[4] due to the way rules are evaluated, the right hand side of a simulation constraint is always a construct term

4.2 Solution Set of a Constraint Store

As the evaluation algorithm aims at determining an (all-grounding) substitution set for certain variables, each constraint store conceptually represents a (all-grounding) substitution set in which each substitution provides assignments for all conceivable variable names. This set is called the *solution set* of the constraint store, and represents the possible answers that the evaluation of the constraint store yields. Depending on the constraint store, this solution set is restricted to substitutions fulfilling certain conditions. For example, the constraint $X \preceq_u f\{a\}$ requires that all substitutions in the solution set provide an assignment for the variable X that is compatible (i.e. *simulates*) with $f\{a\}$. Likewise, the constraint $f\{\{\}\} \preceq_u X$ requires that the solution set only contains substitutions that provide an assignment t for X such that $f\{\{\}\} \preceq t$.

In the following, we will consider only the solution set of a fully solved constraint store. Such a constraint store contains only simulation constraints where one side of the inequation is a variable, of conjunctions or disjunctions of constraints, and of the boolean constraints *True* and *False*. This notion of solution sets will be used in the formalisation of simulation unifiers later in this chapter. Recall that all-grounding substitutions are substitutions that map every possible variable to a data term.

Definition 21 (Solution Set of a Constraint Store)
Let *CS* be a completely solved constraint store, i.e. consisting only of simulation constraints where one side is a variable, conjunctions, disjunctions, and the boolean constraints *True* and *False*. The solution set $\Omega(CS)$ is a grounding substitution set recursively defined as follows:

- $\Omega(True)$ is the set of all all-grounding substitutions (cf. Section 2.2)
- $\Omega(False) = \{\}$, i.e. the empty set
- $\Omega(X \preceq_u t)$ is the set of all all-grounding substitutions σ such that $\sigma(X) \cong \sigma(t)$
- $\Omega(t \preceq_u X)$ is the set of all all-grounding substitutions σ such that $\sigma(t) \preceq \sigma(X)$
- $\Omega(C_1 \wedge C_2) = \Omega(C_1) \cap \Omega(C_2)$
- $\Omega(C_1 \vee C_2) = \Omega(C_1) \cup \Omega(C_2)$
- $\Omega(C) = \Omega(True) \setminus \Omega(C)$

The rationale behind using sets of all-grounding substitutions is that a constraint store in general merely restricts the possible answers. Further constraints might add new variables that would have to be considered. Using infinite substitutions also simplifies working with the solution set, because it suffices to use simple set operations instead of introducing a new "substitution theory". For example, merging of two all-grounding substitution sets merely requires the intersection of both.

Note that the solution set of a constraint store *CS* is in general always infinite, because each substitution contains assignments for an infinite number of variables. However, restricting this set to only finitely many variables V (i.e. those variables occurring in *CS*), yields a finite set in case every such variable occurs in each conjunct of the disjunctive normal form of *CS* on the right side of a simulation constraint.

The following result is important because it relates the abstract notion of solution set to the actually computed substitutions. It follows trivially from the definition of solution sets and the definition of the function $substitutions(\cdot)$. Recall that $\Sigma_{|V}$ is the substitution set Σ restricted to the variables in V.

Corollary 22
Let $CS = C_1 \vee \cdots \vee C_n$ be a constraint store in disjunctive normal form, and V the set of variables occurring in CS. If in every conjunct C_i, each variable $X \in V$ occurs in a simulation constraint of the form $X \preceq_u t$ where t is a data term, then $substitutions(CS) = \Omega(CS)_{|V}$.

Note that as Xcerpt programs are range restricted, this corollary holds for every full evaluation of an Xcerpt program.

4.3 Constraint Simplification

The usual simplification rules for formulas apply, for example:

- $False \wedge C$ reduces to $False$ for any constraint C, $False \vee C$ reduces to C for any constraint C
- $True \wedge C$ reduces to C for any constraint C, $True \vee C$ reduces to $True$ for any constraint C
- $(C \wedge D)$ simplifies to $C \vee D$, $(C \vee D)$ simplifies to $C \wedge D$
- C simplifies to C
- $False = True$ and $True = False$

Note, however, that constraints of the form C (where C is not of the form C') may not be simplified to C, because the range restrictedness disallows variable bindings also for variables that are negated twice or more times.

4.4 Consistency Verification Rules

Before a variable can be bound to a term, it is necessary that the constraints for this variable are *consistent*. There are two kinds of consistency verification rules, *consistency* and *transitivity*, divided into four rules to distinguish the cases with and without negation. The fifth rule described here reduces certain kinds of negated simulation constraints.

All consistency verification rules are considered to be part of the constraint solver and are needed both for the simulation unification and the backward chaining algorithm. It is assumed that they are always applied if possible and that the constraint store can always be treated as consistent.

4.4.1 Rule 1: Consistency

The *consistency* rule guarantees that upper bounds for a variable are consistent. This verification rule implements the solution set definition of $\Omega(C \wedge D) = \Omega(C) \cap \Omega(D)$ and ensures that a conjunct does not induce two assignments for a variable that are not simulation equivalent.

$$\frac{X \preceq_u t_1 \quad X \preceq_u t_2}{X \preceq_u t_1 \wedge t_1 \preceq_u t_2 \wedge t_2 \preceq_u t_1}$$

Note that both t_1 and t_2 are necessarily construct or data terms. Thus, the constraint $\preceq_u$ is applicable, which requires a construct or data term on the right hand side.

Example 23 (Consistency Rule)

1. consider the two simulation constraints $X \preceq_u f\{var\ Y\}$ and $X \preceq_u f\{a\}$; applying the consistency rule yields $X \preceq_u f\{var\ Y\} \wedge a \preceq_u Y \wedge Y \preceq_u a$ (after mutual unification), which limits the bindings for Y to a.

2. consider the two simulation constraints $X \preceq_u f\{a\}$ and $X \preceq_u f\{b\}$; applying the consistency rule determines that they are inconsistent, because $f\{a\}$ and $f\{b\}$ do not simulate.

4.4.2 Rule 2: Transitivity

The *transitivity* rule replaces variable occurrences of a variable X in the upper bounds of a variable by the upper bound of X. This rule is justified by the simulation pre-order defined in [SBF05] and is needed to ultimately create ground terms as bindings for all variables. In the following, the notation $t[t'/X]$ denotes "replace all occurrences of X in t by t'".

$$\frac{\begin{array}{l} t_1 \preceq_u t_1' \text{ such that } t_1' \text{ contains the variable } X \\ X \preceq_u t_2 \end{array}}{X \preceq_u t_2 \wedge t_1 \preceq_u t_1'[t_2/X]}$$

Note that the first constraint is consumed by this rule. This might appear somewhat unusual, as further applications of the transitivity rule might yield new constraints. However, if some constraint of the form $X \preceq_u t_2'$ is added, it needs to be compatible with the constraint $X \preceq_u t_2$ (which is still in the conjunction) and would thus not yield differing information.

Example 24 (Transitivity Rule)

1. consider the simulation constraints $X \preceq_u Y$ and $Y \preceq_u a$; applying the transitivity rule yields the additional constraint $X \preceq_u a$ and removes $X \preceq_u Y$.

2. consider the simulation constraints $X \preceq_u f\{var\ Y\}$ and $Y \preceq_u a$; applying the transitivity rule yields the additional constraint $X \preceq_u f\{a\}$ and removes $X \preceq_u f\{var\ Y\}$.

It would be possible to define a similar transitivity rule for the lower bounds in a simulation constraints. This is, however, not necessary, as the lower bounds do not yield variable bindings and thus need not be ground.

4.5 Constraint Negation

Negated constraints represent exclusion of certain variable bindings, and may result from the evaluation of the constructs `without` (subterm negation), `optional` (optional subterms), and `not` (query negation). For example, the constraint $(X \preceq f\{a,b\})$ disallows bindings for X that are simulation equivalent with $f\{a,b\}$. Note that, although these constructs implement negation as failure, constraint negation is the ordinary negation of classical logic. The usual transformation rules apply, namely $(C \wedge D) = C \vee D$, $(C \vee D) = C \wedge D$, $True = False$, and $False = True$. Note, however, that $C \neq C$, because C is not allowed to define variable bindings (cf. *range restrictedness*, [SBF05]).

The following three additional consistency verification rules are used in the constraint solver to treat constraint negation. All three rules assume that the negation appears immediately in front of an atomic constraint. This assumption is safe when the constraint store is in disjunctive normal form. The rules continue the numbering scheme of the previous consistency verification rules. Therefore, the first rule has number 3.

4.5.1 Rule 3: Consistency with Negation

To detect inconsistencies between a non-negated and a negated simulation constraints, the consistency rule needs to be modified to yield inconsistency in case a non-negated constraint for a variable is consistent with a negated constraint for the same variable. The following rule means that if a simulation constraint provides an upper bound for a variable (which represents a candidate binding for the variable), then there must not be a negated simulation constraint that excludes this upper bound:

$$\frac{\begin{array}{l} X \preceq_u t_1 \\ (X \preceq_u t_2) \end{array}}{X \preceq_u t_1 \wedge (t_1 \preceq_u t_2 \wedge t_2 \preceq_u t_1)}$$

Example 25 (Consistency Rule with Negation)
Consider the constraint store

$$X \preceq_u f\{a,b\} \wedge (X \preceq_u f\{b,a\}) \wedge (X \preceq_u g\{a\})$$

Applying the consistency rule with negation yields

$$X \preceq_u f\{a,b\} \wedge (f\{a,b\} \preceq_u f\{b,a\} \wedge f\{b,a\} \preceq_u f\{a,b\}) \wedge (X \preceq_u g\{a\})$$

the DNF of which is

$$\begin{array}{l} X \preceq_u f\{a,b\} \wedge (f\{a,b\} \preceq_u f\{b,a\}) \wedge (X \preceq_u g\{a\}) \vee \\ X \preceq_u f\{a,b\} \wedge (f\{b,a\} \preceq_u f\{a,b\}) \wedge (X \preceq_u g\{a\}) \end{array}$$

and after further decomposition steps

$$\begin{array}{l} X \preceq_u f\{a,b\} \wedge (True) \wedge (X \preceq_u g\{a\}) \vee \\ X \preceq_u f\{a,b\} \wedge (True) \wedge (X \preceq_u g\{a\}) \end{array}$$

which ultimately yields *False*, i.e. no valid bindings.

Note that although subterm and query negation can never yield variable bindings themselves, there might be variables that only appear in negated simulation constraints but nowhere else in a non-negated simulation constraint, e.g. as the result of decomposition with `without` or `optional`. These are treated by Rule 5 below.

4.5.2 Rule 4: Transitivity with Negation

Like the consistency rule, the transitivity rule needs to be adapted to cover negation properly. The following rule specifies that if there is a negated simulation constraint where the upper bound t_1' contains a variable, and this variable is bounded in a non-negated simulation constraint, then substituting the upper bound for the variable in the first constraint must not yield a simulation.

$$\frac{\begin{array}{l} (t_1 \preceq_u t_1') \text{ such that } t_1' \text{ contains the variable } X \\ X \preceq_u t_2 \end{array}}{(t_1 \preceq_u t_1') \wedge X \preceq_u t_2 \wedge (t_1 \preceq_u t_1'[t_2/X])}$$

Likewise, if there is a non-negated simulation constraint where the upper bound contains a variable occurring in a negated simulation constraint, then substituting the upper bound for the variable in the first constraint must not yield a simulation.

$$\frac{\begin{array}{l} t_1 \preceq_u t_1' \text{ such that } t_1' \text{ contains the variable } X \\ (X \preceq_u t_2) \end{array}}{t_1 \preceq_u t_1' \wedge (X \preceq_u t_2) \wedge (t_1 \preceq_u t_1'[t_2/X])}$$

Note that unlike rule 2, transitivity with negation may not remove any of the original constraints, because information would be lost.

4.5.3 Rule 5: Negation as Failure

The last rule is necessary for cases where a variable only appears in a negated simulation constraint, but nowhere else in a non-negated simulation constraint of the constraint store. Due to the range restrictedness of Xcerpt rules, such constraints can never be produced directly in the treatment of `not` or `without` (range restrictedness enforces that each variable occurring in a negated part also appears elsewhere in a non-negated part). They may, however, be the consequence of applications of rules 3 and 4, and might be produced when decomposing a query term containing the construct `optional` (see Section 5.1 below).

Such constraints are reduced to *False*. The rationale behind this is that, in case the variable does not occur elsewhere outside a negation, the simulation constraint inside the negation represents a solution for a negated query or subterm, and therefore the negated constraint must fail. In case the variable does also appear elsewhere outside a negation rules 3 and 4 are applicable (which again might yield negated simulation constraints).

$$\frac{(X \preceq_u t) \text{ such that } X \text{ does not appear in a non-negated simulation constraint}}{False}$$

Constraints of the form *True* and *False* are treated by the formula simplification described above. Example 33 shows a case where this consistency rule is needed. An interesting application of this rule involves double negation:

Example 26 (Negation as Failure Rule)
Consider the simulation constraint $(X \preceq_u t)$ such that X does not occur elsewhere in a non-negated simulation constraint. Applying Rule 5 to this constraint yields $False = True$ (and not $X \preceq_u t$ as one might expect). The rationale for this is that the negation used is negation as failure and not classical negation, and variables within a simulation constraint that are negated twice do not define variable bindings (see also the definition of *range restrictedness* in [SBF05].

4.6 Program Evaluation

Program evaluation starts at the program goals, and tries to determine answer terms by evaluating the query parts for each goal in a backward chaining fashion. Given a program P, the general scheme of program evaluation is as follows:

Algorithm 27

procedure main():
  foreach goal $t \leftarrow Q \in \mathscr{P}$ **do**:
    let Subst **:=** solve($\langle Q \rangle_\emptyset$)
    print apply(t,Subst)

Of course, printing the result in the scheme above has to respect a possible output resource associated with the head of a goal. The backward chaining algorithm itself is called with the function *solve*(C) (where C is a constraint) which returns a list of substitutions that result from solving the constraint given as parameter. The general scheme of the function *solve* is as follows (cf. the function `substitutions(·)` above):

Algorithm 28

function solve(Constraint C):
  while a rule can be applied to C **do**:
    select some constraint D in C and some rule R applicable to D
    let D' **:=** apply rule R to D
    replace D by D' in C
    put C in disjunctive normal form and verify consistency
  return substitutions(C)

Note that "rule" in the algorithm above denotes a simplification rule of the constraint solver and not an Xcerpt rule. Rules from all three parts may be interleaved and the decision on the selection of rule applications is deliberately left open (i.e. the algorithm described here is non-deterministic), as long as the selection is "fair" (i.e. each possible rule is applied within finitely many steps). This non-determinism allows for interesting considerations about selection strategies that have not been investigated much in logic programming.

5 Simulation Unification

Simulation Unification consists mainly of decomposition rules that operate recursively and in parallel on the two unified terms (Section 5.1). When all terms are completely decomposed, the result is a constraint store containing conjunctions and disjunctions of simulation constraints where the left or the right side is a variable. These yield variable bindings by replacing simulation constraints of the form $X \preceq_u t$ by $X = t$. The consistency verification rules described above ensure that all simulation constraints are consistent and can be interleaved at any point.

5.1 Decomposition Rules

Decomposition rules take a single simulation constraint and try to recursively decompose the two terms in parallel until no further rules are applicable. Each decomposition step yields one or more subsequent constraints, often even a large disjunction containing alternatives. This reflects the many different alternative ground query term simulations that need to be considered in case of partial term specifications.

All decomposition rules are first given without examples, because the examples tend to be very extensive, and mutually depend on other decomposition rules. Section 5.2 illustrates important aspects of simulation unification on several more extensive examples.

5.1.1 Preliminaries

In the following, let l (with or without indices) denote a label, and let t^1 denote query terms and t^2 construct terms (both with or without indices). Furthermore, let $\perp$ be a special term (not occurring as subterm in any actual term) with the property that for all $t \neq \perp$ holds that $t \preceq_u \perp = \mathit{False}$, i.e. no term unifies with $\perp$. In the following sections, it is furthermore assumed that t^2 contains neither grouping

constructs, functions, aggregations, nor optional subterms. In practice, this restriction is insignificant, because construct terms containing one of these constructs are always made ground before computing the simulation unification (see *Dependency Constraint* below).

Definition 29
Given two terms $t^1 = l\{t^1_1, \ldots, t^1_n\}$ and $t^2 = l\{t^2_1, \ldots, t^2_m\}$, the following sets of functions $\Pi_X : \langle t^1_1, \ldots, t^1_n \rangle \rightarrow \langle t^2_1, \ldots, t^2_m \rangle$ are defined (cf. Definition 14):

- $SubT^+ \subseteq \langle t^1_1, \ldots, t^1_n \rangle$ is the sequence of all non-negated subterms of t^1 and $SubT^- \subseteq \langle t^1_1, \ldots, t^1_n \rangle$ is the sequence of all negated subterms of t^1
- $SubT^! \subseteq \langle t^1_1, \ldots, t^1_n \rangle$ is the sequence of all non-optional subterms of t^1 and $SubT^? \subseteq \langle t^1_1, \ldots, t^1_n \rangle$ is the sequence of all optional subterms of t^1
- Π is the set of partial, index injective functions π from $\langle t^1_1, \ldots, t^1_n \rangle$ to $\langle t^2_1, \ldots, t^2_m \rangle$ that are total on $SubT^+$ and on $SubT^!$, each completed by $t \mapsto \bot$ for all t on which π is not defined
- Π_{mon} is the set Π restricted to all index monotonic functions
- Π_{bij} is the set Π restricted to all index bijective functions
- Π_{pp} is the set of all position *preserving* functions
- Π_{pr} is the set of all position *respecting* functions
- $\Pi_{m-pr} = \Pi_{mon} \cap \Pi_{pr}$, $\Pi_{b-pr} = \Pi_{bij} \cap \Pi_{pr}$, $\Pi_{b-pp} = \Pi_{bij} \cap \Pi_{pp}$, and $\Pi_{m-b} = \Pi_{bij} \cap \Pi_{mon}$

To simplify the rules below, all *partial* mappings in Π are assumed to be completed by mapping all values on which the mappings are undefined to the special term $\bot$. In this manner, every mapping in Π can be considered to be total in case the distinction is not necessary, whereas in the cases where partial mappings are considered (`optional` and `without`), the distinction is made explicitly.

Example 30
Consider the terms $t^1 = f[[a, \text{without } b]]$ and $t^2 = f[a,b,c]$. The set of index monotonic mappings of the set of subterms of t^1 into the set of subterms of t^2 (Π_{mon}) is as follows (`without` b abbreviated as b):

$$\{a \mapsto a, b \mapsto \bot\} \quad \{a \mapsto b, b \mapsto \bot\} \quad \{a \mapsto c, b \mapsto \bot\}$$
$$\{a \mapsto a, b \mapsto b\} \quad \{a \mapsto b, b \mapsto c\}$$
$$\{a \mapsto a, b \mapsto c\}$$

Note that all these mappings can be generated in a rather straightforward manner by creating a table with the terms $t^1_1 \cdots t^1_n$ arranged top-down and the terms $t^2_1 \cdots t^2_m$ arranged left-right and then determining paths from the first line to the n^{th} line that fulfil certain properties. This technique is called the *memoisation matrix*.

5.1.2 Root Elimination

Root elimination rules compare the roots of the two terms and distribute the unification to the subterms.

Brace Incompatibility The first set of rules treat incompatibility between braces and thus all of these rules reduce the simulation constraint to *False*. For instance, an ordered simulation into an unordered term is not reasonable, as the order cannot be guaranteed.
Decomposition Rule ***decomp.1***:

$$\frac{l[t_1^1,\ldots,t_n^1] \preceq_u l\{t_1^2,\ldots,t_m^2\}}{False} \qquad \frac{l[[t_1^1,\ldots,t_n^1]] \preceq_u l\{t_1^2,\ldots,t_m^2\}}{False}$$

Left Term without Subterms This set of rules consider all such cases where the left term does not contain subterms. These cases have to be treated separately from the general decomposition rules below, since using the latter would yield the wrong result in such cases. For instance, an empty *or* is equvialent to *False* but the result should always be *True* in case the left term is only a partial specification. In the following, let $m \geq 0$ and $k \geq 1$:
Decomposition Rule ***decomp.2***:

$$\frac{l\{\{\ \}\} \preceq_u l\{t_1^2,\ldots,t_m^2\}}{True} \qquad \frac{l\{\{\ \}\} \preceq_u l[t_1^2,\ldots,t_m^2]}{True} \qquad \frac{l[[\ ]] \preceq_u l[t_1^2,\ldots,t_m^2]}{True}$$

$$\frac{l\{\ \} \preceq_u l\{t_1^2,\ldots,t_k^2\}}{False} \qquad \frac{l\{\ \} \preceq_u l[t_1^2,\ldots,t_k^2]}{False} \qquad \frac{l[\ ] \preceq_u l[t_1^2,\ldots,t_k^2]}{False}$$

$$\frac{l\{\ \} \preceq_u l\{\ \}}{True} \qquad \frac{l\{\ \} \preceq_u l[\ ]}{True} \qquad \frac{l[\ ] \preceq_u l[\ ]}{True}$$

As specified by these rules, a term without subterms but a partial specification (double braces) matches with any term which has the same label. If the term specification is total, it matches only with such terms that also do not have subterms.

Decomposition without `all`, `some`, `without`, and `optional` The general decomposition rules eliminate the two root nodes in parallel and distributes the unification to the various combinations of subterms that result from ordered/unordered specification and from total/partial term specifications. If there exists no such combination, then the result is an empty *or*, which is equivalent to *False*. These term specifications are represented by the different sets of mappings Π, Π_{bij}, Π_{mon}, Π_{pr}, and Π_{pp}. In the following, let $n, m \geq 1$.
Decomposition Rule ***decomp.3***:

$$\frac{l\{\{t_1^1,\ldots,t_n^1\}\} \preceq_u l\{t_1^2,\ldots,t_m^2\}}{\bigvee_{\pi\in\Pi_{pp}} \bigwedge_{1\leq i\leq n} t_i^1 \preceq_u \pi(t_i^1)} \qquad \frac{l\{\{t_1^1,\ldots,t_n^1\}\} \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{pr}} \bigwedge_{1\leq i\leq n} t_i^1 \preceq_u \pi(t_i^1)}$$

$$\frac{l\{t_1^1,\ldots,t_n^1\} \preceq_u l\{t_1^2,\ldots,t_m^2\}}{\bigvee_{\pi\in\Pi_{bij}\cap\Pi_{pp}} \bigwedge_{1\leq i\leq n} t_i^1 \preceq_u \pi(t_i^1)} \qquad \frac{l\{t_1^1,\ldots,t_n^1\} \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{bij}\cap\Pi_{pr}} \bigwedge_{1\leq i\leq n} t_i^1 \preceq_u \pi(t_i^1)}$$

$$\frac{l[[t_1^1,\ldots,t_n^1]] \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{mon}\cap\Pi_{pr}} \bigwedge_{1\leq i\leq n} t_i^1 \preceq_u \pi(t_i^1)} \qquad \frac{l[t_1^1,\ldots,t_n^1] \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{mon}\cap\Pi_{bij}} \bigwedge_{1\leq i\leq n} t_i^1 \preceq_u \pi(t_i^1)}$$

For instance, if the left term has a partial, unordered specification for the subterms, the simulation unification has to consider as alternatives all combinations of subterms of the left term with subterms of the right term, provided that each child on the left gets a matching partner on the right.

Label Mismatch In case of a label mismatch, the unification fails. In the following, let $l_1 \neq l_2$.
*Decomposition Rule **decomp.4***:

$$\frac{l_1\{\{t_1^1,\ldots,t_n^1\}\} \preceq_u l_2\{t_1^2,\ldots,t_m^2\}}{False} \qquad \frac{l_1\{t_1^1,\ldots,t_n^1\} \preceq_u l_2\{t_1^2,\ldots,t_m^2\}}{False}$$

$$\frac{l_1\{\{t_1^1,\ldots,t_n^1\}\} \preceq_u l_2[t_1^2,\ldots,t_m^2]}{False} \qquad \frac{l_1\{t_1^1,\ldots,t_n^1\} \preceq_u l_2[t_1^2,\ldots,t_m^2]}{False}$$

$$\frac{l_1[[t_1^1,\ldots,t_n^1]] \preceq_u l_2[t_1^2,\ldots,t_m^2]}{False} \qquad \frac{l_1[t_1^1,\ldots,t_n^1] \preceq_u l_2[t_1^2,\ldots,t_m^2]}{False}$$

5.1.3 $\rightsquigarrow$ Elimination

Pattern restrictions of the form $X \rightsquigarrow t^1 \preceq_u t^2$ are decomposed by adding t_2 as upper bound for the variable X (as usual), adding the pattern restriction as lower bound for X (to ensure that there exists no upper bound that is incompatible with the pattern restriction), and immediately trying to unify t_1 and t_2. The latter step is not strictly necessary, as it would also be performed by consistency rule 2 (transitivity). However, immediate evaluation is advantageous as it excludes incompatible upper bounds immediately.
*Decomposition Rule **var***:

$$\frac{X \rightsquigarrow t^1 \preceq_u t^2}{t^1 \preceq_u t^2 \wedge t^1 \preceq_u X \wedge X \preceq_u t^2}$$

5.1.4 Descendant Elimination

The descendant construct in terms of the form *desc t* is decomposed by first trying to unify t with the other term, and then trying to unify *desc t* with each of the subterms of the other term in turn. In this manner, unifying subterms at all depths can be determined. Let $m \geq 0$.
*Decomposition Rule **desc***:

$$\frac{\text{desc}\, t^1 \preceq_u l\{t_1^2,\ldots,t_m^2\}}{t^1 \preceq_u l\{t_1^2,\ldots,t_m^2\} \vee \bigvee_{1\leq i\leq m} desc\, t^1 \preceq_u t_i^2} \qquad \frac{\text{desc}\, t^1 \preceq_u l[t_1^2,\ldots,t_m^2]}{t^1 \preceq_u l[t_1^2,\ldots,t_m^2] \vee \bigvee_{1\leq i\leq m} desc\, t^1 \preceq_u t_i^2}$$

5.1.5 Decomposition with `without`

The declarative specification of `without` in the ground query term simulation of Section 3.2 requires that a partial function (of the set of non-negated subterms into the set of subterms of the second term) is not completable to a (partial or total) function such that one of the negated subterm is mapped to a subterm in which it simulates. Since the term on the right hand side of a simulation constraint is always a data or construct term, it is sufficient to consider the case where the right term does not contain negated subterms. For a simulation constraint $t^1 \preceq_u t^2$, the decomposition rules for the case without negated subterms is intuitively described as follows:

- A mapping π is first restricted to the non-negated subterms of t^1, i.e. the subterms of the left term that are not of the form `without` t, on which the decomposition is performed in the same way as for decomposition without `without`. Note that there might be several different mappings that are identical with π for all the non-negated subterms and only differ on the negated subterms.

- It is then necessary to verify whether there exists a mapping π' that maps the non-negated subterms of t^1 to the same subterms of t^2 as π (in particular, π' might be π itself), and permits to map at least one negated subterm `without` s^1 of t^1 to a subterm s^2 of t^2 such that $s^1 \preceq s^2$. In this case, the mapping restricted to the positive subterms of t^1 is considered to be invalid, because it is completable to a mapping that allows to map a negated subterm of t^1 to a matching non-negated subterm of t^2. Thus, *all* mappings that map the positive subterms of t^1 to the same subterms of t^2 have to be ruled out.

It is important to note that the set of mappings Π is defined (in the Preliminaries above) as the set of all *partial* functions that are *total* on the set of positive subformulas. Recall furthermore, that the mappings in Π are completed by mapping all undefined values to $\perp$.

In the following, let $SubT^+ \subseteq \langle t_1^1, \ldots, t_n^1 \rangle$ be the sequence of all subterms not of the form `without` t, and let $SubT^- \subseteq \langle t_1^1, \ldots, t_n^1 \rangle$ be the sequence of all subterms of the form `without` t. Also, two functions π and π' are considered to be equal on the positive part, denoted $\pi(SubT^+) = \pi'(SubT^+)$, if for all $t \in SubT^+$ holds that $\pi(t) = \pi'(t)$. Furthermore, let $p(\cdot)$ be a function that removes the `without` construct in front of a negated subterm, i.e. $p(\text{without } t) = t$.
Decomposition Rule ***without***:

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{\bigvee_{\pi \in \Pi_{pp}} \left(\bigwedge_{t^+ \in SubT^+} t^+ \preceq_u \pi(t^+) \wedge \left(\bigvee_{\pi' \in \Pi_{pp} \text{ with } \pi(SubT^+) = \pi'(SubT^+)} \bigvee_{t^- \in SubT^-} p(t^-) \preceq_u \pi'(t^-) \right) \right)}$$

$$\frac{l[[t_1^1, \ldots, t_n^1]] \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{\text{m-pr}}} \left(\bigwedge_{t^+ \in SubT^+} t^+ \preceq_u \pi(t^+) \wedge \left(\bigvee_{\pi' \in \Pi_{\text{m-pr}} \text{ with } \pi(SubT^+) = \pi'(SubT^+)} \bigvee_{t^- \in SubT^-} p(t^-) \preceq_u \pi'(t^-) \right) \right)}$$

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{pr}} \left(\bigwedge_{t^+ \in SubT^+} t^+ \preceq_u \pi(t^+) \wedge \left(\bigvee_{\pi' \in \Pi_{pr} \text{ with } \pi(SubT^+) = \pi'(SubT^+)} \bigvee_{t^- \in SubT^-} p(t^-) \preceq_u \pi'(t^-) \right) \right)}$$

Note that decomposition with `without` is currently not covered in the completeness and correctness proofs of Section 5.3.

5.1.6 Decomposition with `optional` in the query term

Intuitively, decomposition with `optional` in the query term should "enable" the maximal number of optional subterms such that they can participate in the simulation. In the following, this is expressed as follows:

- for all required subterms (i.e. not of the form `optional` t), the treatment is as before (since all negated subterms are required, they must be treated here as well, but this is omitted in the rules below to enhance readability)
- for all optional subterms, a certain number is "enabled" by adding appropriate simulation constraints, and all others are "disabled" by adding appropriate negated simulation constraints

In the following, these requirements are expressed as follows: given a partial mapping $\pi \in \Pi$ (by definition π must be total on the set of non-optional subterms, but may be partial on the set of optional subterms), it is first verified whether π yields a simulation by unifying all terms on which π is defined with their mapping (in the same manner as before). In the second part of the formula, it is then necessary to ensure that π is also the *maximal* mapping with this property, i.e. π is not completable to a mapping

π' such that this would also yield a simulation. This is ensured by adding a negated disjunction testing for all mappings that are identical with π on the subterms for which π is defined, but differ on the other subterms, whether there exists an additional subterm that would unify with the subterm it is mapped to in π'. If yes, π is not maximal and completable to π'. If no, π is maximal.

For a given mapping π, let $SubT_\pi \subseteq SubT$ be the sequence on which π is defined and not mapped to $\bot$, i.e. for all $t \in SubT_\pi$ holds that $\pi(t) \neq \bot$, and let $\overline{SubT_\pi} = SubT \setminus SubT_\pi$. Also, two functions π and π' are considered to be equal on a set of subterms $X \subseteq SubT$, denoted $\pi(X) = \pi'(X)$, if for all $t \in X$ holds that $\pi(t) = \pi'(t)$. Furthermore, let $p(\cdot)$ be a function that removes the `optional` construct in front of an optional subterm, i.e. $p(\text{optional } t) = t$.
Decomposition Rule ***optional***:

$$\frac{l\{t_1^1,\ldots,t_n^1\} \preceq_u l\{t_1^2,\ldots,t_m^2\}}{\bigvee_{\pi\in\Pi_{b-pp}}\left(\bigwedge_{t\in SubT_\pi} t \preceq_u \pi(t) \wedge \left(\bigvee_{\pi'\in\Pi_{b-pp} \text{ with } \pi(SubT_\pi)=\pi'(SubT_\pi)} \bigvee_{t'\in\overline{SubT_\pi}} p(t') \preceq_u \pi'(t')\right)\right)}$$

$$\frac{l\{\{t_1^1,\ldots,t_n^1\}\} \preceq_u l\{t_1^2,\ldots,t_m^2\}}{\bigvee_{\pi\in\Pi_{pp}}\left(\bigwedge_{t\in SubT_\pi} t \preceq_u \pi(t) \wedge \left(\bigvee_{\pi'\in\Pi_{pp} \text{ with } \pi(SubT_\pi)=\pi'(SubT_\pi)} \bigvee_{t'\in\overline{SubT_\pi}} p(t') \preceq_u \pi'(t')\right)\right)}$$

$$\frac{l[t_1^1,\ldots,t_n^1] \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{m-b}}\left(\bigwedge_{t\in SubT_\pi} t \preceq_u \pi(t) \wedge \left(\bigvee_{\pi'\in\Pi_{m-b} \text{ with } \pi(SubT_\pi)=\pi'(SubT_\pi)} \bigvee_{t'\in\overline{SubT_\pi}} p(t') \preceq_u \pi'(t')\right)\right)}$$

$$\frac{l[[t_1^1,\ldots,t_n^1]] \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{m-pr}}\left(\bigwedge_{t\in SubT_\pi} t \preceq_u \pi(t) \wedge \left(\bigvee_{\pi'\in\Pi_{m-pr} \text{ with } \pi(SubT_\pi)=\pi'(SubT_\pi)} \bigvee_{t'\in\overline{SubT_\pi}} p(t') \preceq_u \pi'(t')\right)\right)}$$

$$\frac{l\{t_1^1,\ldots,t_n^1\} \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{b-pr}}\left(\bigwedge_{t\in SubT_\pi} t \preceq_u \pi(t) \wedge \left(\bigvee_{\pi'\in\Pi_{b-pr} \text{ with } \pi(SubT_\pi)=\pi'(SubT_\pi)} \bigvee_{t'\in\overline{SubT_\pi}} p(t') \preceq_u \pi'(t')\right)\right)}$$

$$\frac{l\{\{t_1^1,\ldots,t_n^1\}\} \preceq_u l[t_1^2,\ldots,t_m^2]}{\bigvee_{\pi\in\Pi_{pr}}\left(\bigwedge_{t\in SubT_\pi} t \preceq_u \pi(t) \wedge \left(\bigvee_{\pi'\in\Pi_{pr} \text{ with } \pi(SubT_\pi)=\pi'(SubT_\pi)} \bigvee_{t'\in\overline{SubT_\pi}} p(t') \preceq_u \pi'(t')\right)\right)}$$

Note the close similarity to the decomposition rules for terms containing `without`. Intuitively, this similarity means that decomposition with `optional` corresponds to creating all different alternatives where zero or more `optional` subterms are "turned on" by omitting the `optional` and the others are "turned off" by replacing `optional` by `without`, and evaluating all resulting terms as alternatives. Consider for example the term

$$f\{\{var\ X \to a, \text{optional } var\ Y \to b, \text{optional } var\ Z \to c\}\}$$

The substitution resulting from the evaluation of this query term is equivalent to the union of the results of the four terms

$$f\{\{var\ X \to a, var\ Y \to b, var\ Z \to c\}\}$$
$$f\{\{var\ X \to a, var\ Y \to b, \text{without } var\ Z \to c\}\}$$
$$f\{\{var\ X \to a, \text{without } var\ Y \to b, var\ Z \to c\}\}$$
$$f\{\{var\ X \to a, \text{without } var\ Y \to b, \text{without } var\ Z \to c\}\}$$

Note that this representation might be surprising on a first glance, because the intuitive understanding of `optional` would be to simply leave out the optional subterms instead of replacing them by negated subterms, as in:

$$f\{\{var\ X \rightarrow\ a, var\ Y \rightarrow\ b, var\ Z \rightarrow\ c\}\}$$
$$f\{\{var\ X \rightarrow\ a, var\ Y \rightarrow\ b\}\}$$
$$f\{\{var\ X \rightarrow\ a, var\ Z \rightarrow\ c\}\}$$
$$f\{\{var\ X \rightarrow\ a\}\}$$

However, this term representation does not reflect that an optional subterm is *required* to match, if it is *possible* to match. Consider for example a unification with the term $f\{a,c\}$. The correct solution would be the substitution set

$$\Sigma = \big\{\{X \mapsto a, Z \mapsto c\}\big\}$$

whereas the evaluation of the second set of terms would yield

$$\Sigma = \big\{\{X \mapsto a, Z \mapsto c\}, \{X \mapsto a\}\big\}$$

Note that decomposition with `optional` is currently not covered in the completeness and correctness proofs of Section 5.3.

Example 33 on page 36 illustrates the decomposition of a term containing two optional subterms. Note that more efficient evaluation techniques for the decomposition rules above are conceivable. For example, if one of the unification steps in the part for which π is defined already fails, it is not necessary to consider all different alternative mappings that are equal on the subterms on which π is defined.

5.1.7 Incomplete Decomposition with grouping constructs, functions, aggregations, and optional subterms in construct terms

A unification with a term containing grouping constructs, functions, or aggregations is in general incomplete because a complete decomposition requires to handle meta-constraints over the constraint store itself, which is very inconvenient. Consider for instance a unification $f\{a,b,c\} \preceq_u f[all\ X]$. To provide the full information stated in this constraint, it would be necessary to add a meta-constraint stating that there must be exactly three alternative bindings for X, and of those, one must be a, another b and the third c. Evaluation of a query containing X would thus become very complex.

Although a complete decomposition is preferable, it is (fortunately) not necessary for evaluating Xcerpt programs, as grouping constructs always depend on the bindings of the variables in the query part of a rule. Rules containing grouping constructs are treated by the *dependency constraint*, which performs an auxiliary computation for solving the query part of a rule and then substitutes the results in the rule head. Thus, in this case it is sufficient to treat the unification of a query term with a data term, which does not contain grouping constructs (and obviously also no variables).

However, it is still desirable to unify a term containing grouping constructs as far as possible in order to exclude irrelevant evaluations of query parts in the dependency constraint as early as possible. For example, the terms $f\{a,b\}$ and $g\{all\ var\ X\}$ will never yield terms that unify, regardless of the bindings for X. Likewise, the terms $f\{g\{a\},g\{b\}\}$ and $f\{all\ h\{var\ X\}\}$ will never yield terms that unify, because neither $g\{a\}$ nor $g\{b\}$ can be successfully unified with any of the ground instances of $h\{var\ X\}$.

Therefore, the algorithm described here takes a different approach, in which a unification with *all* only yields a *necessary* set of constraints, not a *sufficient* set. The algorithm is thus *incomplete* (or "partial") in this respect.

The following decomposition rule is used, where the return value is either simply *True* or *False*, with the informal meaning "there might be a result" or "a result is precluded".

Decomposition Rule ***grouping***:

$$\frac{t^1 \preceq_u \text{ all } t^2}{(t^1 \preceq_u t^2) \neq \mathit{False}}$$

In the case where the constraint is reduced to *True*, it is possible that there is a result, but it is also possible that there is none, depending on the further evaluation of the variables in t^2.

5.1.8 Term References: Memoing of Previous Computations

Resolving References. References occurring in either term of a simulation constraint are dereferenced in a straightforward manner using the $deref(\cdot)$ function described above:
Decomposition Rule ***deref***:

$$\frac{\uparrow id \preceq_u t^2}{t^1 \preceq_u t^2} t^1 = deref(id) \qquad \frac{t^1 \preceq_u \uparrow id}{t^1 \preceq_u t^2} t^2 = deref(id)$$

Memoing. Dereferencing alone is not sufficient for treating references, because the simulation unification would not terminate in case both terms contain cyclic references. The technique used by the algorithm to avoid this problem is *memoing* (also known as *tabling*). In general, memoing is used to avoid redundant computations by storing the result of all previous computations in memory (e.g. in a table). If a computation has already been performed previously, it is not necessary to repeat it as the result can simply be retrieved from memory. This technique is among others used in certain implementations of Prolog [War92, CW96].

Consider for example the following (nave) implementation of the Fibonacci numbers in Haskell:

```
fib :: Int    Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Without memoing, this implementation performs many redundant computations.[5] For example, for the computation of $fib(n)$ it is necessary to compute $fib(n-1)$ and $fib(n-2)$, and for the computation of $fib(n-1)$ it is necessary to compute $fib(n-2)$ and $fib(n-3)$. Thus, $fib(n-2)$ needs to be computed twice. With memoing, the second computation could instead refer to the previous computation.

In Xcerpt, memoing for unification with references can be implemented by keeping for each conjunct in the disjunctive normal form a history of all previous applications of simplification rules (without their results) that were used for the creation of the conjunct. In every decomposition step it is then first verified whether the considered constraints have already been evaluated in a previous application of this simplification rule. If yes, the constraint reduces to *True*; if no, the computation is continued as usual.

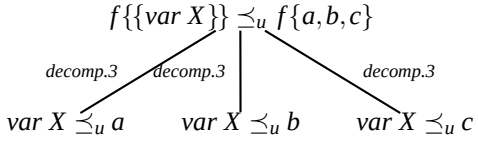
In the following rule, let $\mathscr{H}$ be a set of constraints that have been considered in previous applications of simplification rules in the current conjunct of the disjunctive normal form (history). Furthermore, t^1 is considered to be not of the form $desc\ t$.
Decomposition Rule ***memoing***:

$$\frac{desc\ t^1 \preceq_u t^2 \text{ such that } desc\ t^1 \preceq_u t^2 \in \mathscr{H}}{\mathit{False}} \qquad \frac{t^1 \preceq_u t^2 \text{ such that } t^1 \preceq_u t^2 \in \mathscr{H}}{\mathit{True}}$$

It might be somewhat surprising that the constraint is reduced to $True/False$ instead of inserting the result of a previous computation. The rationale behind this is that the result of the previous computation

[5] Note that Haskell's lazy evaluation performs a technique similar to memoing

Figure 3 Derivation tree of $f\{\{var\ X\}\} \preceq_u f\{a,b,c\}$ (Example 31, part 1). Different paths denote different alternatives, nodes represent conjuncts, and edges represent applications of simplification rules.



is already part of the current conjunct in the disjunctive normal form. *True* and *False* are the neutral elements of conjunction and disjunction, and thus terminate the unification while keeping results of previous computations. Examples 34 and 35 illustrate the simulation unification with references.

5.2 Examples

Since most examples for the decomposition rules are rather extensive, they are all grouped in this Section to improve readability. As in the examples in Section 3.2, the construct `optional` is sometimes abbreviated by `opt`, the construct `position` is sometimes abbreviated by `pos`, and the construct `without` is sometimes abbreviated by . The latter abbreviation is unproblematic, as can otherwise never occur within a term. Some of the more complicated examples also provide a "decomposition tree" which shows the application of decomposition steps in the different conjuncts of the DNF. In these trees, nodes represent conjuncts and edges represent decompositions. If applying a simplification rule to a conjunct yields a disjunction, its corresponding node has more than one alternative successors. Read from the root to the leaves, these trees allow to follow the sequences of decomposition steps that lead to substitutions. The consistent end states of the constraint store are often emphasised by a rectangular frame.

Example 31 (Decomposition)
This example consists of three decompositions of simple simulation constraints. Figures 3, 4, and 5 provide a graphical illustration of the decompositions.

1. Consider the simulation constraint (cf. Figure 3)

   $$C = f\{\{var\ X\}\} \preceq_u f\{a,b,c\}$$

   Applying the decomposition rule *decomp.3* with three different mappings $\pi \in \Pi$ to this simulation constraint yields

   $$var\ X \preceq_u a \ \vee\ var\ X \preceq_u b \ \vee\ var\ X \preceq_u c$$

   No further simplification rules are applicable.

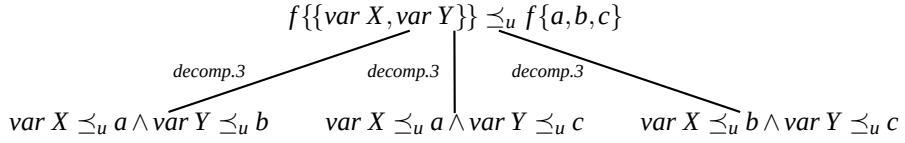
2. Consider the simulation constraint (cf. Figure 4)

   $$C = f[[var\ X, var\ Y]] \preceq_u f[a,b,c]$$

   Note the partial, ordered term specification of the left term. Decomposition with rule *decomp.3* and the three different index monotonic mappings $\pi \in \Pi_{mon}$ yields

   $$\begin{array}{ll} & var\ X \preceq_u a \wedge var\ Y \preceq_u b \\ \vee & var\ X \preceq_u a \wedge var\ Y \preceq_u c \\ \vee & var\ X \preceq_u b \wedge var\ Y \preceq_u c \end{array}$$

Figure 4 Derivation tree of $f[[var\ X, var\ Y]] \preceq_u f[a,b,c]$ (Example 31, part 2). Different paths denote different alternatives, nodes represent conjuncts, and edges represent applications of simplification rules.



3. Consider the simulation constraint (cf. Figure 5)

$$C = f\{\{var\ X \to b\}\} \preceq_u f\{a,b,c\}$$

As both terms are unordered, decomposition rule *decomp.3* with the three different $\pi \in \Pi$ yields

$$var\ X \to b \preceq_u a \ \vee\ var\ X \to b \preceq_u b \ \vee\ var\ X \to b \preceq_u c$$

Decomposition of the $\to$ construct reduces the constraint store to

$$\begin{array}{ll} & b \preceq_u a \ \wedge\ var\ X \preceq_u a \ \wedge\ b \preceq_u var\ X \\ \vee & b \preceq_u b \ \wedge\ var\ X \preceq_u b \ \wedge\ b \preceq_u var\ X \\ \vee & b \preceq_u c \ \wedge\ var\ X \preceq_u c \ \wedge\ b \preceq_u var\ X \end{array}$$

Simulation unification in all three conjuncts yields

$$\begin{array}{ll} & False \ \wedge\ var\ X \preceq_u a \ \wedge\ b \preceq_u var\ X \\ \vee & True \ \wedge\ var\ X \preceq_u b \ \wedge\ b \preceq_u var\ X \\ \vee & False \ \wedge\ var\ X \preceq_u c \ \wedge\ b \preceq_u var\ X \end{array}$$

and formula simplification simplifies this constraint store to

$$var\ X \preceq_u b \ \wedge\ b \preceq_u var\ X$$

Example 32 (Simulation Unification with `without`)

1. Consider

$$C = f\{\{a, without\ b\}\} \ \preceq_u\ f\{a,c\}$$

The set Π of partial mappings that are total on $SubT^+$ is as follows (partial mappings completed by mapping undefined values to $\bot$)[6]:

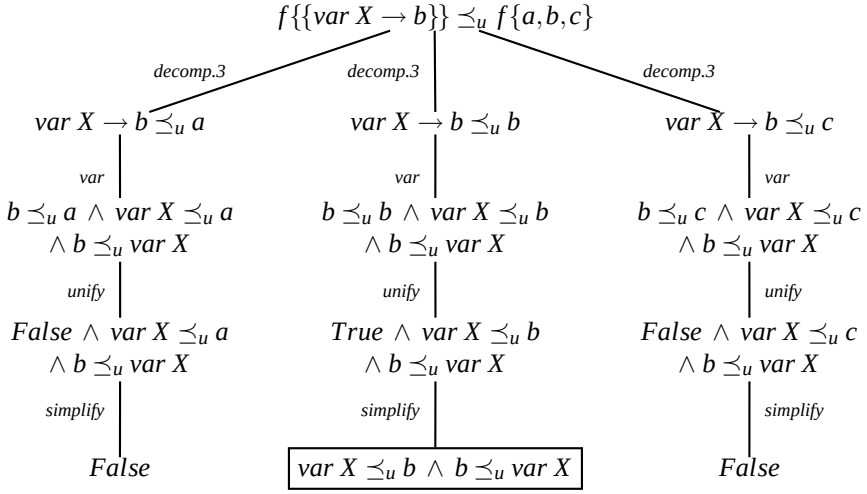
$$\begin{array}{ll} \{a \mapsto a, b \mapsto \bot\} & \{a \mapsto c, b \mapsto \bot\} \\ \{a \mapsto a, b \mapsto c\} & \{a \mapsto c, b \mapsto a\} \end{array}$$

From this set, the constraint C is decomposed into the following constraint formula (using the decomposition rule for terms containing `without`):

$$\begin{array}{ll} & a \preceq_u a \wedge (b \preceq_u \bot \vee b \preceq_u c) \\ \vee & a \preceq_u c \wedge (b \preceq_u \bot \vee b \preceq_u a) \end{array}$$

[6] note that *without b* is abbreviated by *b*

Figure 5 Derivation tree of $f\{\{var\ X \to b\}\} \preceq_u f\{a,b,c\}$ (Example 31, part 3). Different paths denote different alternatives, nodes represent conjuncts, and edges represent applications of simplification rules.



Note that $t \preceq_u \bot$ always evaluates to $False$. Evaluating the constraints contained in the negated subformulas yields:

$$\begin{array}{ll} & a \preceq_u a \wedge (False \vee False) \\ \vee & a \preceq_u c \wedge (False \vee False) \end{array}$$

and formula simplification results in

$$a \preceq_u a \vee a \preceq_u c$$

which of course can be further decomposed to $True$.

2. Consider $C = f\{\{a, without\ b\}\} \preceq_u f\{a,b\}$

   The set Π of partial mappings that are total on $SubT^+$ is as follows (completed by mapping all terms on which the mappings are undefined to $\bot$):

$$\begin{array}{ll} \{a \mapsto a, b \mapsto \bot\} & \{a \mapsto b, b \mapsto \bot\} \\ \{a \mapsto a, b \mapsto b\} & \{a \mapsto b, b \mapsto b\} \end{array}$$

   From this set, the constraint C is decomposed into the following constraint formula (using the decomposition rule for terms containing `without`):

$$\begin{array}{ll} & a \preceq_u a \wedge (b \preceq_u \bot \vee b \preceq_u b) \\ \vee & a \preceq_u b \wedge (b \preceq_u \bot \vee b \preceq_u a) \end{array}$$

   Evaluating the constraints contained in the negated subformulas yields:

$$\begin{array}{ll} & a \preceq_u a \wedge (False \vee True) \\ \vee & a \preceq_u c \wedge (False \vee False) \end{array}$$

   and formula simplification results in

$$a \preceq_u c$$

   which of course can be further decomposed to $False$.

Example 33 (Simulation Unification with `optional`)
Consider the constraint $C = f[[a, \text{opt } g\{var\ X\}, \text{opt } h\{var\ Y\}\ ]] \preceq_u f[a, g\{b\}]$

The set Π_{mon} of partial, index monotonic mappings that are total on $SubT^!$ (the non-optional subterms of the left term) is as follows (partial mappings are completed by mapping undefined values to $\bot$):

$$\Pi_{mon} = \{ \begin{array}{lll} \{a \mapsto a, & \text{opt } g\{var\ X\} \mapsto \bot, & \text{opt } h\{var\ Y\} \mapsto \bot\} \\ \{a \mapsto a, & \text{opt } g\{var\ X\} \mapsto g\{b\}, & \text{opt } h\{var\ Y\} \mapsto \bot\} \\ \{a \mapsto a, & \text{opt } g\{var\ X\} \mapsto \bot, & \text{opt } h\{var\ Y\} \mapsto g\{b\}\} \\ \{a \mapsto g\{b\}, & \text{opt } g\{var\ X\} \mapsto \bot, & \text{opt } h\{var\ Y\} \mapsto \bot\} \end{array} \}$$

From this set, the constraint C is decomposed into the following constraint formula (using the decomposition rule for terms containing `optional`). The construct `optional` is already eliminated using the helper rule described above:

$$\begin{array}{lll} & a \preceq_u a \wedge & (\ g\{var\ X\} \preceq_u \bot \ \vee \ h\{var\ Y\} \preceq_u \bot \vee \\ & & \ \ g\{var\ X\} \preceq_u g\{b\} \ \vee \ h\{var\ Y\} \preceq_u \bot \vee \\ & & \ \ g\{var\ X\} \preceq_u \bot \ \vee \ h\{var\ Y\} \preceq_u g\{b\} \ ) \\ \vee & a \preceq_u a \wedge g\{var\ X\} \preceq_u g\{b\} \wedge & (\ h\{var\ Y\} \preceq_u \bot \ ) \\ \vee & a \preceq_u a \wedge h\{var\ Y\} \preceq_u g\{b\} \wedge & (\ g\{var\ X\} \preceq_u \bot \ ) \\ \vee & a \preceq_u g\{b\} \wedge & (\ g\{var\ X\} \preceq_u \bot \ \vee \ h\{var\ Y\} \preceq_u \bot \ ) \end{array}$$

Note that $t \preceq_u \bot$ always evaluates to *False*. Evaluating the constraints contained in the negated subformulas yields:

$$\begin{array}{lll} & a \preceq_u a \wedge & (\ \mathit{False} \ \vee \ \mathit{False} \vee \\ & & \ \ \mathit{var}\ X \preceq_u b \ \vee \ \mathit{False} \vee \\ & & \ \ \mathit{False} \ \vee \ \mathit{False} \ ) \\ \vee & a \preceq_u a \wedge g\{var\ X\} \preceq_u g\{b\} \wedge & (\ \mathit{False} \ ) \\ \vee & a \preceq_u a \wedge h\{var\ Y\} \preceq_u g\{b\} \wedge & (\ \mathit{False} \ ) \\ \vee & a \preceq_u g\{b\} \wedge & (\ \mathit{False} \ \vee \ \mathit{False} \ ) \end{array}$$

Formula simplification and application of consistency rule 5 (negation) yields

$$\begin{array}{lll} & a \preceq_u a \wedge & \mathit{False} \\ \vee & a \preceq_u a \wedge g\{var\ X\} \preceq_u g\{b\} \wedge & \mathit{True} \\ \vee & a \preceq_u a \wedge h\{var\ Y\} \preceq_u g\{b\} \wedge & \mathit{True} \\ \vee & a \preceq_u g\{b\} \wedge & \mathit{True} \end{array}$$

Note that reducing the first line to *False* informally states "the mapping is completable", whereas the *True* values in lines 2–4 state that "the mapping is not completable" (because the right term only contains two subterms and the mapping needs to be injective). After further decomposition and simplification steps, this formula is simplified to $\mathit{var}\ X \preceq_u b$ (as desired).

Example 34 (Simulation Unification with References)
Consider the simulation constraint

$$C = f\{\{o1@g\{\{var\ X \rightarrow\uparrow o1\}\}\ \}\} \preceq_u f\{g\{a\}, o2@g\{b, \uparrow o2\}\}$$

In the following, the sequence of decomposition steps that result in a complete simulation unification of the simulation constraint is described. For each conjunct, the set $\mathscr{H}_i$ denotes the current memoing

history of the conjunct. So as to better distinguish the path that lead to this history, the index is composed of the numbers of the branches followed in previous steps. For example, $\mathscr{H}_1 21$ is the history of the node that can be located by following the first branch on the top level, the second branch on the second level, and the first branch on the third level. Note that Figure 6 gives a graphical representation of the decomposition tree that might be easier to read. In this tree, the history of a node is easily determined by following the path from the root node to the current node, and thus not given explicitly. The first decomposition step yields

$$\begin{array}{ll} & o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u g\{a\} \qquad \mathscr{H}_1 = \{C\} \\ \vee & o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u o2@g\{b, \uparrow o2\} \qquad \mathscr{H}_2 = \{C\} \end{array}$$

Note that the $\mathscr{H}_i$ denote the history for every conjunct, and is in this step the same for both conjuncts, as they "share the same history". Further decomposition results in

$$\begin{array}{ll} & var\ X \rightarrow\uparrow o1 \preceq_u a \qquad \mathscr{H}_{11} = \mathscr{H}_1 \cup \{o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u g\{a\}\ \} \\ \vee & var\ X \rightarrow\uparrow o1 \preceq_u b \qquad \mathscr{H}_{21} = \mathscr{H}_2 \cup \{o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u o2@g\{b, \uparrow o2\}\} \\ \vee & var\ X \rightarrow\uparrow o1 \preceq_u \uparrow o2 \qquad \mathscr{H}_{22} = \mathscr{H}_2 \cup \{o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u o2@g\{b, \uparrow o2\}\} \end{array}$$

Application of the $\rightsquigarrow$ decomposition in all three conjuncts yields

$$\begin{array}{ll} & \uparrow o1 \preceq_u a \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u a \qquad \mathscr{H}_{111} = \mathscr{H}_{11} \cup \{var\ X \rightarrow\uparrow o1 \preceq_u a\} \\ \vee & \uparrow o1 \preceq_u b \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u var\ X \preceq_u b \qquad \mathscr{H}_{211} = \mathscr{H}_{21} \cup \{var\ X \rightarrow\uparrow o1 \preceq_u b\} \\ \vee & \uparrow o1 \preceq_u \uparrow o2 \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u \uparrow o2 \qquad \mathscr{H}_{221} = \mathscr{H}_{22} \cup \{var\ X \rightarrow\uparrow o1 \preceq_u \uparrow o2\} \end{array}$$

In the next step, $o1$ is dereferenced to $o1@g\{\{var\ X \rightarrow\uparrow o1\}\}$ in all conjuncts. This gives the result:

$$\begin{array}{ll} & o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u a \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u a \qquad \mathscr{H}_{1111} = \mathscr{H}_{111} \cup \{\uparrow o1 \preceq_u a\} \\ \vee & o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u b \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u b \qquad \mathscr{H}_{2111} = \mathscr{H}_{211} \cup \{\uparrow o1 \preceq_u b\} \\ \vee & o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u \uparrow o2 \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u \uparrow o2 \qquad \mathscr{H}_{2211} = \mathscr{H}_{221} \cup \{\uparrow o1 \preceq_u \uparrow o2\} \end{array}$$

Decomposition in the first two conjuncts and dereferencing of $o2$ in the third conjunct then yields:

$$\begin{array}{ll} & False \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u a \\ & \qquad \mathscr{H}_{11111} = \mathscr{H}_{1111} \cup \{o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u a\} \\ \vee & False \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u b \\ & \qquad \mathscr{H}_{21111} = \mathscr{H}_{2111} \cup \{o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u b\} \\ \vee & o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u o2@g\{b, \uparrow o2\} \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u o2@g\{b, \uparrow o2\} \\ & \qquad \mathscr{H}_{22111} = \mathscr{H}_{2211} \cup \{o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u \uparrow o2, var\ X \preceq_u \uparrow o2\} \end{array}$$

The next step eliminates the first two conjuncts because they contain *False*. In the third conjunct, the *memoing* rule is applicable to the first simulation constraint: $o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u o2@g\{b, \uparrow o2\} \in \mathscr{H}_{22} \subseteq \mathscr{H}_{22111}$. It thus reduces to *True* and terminates the otherwise infinite computation:

$$True \ \wedge\ \uparrow o1 \preceq_u var\ X \ \wedge\ var\ X \preceq_u var\ X \preceq_u o2@g\{b, \uparrow o2\} \qquad \mathscr{H}_{221111} = \mathscr{H}_{22111}$$

Now the second occurrence of $o1$ can be dereferenced. The following constraint store is the result of the simulation unification:

$$\begin{array}{l} o1@g\{\{var\ X \rightarrow\uparrow o1\}\} \preceq_u var\ X \ \wedge\ var\ X \preceq_u var\ X \preceq_u o2@g\{b, \uparrow o2\} \\ \qquad \mathscr{H}_{2211111} = \mathscr{H}_{221111} \cup \{\uparrow o1 \preceq_u var\ X\} \end{array}$$

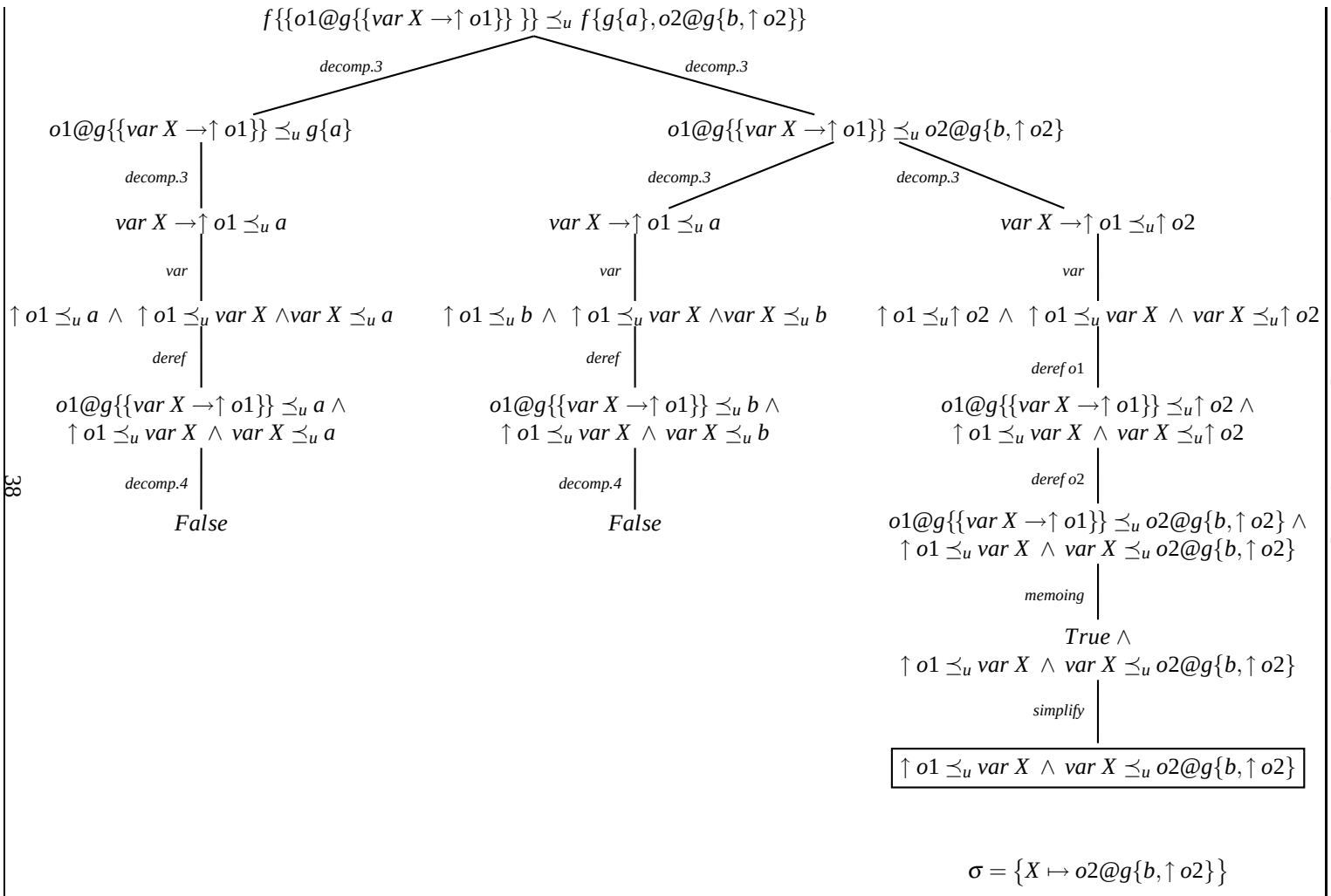
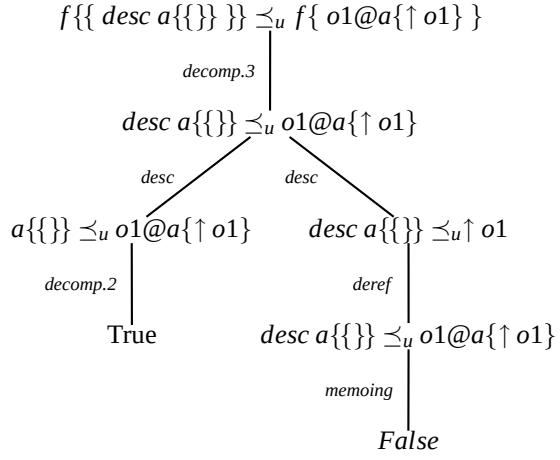


Figure 6 Derivation tree of $f\{\{o1@g\{\{var\ X \to\uparrow o1\}\}\} \preceq_u f\{g\{a\}, o2@g\{b,\uparrow o2\}\}$ (Example 34). The memoing history $\mathscr{H}$ of a node is represented by the path from the root to that node.

Figure 7 Derivation tree of $f\{\{\ desc\ a\{\{\}\}\ \}\} \preceq_u f\{\ o1@a\{\uparrow o1\}\ \}$ (Example 35). In this graph, the memoing history $\mathscr{H}$ of a node is represented by the path from the root to that node.



Example 35 (Simulation Unification with References and Descendant)
Consider the simulation constraint

$$C = f\{\{\ desc\ a\{\{\}\}\ \}\} \preceq_u f\{\ o1@a\{\uparrow o1\}\ \}$$

The sequence of decomposition steps is as follows (cf. Figure 7 for a graphical illustration). The first decomposition step (*decomp.3*) yields

$$desc\ a\{\{\}\} \preceq_u o1@a\{\uparrow o1\} \qquad \mathscr{H}_1 = \{C\}$$

Application of the descendant decomposition splits the constraint store into two conjuncts as follows:

$$\begin{array}{ll} & a\{\{\}\} \preceq_u o1@a\{\uparrow o1\} \qquad \mathscr{H}_{11} = \mathscr{H}_1 \cup \{desc\ a\{\{\}\} \preceq_u o1@a\{\uparrow o1\}\} \\ \vee & desc\ a\{\{\}\} \preceq_u \uparrow o1 \qquad \mathscr{H}_{12} = \mathscr{H}_1 \cup \{desc\ a\{\{\}\} \preceq_u o1@a\{\uparrow o1\}\} \end{array}$$

Decomposition in the first conjunct yields *True*, and in the second conjunct, $o1$ can be dereferenced:

$$\begin{array}{ll} & True \qquad \mathscr{H}_{111} = \mathscr{H}_{11} \cup \{a\{\{\}\} \preceq_u o1@a\{\uparrow o1\}\} \\ \vee & desc\ a\{\{\}\} \preceq_u o1@a\{\uparrow o1\} \qquad \mathscr{H}_{121} = \mathscr{H}_{12} \cup \{desc\ a\{\{\}\} \preceq_u \uparrow o1\} \end{array}$$

As $desc\ a\{\{\}\} \preceq_u o1@a\{\uparrow o1\} \in \mathscr{H}_1 \subseteq \mathscr{H}_{121}$, the memoing rule is applicable and reduces the second conjunct to *False*, and the process terminates as no more rule is applicable.

$$\begin{array}{ll} & True \qquad \mathscr{H}_{1111} = \mathscr{H}_{111} \\ \vee & False \qquad \mathscr{H}_{1211} = \mathscr{H}_{121} \end{array}$$

5.3 Soundness and Completeness

The following theorem shows soundness and completeness for the simulation unification algorithm applied to a simulation constraint of the form $t^q \preceq_u t^c$. t^q is assumed to not contain subterm negation

or optional subterms. Also, as rules with grouping constructs are always evaluated in an auxiliary computation using the dependency constraint, it is assumed that t^c does not contain grouping constructs. Furthermore, t^c is assumed not to contain functions, aggregations or optional subterms.

Theorem 36 (Soundness and Completeness of Simulation Unification)
Let t^q be a query term without subterm negation and optional subterms and let t^c be a construct term without grouping constructs, functions/aggregations, and optional subterms. A substitution set Σ is a most general simulation unifier of t^q and t^c if and only if the simulation unification of $t^q \preceq_u t^c$ terminates with a constraint store CS such that $\Sigma = \Omega(CS)$.

We first show that simulation unification terminates for any query term t^q and construct term t^c, and then show soundness and completeness by induction over the number of rule applications.

Lemma 37 (Termination of Simulation Unification)
Let t^q be a query term without subterm negation and optional subterms and let t^c be a construct term without grouping constructs, functions/aggregations, and optional subterms. Simulation unification of $t^q \preceq_u t^c$ terminates.

Proof. We prove termination by assigning a rank to atomic constraints and showing that the rank decreases with every rule application. Consider a tree where each node is an atomic constraint (i.e. either a boolean or a simulation constraint). Application of a simulation unification rule yields the constraints that are successors of this node. Conjunctions and disjunctions split into several successors. For example, application of *decomp.3* to a simulation constraint of the form $f\{a,b\} \preceq_u f\{c,d\}$ yields the successor nodes $a \preceq_u c$, $a \preceq_u d$, $b \preceq_u c$, and $b \preceq_u d$. By Knig's Lemma, it suffices to show that every successor of a node has a strictly lower rank than its predecessor. Ranks of constraints are defined as follows:

$$
\begin{array}{lcllcl}
rank(True) & = & 0 & depth(l\{t_1,\ldots,t_n\}) & = & 1+max_{i=1}^{n}(depth(t_i)) \\
rank(False) & = & 0 & depth(l\{\{t_1,\ldots,t_n\}\}) & = & 1+max_{i=1}^{n}(depth(t_i)) \\
rank(t_1 \preceq_u t_2) & = & depth(t_1)+depth(t_n) & depth(l[t_1,\ldots,t_n]) & = & 1+max_{i=1}^{n}(depth(t_i)) \\
 & & & depth(l[[t_1,\ldots,t_n]]) & = & 1+max_{i=1}^{n}(depth(t_i)) \\
depth(var\ X) & = & 1 & depth(desc\ t) & = & 1+depth(t) \\
depth(var\ X \preceq_u t) & = & 1+depth(t) & depth(id@t) & = & 1+depth(t)
\end{array}
$$

Furthermore, $depth(\uparrow id)$ is defines as $(n+1)\cdot depth(t)$, where n is the number of remaining applications of the *deref* rule to $\uparrow id$ in the course of the evaluation, and t is the referenced term. Obviously, n is finite because the memoing rule eventually terminates a path when a pair of terms is unified that has already been considered. Since there are only finitely many subterms in each term, this happens inevitably in every computation that would otherwise not terminate.

1. application of *decomp.1*, *decomp.2*, or *decomp.4*.
   The rank trivially decreases, because all three kinds of rules reduce the constraint store to either *True* or *False*.
2. application of *decomp.3*
   A constraint of the form $t^q \preceq_u t^c$ where $t^q = l\{\{t_1^1,\ldots,t_m^1\}\}$ and $t^c = l\{t_1^2,\ldots,t_n^2\}$ (independent of the kinds of braces) is reduced to finitely many successors of the form $t_i^1 \preceq_u t_j^2$ for some children t_i^1 of t^q and t_j^2 of t^c. Let t_i^1 and t_j^2 be any such children. Obviously, $depth(t_i^1) < depth(t^q)$ and $depth(t_j^2) < depth(t^c)$. Then, $rank(t_i^1 \preceq_u t_j^2) < rank(t^q \preceq_u t^c)$.
3. application of *var*
   A constraint of the form $var\ X \rightsquigarrow t \preceq_u t^c$ is reduced to three successors:
   - $rank(t \preceq_u t^c) = depth(t)+depth(t^c) < (1+depth(t))+depth(t^c) = rank(t^q \preceq_u t^c)$
   - $rank(var\ X \preceq_u t^c) = 1+depth(t^c) < (1+depth(t))+depth(t^c) = rank(t^q \preceq_u t^c)$, as $depth(t) \geq 1$

- $rank(t \preceq_u var\ X) = depth(t)+1 < (1+depth(t))+depth(t^c) = rank(t^q \preceq_u t^c)$, as $depth(t^c) \geq 1$

4. application of *desc*

   A constraint of the form $desc\ t \preceq_u t^c$ where $t^c = l\{t_1^2,\ldots,t_n^2\}$ is reduced to two kinds of successors:

   - $rank(t \preceq_u t^c) = rank(t)+rank(t^c) < (1+rank(t))+rank(t^c) = rank(desc\ t \preceq_u t^c)$
   - $rank(desc\ t \preceq_u t_i^2) = 1+rank(t)+rank(t_i^2) < 1+rank(t)+rank(t^c) = rank(desc\ t \preceq_u t^c)$ for some $1 \leq i \leq n$

5. application of *deref*

   A constraint of the form $\uparrow id \preceq_u t$ is reduced to $deref(id) \preceq_u t$. Let n be the number of remaining applications of the dereferencing rule

   $$rank(\uparrow id \preceq_u t) = (n+1)\cdot depth(deref(id))+depth(t) > depth(deref(id))+depth(t) = rank(deref(id) \preceq_u t)$$

   because if $\uparrow id$ occurs in $deref(id)$, then the *deref* rule is only applicable $n-1$ times and thus the rank is strictly lower.

6. application of *memoing*

   A constraint of the form $t^q \preceq_u t^c$ is reduced to *True* or *False* in case it has already been considered. Since $rank(True) = rank(False) = 0$, the rank is trivially reduced to a lower value.

Proof of Theorem 36.

We prove theorem 36 by induction over the number k of applications of decomposition rules to the constraint store C initialised by $C = t^q \preceq_u t^c$. In every case, it is to show that $\Omega(C)$ is the most general simulation unifier of t^q in t^c.

Since t^c does not contain grouping constructs, we know that every $\in /_{\downarrow_{FV(t^c)}}$ consists of a single substitution. This simplifies matters significantly, as it requires that a substitution set Σ is a simulation unifier only if for all $\sigma \in \Sigma$ holds that $\sigma(t^q) \preceq \sigma(t^c)$ (*).

Induction Base. Let $k = 0$, i.e. no rules are applicable. We have to consider two cases:

1. C is of the form $var\ X \preceq_u t^c$ for a variable X and a construct term t.

   By definition, $\Omega(C)$ contains exactly the substitutions σ where $\sigma(X) = t'$ s.t. $t' \cong \sigma(t^c)$. Obviously, $\Omega(C)$ is a simulation unifier of t^q in t^c.

   $\Omega(C)$ is also the most general simulation unifier of t^q in t^c. Assume it was not. Then there exists $\Sigma \not\subseteq \Omega(C)$ s.t. Σ is a simulation unifier of t^q in t^c, i.e. (with *) for every $\sigma \in \Sigma$ holds that $t^{q'} = \sigma(t^q) = \sigma(X)$ simulates into $t^{c'} = \sigma(t^c)$. Let now $\sigma \in \Sigma$ and let $t^{q'} = \sigma(X)$ be one of the ground instances of t^q s.t. $\sigma \notin \Omega(C)$, but $t^{q'}$ simulates into the ground instance of t^c in σ. Because Σ is a simulation unifier and thus an all-grounding substitution set, $t^{q'}$ is a data term. By definition of $\cong$, it thus holds that $t^{q'} \cong t^c$. Contradiction with $t^{q'} \notin \Omega(C)$ $\lightning$

2. C is of the form $t^q \preceq_u var\ X$ for a variable X and a query term t.

   By definition, $\Omega(C)$ contains exactly the substitutions σ where $\sigma(X) = t'$ s.t. $\sigma(t^q) \preceq t'$. Obviously, $\Omega(C)$ is a simulation unifier of t^q in t^c.

   $\Omega(C)$ is also the most general simulation unifier of t^q in t^c. Assume it was not. Then there exists $\Sigma \not\subseteq \Omega(C)$ s.t. Σ is a simulation unifier of t^q in t^c, i.e. (with *) for every $\sigma \in \Sigma$ holds that $t^{q'} = \sigma(t^q)$ simulates into $t^{c'} = \sigma(t^c) = \sigma(X)$. Let now $\sigma \in \Sigma$ and let $t^{q'} = \sigma(t^q)$ be one of the ground instances of t^q s.t. $\sigma \notin \Omega(C)$, but $t^{q'}$ simulates into the ground instance $t^{c'}$ of t^c in σ. Then it holds that $\sigma(t^q) \preceq \sigma(X)$, and thus σ is in $\Omega(C)$.

Induction Step. Assume now that the number of decomposition steps is k. By induction hypothesis, Theorem 36 holds for all $i < k$. We have to consider the following cases:

1. application of *decomp.1* (brace incompatibility)

let $t^q = l[t_1^1,\ldots,t_m^1]$ and $t^c = l\{t_1^2,\ldots,t_n^2\}$
or let $t^q = l[[t_1^1,\ldots,t_m^1]]$ and $t^c = l\{t_1^2,\ldots,t_n^2\}$

As the braces of t^q and t^c are incompatible, ground instances of t^q will not simulate in ground instances of t^c regardless of the substitutions. Thus, the mgsu of t^q in t^c, defined as the union of all simulation unifiers, is empty. *decomp.1* reduces both cases to the constraint store $False$. By definition, $\Omega(False) = \{\ \}$, and thus the theorem is correct.

2. application of *decomp.2* (left term without subterms)

   - let $t^q = l\{\{\ \}\}$ and $t^c = l\{t_1^2,\ldots,t_n^2\}$ or
     let $t^q = l\{\{\ \}\}$ and $t^c = l[t_1^2,\ldots,t_n^2]$ or
     let $t^q = l[[\ ]]$ and $t^c = l[t_1^2,\ldots,t_n^2]$ and $n \geq 1$

     Then t^q simulates in t^c for every grounding substitution set of t^c. Thus, the mgsu of t^q in t^c is the set of all all-grounding substitutions. *decomp.2* reduces all three cases to $True$, and with the definition of $\Omega(True)$ as the set of all all-grounding substitutions, the theorem is correct.
   - let $t^q = l\{\ \}$ and $t^c = l\{t_1^2,\ldots,t_n^2\}$ or
     let $t^q = l\{\ \}$ and $t^c = l[t_1^2,\ldots,t_n^2]$ or
     let $t^q = l[\ ]$ and $t^c = l[t_1^2,\ldots,t_n^2]$ and $n \geq 1$

     Then t^q never simulates in ground instances of t^c, because there exists no index bijective function from $\langle\ \rangle$ to $\langle t_1^2,\ldots,t_n^2\rangle$ for $n \geq 1$. Thus, the mgsu of t^q in t^c, defined as the union of all simulation unifiers, is empty. *decomp.2* reduces all three cases to the constraint store $False$. By definition, $\Omega(False) = \{\ \}$, and thus the theorem is correct.
   - let $t^q = l\{\ \}$ and $t^c = l\{\ \}$ or
     let $t^q = l\{\ \}$ and $t^c = l[\ ]$ or
     let $t^q = l[\ ]$ and $t^c = l[\ ]$

     Then t^q simulates in t^c for every substitution set. Thus, the mgsu of t^q in t^c is the set of all all-grounding substitutions. *decomp.2* reduces all three cases to $True$, and with the definition of $\Omega(True)$ as the set of all all-grounding substitutions, the theorem is correct.

3. application of *decomp.3* (general decomposition)

   Let $t^q = l\{\{t_1^1,\ldots,t_m^1\}\}$ and let $t^c = l\{t_1^2,\ldots,t_n^2\}$.

   The mgsu of t^q in t^c is the set Σ of all all-grounding substitutions σ such that $\sigma(t^q) \preceq \sigma(t^c)$. According to Definition 17, it thus holds that there exists a total, index injective, and position preserving mapping π from $SubT(\sigma(t^q)) = \langle t_1^1,\ldots,t_m^1\rangle$ to $SubT(\sigma(t^c)) = \langle t_1^2,\ldots,t_m^2\rangle$ such that for each $t_i^1 \in SubT(\sigma(t^q))$ holds that $t_i^1 \preceq \sigma(t_i^1)$, and Σ consists of all such σ.

   Application of *decomp.3* to $t^q \preceq_u t^c$ yields $C = \bigvee_{\pi\in\Pi_{pp}} \bigwedge_{1\leq i\leq m} t_i^1 \preceq_u \pi(t_i^1)$. Thus, as by definition, $\Omega(C) = \Omega(\bigvee C') = \bigcup \Omega(C')$, $\Omega(C)$ substitutions for all possible total, index injective, and position preserving functions π. Consider now some $C' = \bigwedge_{1\leq i\leq m} t_i^1 \preceq_u \pi(t_i^1)$ for some mapping π. By definition, we know that $\Omega(C') = \bigcap_{1\leq i\leq n} \Omega(t_i^1 \preceq_u \pi(t_i^1))$, and by induction hypothesis, each $\Omega(t_i^1 \preceq_u \pi(t_i^1))$ is the most general simulation unifier of t_i^1 in $\pi(t_i^1)$. $\Omega(C')$ is thus the maximal all-grounding substitution set that is a simulation unifier for each of the t_i^1 in $\pi(t_i^1)$. Thus, $\Omega(C) = \bigcup \Omega(C')$ is the maximal all-grounding set that is a simulation unifier for any of the mappings π, and as the labels of t^q and t^c match, $\Omega(C)$ is the most general simulation unifier of t^q in t^c.

   The argumentation is identical in the other cases with the exception of the chosen set of functions Π, which is obviously correct.

4. application of *decomp.4* (label mismatch)

   Let t^q and t^c be terms such that the labels mismatch. Hence, ground instances of t^q will not simulate in ground instances of t^c regardless of the substitutions. Thus, the mgsu of t^q in t^c, defined as the union of all simulation unifiers, is empty. *decomp.1* reduces $t^q \preceq_u t^c$ to the constraint store $False$. By definition, $\Omega(False) = \{\ \}$, and thus the theorem is correct.

5. application of *var* ($\rightsquigarrow$ elimination)

   Let $t^q = \mathit{var}\ X \rightsquigarrow t^1$ and let $t^c = t^2$.

   An all-grounding substitution set Σ has to satisfy the following conditions to be a simulation unifier of t^q in t^c:

   (a) Σ must be applicable to $\mathit{var}\ X \rightsquigarrow t^1$, i.e. it may only contain substitutions σ for which holds that $\sigma(t^1) \preceq \sigma(X)$

   (b) it must be a simulation unifier of $\mathit{var}\ X$ in t^2, i.e. for every substitution set σ in Σ holds that $\sigma(X) \preceq \sigma(t^2)$

   We now show that the evaluation of the rule *var* satisfies both conditions and is maximal, i.e. a most general simulation unifier of t^q in t^c. *var* reduces $t^q \preceq_u t^c$ to a constraint store $CS = t^1 \preceq_u t^2 \wedge t^1 \preceq_u X \wedge X \preceq_u t^2$. By definition,

$$\Omega(CS) = \underbrace{\Omega(t^1 \preceq_u t^2)}_{A} \cap \underbrace{\Omega(t^1 \preceq_u X)}_{B} \cap \underbrace{\Omega(X \preceq_u t^2)}_{C}$$

   - B is the mgsu of t^1 in $\mathit{var}\ X$; thus, for every $\sigma \in B$ holds that $\sigma(t^1) \preceq \sigma(X)$
   - C is the mgsu of $\mathit{var}\ X$ and $\sigma(t^1)$

   $B \cap C$ describes exactly the mgsu of t^q in t^c, because it fulfils the requirements (1) and (2) given above and is maximal, because B and C are maximal.

   As, by induction hypothesis, $t^1 \preceq_u t^2$ computes the mgsu of t^1 in t^2, $A \cap B \cap C = B \cap C$ (i.e. $t^1 \preceq_u t^2$ does not remove further substitutions from $B \cap C$). Note that this corresponds to the fact that $t^1 \preceq_u t^2$ is merely used to improve the evaluation performance.

   Thus, the theorem is correct for this case.

6. application of *desc* (descendant elimination)

   Let $t^q = \mathit{desc}\ t$, and let $t^c = l\{t_1^2, \ldots, t_n^2\}$ or $t^c = l[t_1^2, \ldots, t_n^2]$ ($n \geq 0$).

   A substitution set Σ is then a simulation unifier if for every $\sigma \in \Sigma$ holds that there exists a subterm $t^{c'}$ of $\sigma(t^c)$ such that $\sigma(t) \preceq t^{c'}$, and it is the mgsu, if it is the union of all all-grounding simulation unifiers that adhere to this restriction.

   Application of the rule *desc* reduces the constraint $t^q \preceq_u t^c$ to $C = t \preceq_u t^c \vee \bigvee_{1 \leq i \leq n} \mathit{desc}\ t \preceq_u t_i^2$. Thus,

$$\Omega(C) = \underbrace{\Omega(t \preceq_u t^c)}_{A} \cup \underbrace{\bigcup_{1 \leq i \leq n} \Omega(\mathit{desc}\ t \preceq_u t_i^2)}_{B}$$

   By induction hypothesis, A is the mgsu of $t \preceq_u t^c$, and B is the union of the mgsus of $t^q \preceq_u t_i^2$ for some subterm t_i^2 of t^c. By Definition 17, $\Omega(C)$ is thus the maximal set of all-grounding substitutions that is a simulation unifier of t^q in t^c and thus the mgsu.

7. application of *memoing* (termination in case of constraints that have already been treated)

   It suffices to consider the rule *memoing*; the rule *deref* is trivially correct, it simply implements the definition of dereferencing in ground query term graphs.

   In the following, let t^c be some construct term of the forms $id@l\{t_1^2, \ldots, t_n^2\}$ or $id@l[t_1^2, \ldots, t_n^2]$ such that at least one of the t_i^2 contains a reference to id, i.e. t^c contains at least one cycle. It is not necessary to consider other t^c without identifiers or without cycles, because the theorem holds for these as shown in the rest of this proof.

   We already know that simulation unification is sound and complete for all rule applications besides *memoing*. We have to show that the *memoing* rules have no influence on the resulting set of all-grounding substitutions, i.e. with memoing, we get the same result as without memoing (and infinite application of decomposition rules).

- let $t^q = \mathit{desc}\ t$; a substitution set Σ is the mgsu of t^q in t^c, if it contains exactly the substitutions σ for which holds that $\sigma(t^q) \preceq \sigma(t^c)$.

  Evaluation of $C = t^q \preceq_u t^c$ for the first time yields $C = t \preceq_u t^c \vee \bigvee_{1\leq i\leq n} \mathit{desc}\ t \preceq_u t_i^2$ by applying the rule *desc*. Assume that further evaluation of C eventually yields a constraint store (in DNF) of the form $C_1 \vee \cdots \vee C_i \vee \cdots \vee C_m$ for some $m \geq 1$, and that C_i again is of the form $t^q \preceq_u t^c$, because the $\mathit{desc}\ t \preceq_u t_j^2$ leading to C_i contains a cyclic reference to id. Evaluating $t^q \preceq_u t^c$ again then obviously does not yield substitutions that are not already induced by $C_1 \vee \cdots \vee C_{i-1} \vee C_{i+1} \vee \cdots \vee C_m$, and thus replacing C_i by the neutral element for disjunction has no influence on $\Omega(t^q \preceq_u t^c)$. Simulation algorithm is thus sound and complete in this case.

- let t^q be an arbitrary query term of the form $id'@t$

  Decomposition with any of the rules except *desc* reduces $t^q \preceq_u t^c$ to either an atomic constraint or to a disjunction of conjunctions (in DNF), i.e.

  $$C = C_{1,1} \wedge \cdots \wedge C_{1,n_1} \vee \cdots \vee C_{i,1} \wedge \cdots \wedge C_{i,n_i} \vee \cdots \vee C_{m,1} \wedge \cdots \wedge C_{i,n_m}$$

  Assume now that any of the $C_{i,j}$ is again of the form $t^q \preceq_u t^c$ because some subterms of t^q and t^c contain cyclic references to id' and id, i.e. evaluation of $C_{i,j}$ would again yield C. As in the previous case, no new information would be added, and thus replacing $C_{i,j}$ by the neutral element for disjunction (True) has no influence on $\Omega(t^q \preceq_u t^c)$. Simulation algorithm is thus sound and complete in this case.

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

[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.

[CW96] Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[Frü95] Thom Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, Berlin, March 1995.

[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.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM Journal*, 12(1):23–41, January 1965.

[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/`.

[SBF05] Sebastian Schaffert, François Bry, and Tim Furche. Initial Draft of a Possible Declarative Semantics for the Language. Deliverable I4-D4, REWERSE, 2005.

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

[War92] David S. Warren. Memoing for Logic Programs. *Communications of the ACM*, March 1992.

[WM96] Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In *KI - Künstliche Intelligenz*, pages 377–386, 1996.