INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München

LMU

An Approach to Backward Chaining in Xcerpt

Andreas Schroeder

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	17.02.2003
Abgabe der Arbeit:	20.05.2004
Betreuer:	Prof. Dr. François Bry
	Dipl. Inf. Sebastian Schaffert

Abstract

Xcerpt is a declarative pattern based query and transformation language for semistructured data that like xml that bases on concepts and techniques of logic programming. Xcerpt uses both forward and backward chaining for program evaluation. While forward chaining is used for operations like view materialization, backward chaining is used in querying databases such as the web. This thesis presents several calculi that illustrate a possible formalization of the chaining process with respect to the the consequences of the nonstandard unification concept of Xcerpt as well as the special Xcerpt constructs *all* and *some*. These constructs are best comparable to the *bagof* predicate of Prolog. While leaning on positive analytic resolution calculi in the style of SLD, this thesis emphasizes the shortcomings of these classical calculi from logic programming for the evaluation of programs for semistructured data and hence the web, and why a constraint solving and nonlinear approach fits better with Xcerpt's way of backward and forward chaining on semistructured data.

Contents

1 Introduction

The web has grown in the last decade to a nearly infinite information pool. As the retrieval of information from the web gains more and more importance, different programming language are designed and studied in research for the purpose of xml querying and transformation like XQuery or XSLT. Xcerpt has the same purpose as those but with the aim of being more declarative. Xcerpt is a pattern based query language leaning on logic programming, but can also be seen as a transformation language, since the results of a query evaluation are rearranged into a new term of semistructured data. Xcerpt also has the advantage that query operations may be chained as in logic programming. A query may hence operate on the result of another query. This helps to modularize a query or transformation into single easily understandable steps, or to reuse queries that have been written earlier by writing a new query that uses the already existing ones. Query results can be presented in different formats of semistructured data by further transformation steps after producing the query results, while the intermediate query result format may stay the same.

All these possibilities are based on the chaining mechanism of Xcerpt. In this paper, a formal approach is made to define the operational semantics of the backward chaining mechanism by a calculus in the style of SLD resolution.

The aim of this thesis is to discuss several possibilities for the specification of operational semantics for the Xcerpt backward chaining. Each approach presented here is discussed with regards to fitness for Xcerpt, chaining behavior and clear specification.

Nevertheless, this thesis does not attempt to define a denotational semantics for Xcerpt programs. Even if some promising approaches already exist for such semantics, such a discussion is not the scope of this thesis and thus not considered here. From the point of view of mathematics, it may seem awkward to define a calculus for chaining without providing a denotational semantics at the same time. But from the point of view of computer science, it is rather usual, as it describes the implementation of a language processor. The calculus is understood as a first step towards a specification of the operational semantics of the chaining mechanism of Xcerpt.

In this thesis, the following structure is chosen. In Section 2, the syntax of Xcerpt is introduced. The syntax omits certain constructs of the full Xcerpt syntax. It is, however, sufficient to describe the backward chaining approaches discussed here and allows for a more compact presentation. This section also discusses some issues about variable-well formedness of rules. Based on the presented syntax, Section 3 gives an intuitive description of the backward chaining mechanism that is formalized in this paper. A first step for this formalization is made in Section 4, where the nonstandard unification used for Xcerpt terms is presented. The simulation unification is be introduced first for ground terms in Section 4.1, i.e. terms without variables. After this first presentation, the simulation unification is extended to terms with variables in Section 4.2. This allows to specify how variables of a query pattern must be bound or constrained within the unification process. Based on the specification of a simulation unifier, Section 4.3 presents logic equivalences for simulation unification. After the presentation of the unification method of Xcerpt, Section 5 discusses three approaches to a backward chaining algorithm. In this section, after some introductory considerations in Sections 5.1 and 5.2, two different approaches to chaining are presented in the Sections 5.3 and 5.4. The goal of these approaches is to define an operational semantics for Xcerpt

programs and hence to formalize the evaluation of Xcerpt programs. The remainder of the section is devoted to the comparison of the approaches to backward chaining (Section 5.5).

Some of the material presented here has already been published in other papers of the Xcerpt group [BS02d, BS02b, BS02c, BS02a, BS02e, BBSW03, BS03]. Most of the repetition here is done as prerequisite to the formulation of the chaining calculus. Nevertheless, some material is presented here in a different form than in past publications. This reflects the working progress of the Xcerpt research group.

2 Xcerpt Terms

In Xcerpt, there are three types of terms: data terms, query terms and construct terms. The set of all data terms is contained in both the set of all query terms and in the set of all construct terms. The Xcerpt rules and goals themselves, the most frequent elements within an Xcerpt program, consist of a construct term and a query formula.

2.1 Data, Query and Construct Terms

In this section, the sets of data terms, query terms and construct terms, denoted by $\mathcal{T}^d$, $\mathcal{T}^q$ and $\mathcal{T}^c$ respectively are defined. For these definitions, the following (disjunct) sets are needed:

1. A set of variables. Variables are denoted by the uppercase letters X, Y and Z.
2. A set of identifiers. Identifiers are denoted by *id*.
3. A set of labels. Labels are denoted by l.

Note that all variables are written uppercase, while terms, labels and identifiers are all written lowercase. Terms themselves are denoted by t and s. All these notations may also be extended by decoration and indices.

Definition 1 (Data Terms) *The set of data terms, $\mathcal{T}^d$ is inductively defined as follows. Let id an identifier, l a label,* `text` *a string containing no quotation marks (“”), $t_i \in \mathcal{T}^d$ be $1 \leq i \leq n \in \mathbb{N}$ data terms, let $t \in \mathcal{T}^d$ be a data term.*

$$\begin{aligned} \text{“}text\text{”}, l\{\} &\in \mathcal{T}^d \\ l\{t_1, \ldots, t_n\} &\in \mathcal{T}^d \\ l[t_1, \ldots, t_n] &\in \mathcal{T}^d \\ id@t, \uparrow id &\in \mathcal{T}^d \end{aligned}$$

In the following, a standalone label l is used as a convenient abbreviation of the (semantically equivalent) data terms $l\{\}$ and $l[]$ with no children.

Curly braces indicate that the ordering of the subterms $t_1, \ldots, t_n$ (also called *children*) does not matter, while square brackets indicate that the ordering of the subterms does matter.

$$f\{a, b, c\} = f\{a, c, b\} = f\{b, c, a\} = \ldots$$

$$f[a, b, c] \neq f[a, c, b] \neq f[b, c, a] \neq \ldots$$

Example 1 (Data Terms) *The following example models in semistructured data parts of the food web of a river. A single entry of this database is e.g. the data term*

$$\begin{aligned} trout := \; & species\{ \; name\{\text{“trout”}\}, \\ & \quad eats\{\text{“stonefly/mayfly nymph”}\}, \\ & \quad eats\{\text{“bacteria, protozoa, fungus”}\}, \\ & \quad eats\{\text{“fish eggs”}\}\} \end{aligned}$$

The base of the biological system is the sunlight and the $fishEgg$:

$$\begin{aligned} sunlight := \; & species\{ \; name\{\text{“sunlight”}\}\} \\ fishEgg := \; & species\{ \; name\{\text{“fish egg”}\}\} \end{aligned}$$

In the food chain, the species treeAndShrub is just above:

$$\begin{aligned} treeAndShrub := \; & species\{ \; name\{\text{“deciduous tree and shrub”}\}, \\ & \quad eats\{\text{“sunlight”}\}\} \end{aligned}$$

. . . and so on:

$$\begin{aligned} leafMatter := \; & species\{ \; name\{\text{“leaf matter falling into stream”}\}, \\ & \quad eats\{\text{“deciduous tree and shrub”}\}\} \\ bacteriaAndFungus := \; & species\{ \; name\{\text{“bacteria, protozoa, fungus”}\}, \\ & \quad eats\{\text{“leaf matter falling into stream”}\}\} \\ stoneMayFly := \; & species\{ \; name\{\text{“stonefly/mayfly nymph”}\}, \\ & \quad eats\{\text{“bacteria, protozoa, fungus”}\}\} \end{aligned}$$

The seven data terms must be held in a single root element, so the following data term is build:

$$\begin{aligned} foodweb\{ \; & species\{ \; name\{\text{“sunlight”}\} \}, \\ & species\{ \; name\{\text{“fish egg”}\} \}, \\ & species\{ \; name\{\text{“deciduous tree and shrub”}\}, \\ & \quad eats\{\text{“sunlight”}\} \\ & \}, \\ & species\{ \; name\{\text{“leaf matter falling into stream”}\}, \\ & \quad eats\{\text{“deciduous tree and shrub”}\} \\ & \}, \\ & species\{ \; name\{\text{“bacteria, protozoa, fungus”}\}, \\ & \quad eats\{\text{“leaf matter falling into stream”}\} \\ & \}, \\ & species\{ \; name\{\text{“stonefly/mayfly nymph”}\}, \\ & \quad eats\{\text{“bacteria, protozoa, fungus”}\} \\ & \}, \\ & species\{ \; name\{\text{“trout”}\}, \\ & \quad eats\{\text{“stonefly/mayfly nymph”}\}, \\ & \quad eats\{\text{“bacteria, protozoa, fungus”}\}, \\ & \quad eats\{\text{“fish eggs”}\} \\ & \} \\ \} & \end{aligned}$$

Identifiers are used to reference shared subterms or to create cycles in the intended structures of the terms. The smallest possible cycle is described by the term

$$1@a\{\uparrow 1\}$$

in which a has itself as child.

This referencing mechanism makes a dereferencing necessary for the specification of properties of a term containing references. Furthermore, the existence of cycles in a term makes some of the following definitions of sets, if they are understood as build by functions, non-terminating. There are simple methods for recovering from this issue, e.g. memoization. This matter is left apart in the following definitions, because it would take too much attention away from the real intentions of the material presented, and technically complicate the definitions. Nevertheless, a dereferencing operation is needed:

Definition 2 (Dereferencing) *Let id be an identifier. The* dereferencing *of the reference* $\uparrow id$*, is the term tagged with the same, unique identifier of the form* $id@t$*, where* t *must match the term type of the reference* $\uparrow id$*. The procedure of dereferencing is called* $deref$*:*

$$deref(\uparrow id) := t$$

The scope of an id is a program. Hence, an identifier must be unique in a program and can be referenced by several distinct terms in a same program. At the same time it must be guaranteed that the referenced term and referencing terms are of the same type. It is illegal to reference a construct term within a query term and vice versa.

Example 2 (Dereferencing) *Assume that the following two data terms* t_1 *and* t_2 *are given in the same program* P*.*

$$t_1 := f[1@g[a,b], 2@h[a,b]] \qquad t_2 := g[\uparrow 1, 3@f[\uparrow 2, \uparrow 3]]$$

The term t_2 *references subterms of* t_1 *as own subterms and specifies a cycle with id* 3*. In the context of both* t_1 *and* t_2*, the dereferencing of the ids* 1 *and* 2 *yield the same result:*

$$deref(1) = g[a,b] \quad deref(2) = h[a,b]$$

The term t_2 *could be assumed to be a shorthand for*

$$t_2' := g[g[a,b], 3@f[h[a,b], \uparrow 3]]$$

Nevertheless, the dereferencing of 3 *differs between the term versions* t_2 *and* t_2'*:*

$$deref(3) = f[\uparrow 2, \uparrow 3] \qquad deref(3) = f[h[a,b], \uparrow 3]]$$

To perform a query against a data term, a query term is needed. A query term specifies a pattern to match a data term or a construct term with.

Definition 3 (Query Terms) *$\mathcal{T}^q$ denotes the set of query terms, and is inductively defined as follows.*
Let id be an identifier, l a label, X a variable, text a string containing no quotation marks (“”), let $t \in \mathcal{T}^q$ be a data term, let $t_i \in \mathcal{T}^q$ be $1 \leq i \leq n \in \mathbb{N}$ query terms.

$$\begin{aligned} \text{“}text\text{”}, l\{\}, l\{\{\}\} &\in \mathcal{T}^q \\ \uparrow id,\ id@t &\in \mathcal{T}^q \\ X, X \rightsquigarrow t, desc\ t &\in \mathcal{T}^q \\ l\{t_1, \ldots, t_n\}, l[t_1, \ldots, t_n] &\in \mathcal{T}^q \\ l\{\{t_1, \ldots, t_n\}\}, l[[t_1, \ldots, t_n]] &\in \mathcal{T}^q \end{aligned}$$

A query term can yield a full or partial pattern besides specifying the ordering of the searched terms by curly braces or square brackets. A total pattern is given by single braces and a partial pattern by double braces.

In the following section, the simulation unification is introduced on a restricted form of query terms, the ground query terms. Because the whole Xcerpt chaining and matching mechanism base upon this restricted version of Xcerpt, the following definition of ground query terms is of great importance.

Definition 4 (Ground Query Terms) *A query term $t \in \mathcal{T}^q$ is* ground*, if it contains no variable, no desc and no $\rightsquigarrow$. The set of ground query terms is denoted by $\mathcal{T}^q_{ground}$.*

For this paper, the query terms need to be further restricted. For the introduction of substitutions for query terms it is most convenient only to allow a *desc* construct at the right hand side of an $\rightsquigarrow$ construct, in the form $X \rightsquigarrow desc\ t$. Fulfilling this requirement could be achieved by a simple transformation of a general query term by introducing “fresh” variables and an $X \rightsquigarrow$ on every *desc t* pattern. Obviously, this requirement does not restrict the expressive power of Xcerpt in any way.

In the following, especially in the sections about introducing variables, it is assumed that *desc* only occurs as direct subterm on the right hand of a $\rightsquigarrow$-shaped term.

Example 3 (Query Terms) *The following queries match the data terms:*

- *$a\{b, c\}$ matches $a\{c, b\}$ because the labels are the same and all children of the query term left can be found in arbitrary order on the left data term and only those specified children can be found.*
- *$a\{\{b\}\}$ matches $a\{b, c\}$ because the double curly braces specifies a partial query and all children within the partial query can be found on the right hand side.*
- *$a[b, c]$ matches $a[b, c]$ because the children can be found in the same order on the right hand side than specified on the left hand side, and only the specified children are found on the right hand side.*
- *$a[[b, c]]$ matches $a[d, b, c, d]$ because the the children b and c is found in the proper order on the right hand side.*

But the following queries do not match:

- *$a[b, c]$ does neither match $a\{b, c\}$ nor $a[c, b]$ because the first term on the right hand side does not have any child ordering, and the second term on the right hand side has the wrong child ordering.*
- *$a\{b\}$ does not match $a\{b, c\}$ because more children are found on the right hand side than b and the query term on the left hand side specifies a complete query pattern for the children of a.*

It is also possible to query cycles in Xcerpt:

$$X \rightsquigarrow 1@a\{\uparrow 1\} \text{ matches } 2@a\{\uparrow 2\}$$

The construct $X \rightsquigarrow t^q$ binds a variable X to the data term that matches t^q. This allows to bind a variable to a result while restricting the set of possibly bound terms:

$$X \rightsquigarrow a\{\{b\}\} \text{ matches } a\{b, c\} \text{ and yields } X = a\{b, c\} \text{ as binding.}$$

Here are two sample query terms from the food web examples:

$$foodweb\{\{\ species\{\{\ name\{Y\},\ eats\{X\}\ \}\}\ \}\}$$

$$foodweb\{\{\ X \rightsquigarrow species\{\{\ eats\{Y\},\ eats\{Z\}\ \}\}\ \}\}$$

Of course, this query terms matches the food web database presented above. However, there are several possibilities to find a matching of these query against the food web database. Each of these possible matches generates a substitution for the variables X and Y (the formal definition for this process is discussed in Section 4.2). Some substitutions generated from the matching of the second query term would be the following.

```
{
    { "leaf matter falling into stream"/X, "bacteria, protozoa, fungus"/Y},
    { "bacteria, protozoa, fungus"/X, "stonefly/mayfly nymph"/Y},
    { "bacteria, protozoa, fungus"/X, "trout"/Y},
}.
```

But what does "match" really mean in Xcerpt? The examples just provided an intuition on this. Since a calculus for Xcerpt is given here, the definition of this mechanism for matching a query term against a data term must be specified. How the matching is done and how the unification of query terms with data or construct terms is performed is described in the sections to come.

The result of the unification of a query with a data term or rule head, the variable bindings, provided from the evaluation against a data term must be rearranged in a new data term. How the variables have to be reassembled is described by the programmer with a construct term, which gives a pattern for the data term to be built.

Definition 5 (Construct Terms) *$\mathcal{T}^c$ denotes the set of construct terms and is inductively defined as follows.*

Let id be an identifier, l be a label, X a variable, text a string containing no quotation marks (“”), $m \in \mathbb{N}$, let $t_i \in \mathcal{T}^c$ be $1 \leq i \leq n \in \mathbb{N}$ construct terms.

$$\begin{aligned} \text{“}text\text{”}, l\{\}, X &\in \mathcal{T}^c \\ l\{t_1, \ldots, t_n\} &\in \mathcal{T}^c \\ l[t_1, \ldots, t_n] &\in \mathcal{T}^c \\ \uparrow id, id@t &\in \mathcal{T}^c \\ all\ t_1, some\ m\ t_1 &\in \mathcal{T}^c \end{aligned}$$

The root of a construct term, that is, the outermost structure of a data term, may neither be of the form *all t*, *some m t*, *id@all t* or *id@some m t*, nor $\uparrow id$ with *id* referencing a term in one of the four precluded forms above.

In addition to the standard definition of construct terms, a subset of them is defined in the set of grouping construct terms. This definition is useful in the specification of the simulation unification in Section 4, that cannot decompose a term of the form *all* t^c and *some n* t^c.

Definition 6 (Grouping Construct Terms) *The set $\mathcal{T}_g^c \subset \mathcal{T}^c$ is defined as follows. A construct term $t^c \in \mathcal{T}^c$ is* grouping, *$t^c \in \mathcal{T}_g^c$ iff it contains subterms of the form some n t' or all t'.*

A term t^c is non-grouping, *$t^c \in \mathcal{T}_{ng}^c$, iff it contains no subterms of the form some n t' or all t'.*

The non-grouping construct terms can also be seen as an extension of the data terms by allowing variable occurrences.

Example 4 (Construct Terms) *Here is a first example of a non-grouping construct term.*

$$a\{b[X, Y], X, c\{Y\}\ \}$$

Assuming that the bindings for X and Y were given as $\{f/X, g/Y\}$, the construct term would then rearrange the bindings and yield

$$a\{b[f, g], f, c\{g\}\ \}$$

Considering at a second, grouping example:

$$reverse foodweb\{\ all\ species\{\ name\{X\}, all\ eatenby\{Y\}\ \}\ \}$$

This construct term rearranges all possible bindings for variables X and Y so that for a single binding of X all possible bindings of Y are listed within an individual “eatenby” term for every Y. Assume that the possible bindings for X and Y were given as

{
  { *“leaf matter falling into stream”*/*X*, *“bacteria, protozoa, fungus”*/*Y*},
  { *“bacteria, protozoa, fungus”*/*X*, *“stonefly/mayfly nymph”*/*Y*},
  { *“bacteria, protozoa, fungus”*/*X*, *“trout”*/*Y*},
}.

Then the construct term would rearrange the bound terms as

$$
\begin{array}{lll}
reversefoodweb\{ & species\{ & name\{\text{“leaf matter falling into stream”}\}, \\
 & & eatenby\{\text{“bacteria, protozoa, fungus”}\}, \\
 & \}, & \\
 & species\{ & name\{\text{“bacteria, protozoa, fungus”}\}, \\
 & & eatenby\{\text{“stonefly/mayfly nymph”}\}, \\
 & & eatenby\{\text{“trout”}\} \\
 & \} & \\
\} & &
\end{array}
$$

How the bindings for variables are created - or even how the creation of bindings is avoided - is discussed in the sections to come.

2.2 Rules, Goals and Programs

A program is mainly a set of rules, goals and data terms, where the goals are special rules to start the evaluation of an Xcerpt program. The program rules and goals themselves consists of the above presented query and construct terms. A rule consists of one construct term and one or more query terms grouped in a query formula. This query formula may in Xcerpt, in contrast to Prolog, consist of a disjunction or conjuction of queries.

Definition 7 (Query Formulas) *Let $t^q \in \mathcal{T}^q$ be a query term, $Q_i \in QFormula$ be $1 \leq i \leq n \in \mathbb{N}, n \geq 0$ Query Formulas. $QFormula$ denotes the set of Query Formulas and is inductively defined as follows.*

$$
\begin{aligned}
t^q &\in QFormula \\
Q_1 \wedge \ldots \wedge Q_n &\in QFormula \\
Q_1 \vee \ldots \vee Q_n &\in QFormula
\end{aligned}
$$

In the following, the infix notations $Q_1 \wedge Q_2$ and $Q_1 \vee Q_2$ are used when convenient. With these formulas, rules are built as follows.

Definition 8 (Rules and Goals) *Let $t^c \in \mathcal{T}^c$ be a construct term, $Q \in QFormula$ be a query formula.*

$$r := t^c \leftarrow Q$$

is a Rule. *Q is called* body *of r and t^c is called the* head *of r.*

$$g := t^c \leftarrow Q$$

is a Goal. *Q is called* body *of g and t^c is called the* head *of g.*

Goals are “entry points” for the evaluation of an Xcerpt program. The resulting terms of a goal evaluation are the only terms visible from the outside of an Xcerpt program. The semantics of an Xcerpt program could hence be understood as the set of goal results. A language processor must hence evaluate these goals. At the same time, goals may not be

queried by rules of the program itself. This restriction has two reasons. First, it may be unfavorable for an outside application to have a result be created bit by bit. Second, this separation of goals and rules allows the result to be rearranged in a result type other than data terms (i.e. xml data). Hence, any textual type (such as programs in other programming languages) could be created by means of Xcerpt reasoning.

Example 5 (Rules) *consider the following rules that can be executed against the biological database presented above.*

1. $$\begin{array}{l} inversefoodweb\{\ all\ species\{\ name\{X\},\ all\ eatenby\{Y\}\ \}\ \} \\ \leftarrow \\ \quad foodweb\{\{\ species\{\{\ name\{Y\},\ eats\{X\}\ \}\}\ \}\} \end{array}$$

   This rule queries the food web presented in Example 1 and transforms the given data in an inverse presentation. Instead of having the information arranged as

   $$species\{\ name\{\ldots\},\ eats\{\ldots\},\ldots\}$$

   the data is presented as

   $$species\{\ name\{\ldots\},\ eatenby\{\ldots\},\ldots\}$$

   This can be useful for certain operations or knowledge extraction from the database.

2. *Now a second example.*

   $$\begin{array}{l} couldbetter\{\ all\ X\ \} \\ \quad \leftarrow \\ \quad\quad foodweb\{\{\ X \rightsquigarrow species\{\{\ eats\{Y\},\ eats\{Z\}\ \}\}\ \}\} \\ \quad\quad \wedge \\ \quad\quad foodweb\{\{\ species\{\{\ name\{Y\},\ eats\{Z\}\ \}\}\ \}\} \end{array}$$

   This rule lists all species who eats another species that it eats also "transitively" and thus could do better in sense of "eating efficency" by eating only the lower species and leave apart a energy consumer between both species.

However, not every rule or goal is allowed in an Xcerpt program. The rules and goals must be range restricted, that is, every variable of the head must occur in the body. The definition of range restriction for Xcerpt rules has to consider that the bodies of rules or goals does not only consist of conjunctions but may also contain disjunctions. A further restriction on variables occurrences in the rule or goal head arise from the fact that variables are not shadowed in grouping constructs. Variables which occur within the range of a grouping construct may not occur outside of its range - neither outside the grouping term nor nested within in a subterm with another grouping construct. Rules and Goals respecting these restriction are called *variable well formed* rules and goals.

Definition 9 (Variables) *Let $t^q \in \mathcal{T}^q$ be a query term. The* Variables *of t^q, $vars_q(t^q)$ are recursively defined as follows.*

- $vars_q(\text{“}text\text{”}) = vars_q(l\{\}) = vars_q(l\{\{\}\}) = \emptyset$
- $vars_q(l\{t_1^q, \ldots, t_n^q\}) = vars_q(l\{\{t_1^q, \ldots, t_n^q\}\}) = \bigcup_{i=1}^{n} vars_q(t_i^q)$
- $vars_q(l[t_1^q, \ldots, t_n^q]) = vars_q(l[[t_1^q, \ldots, t_n^q]]) = \bigcup_{i=1}^{n} vars_q(t_i^q)$
- $vars_q(id@t'^q) = vars_q(t'^q)$
- $vars_q(\uparrow id) = vars_q(deref(\uparrow id))$
- $vars_q(X) = \{X\}$
- $vars_q(desc\ t'^q) = vars_q(t'^q)$
- $vars_q(X \rightsquigarrow t'^q) = \{X\} \cup vars_q(t'^q)$

Let $t^c \in \mathcal{T}^c$ be a construct term. The Variables *of t^c, $vars_c(t^c)$ are recursively defined as follows.*

- $vars_c(\text{“}text\text{”}) = vars_c(l\{\}) = vars_c(l\{\{\}\}) = \emptyset$
- $vars_c(l\{t_1^c, \ldots, t_n^c\}) = vars_c(l[t_1^c, \ldots, t_n^c]) = \bigcup_{i=1}^{n} vars_c(t_i^c)$
- $vars_c(X) = \{X\}$
- $vars_c(\uparrow id) = vars_c(deref(\uparrow id))$
- $vars_c(id@t'^c) = vars_c(t'^c)$
- $vars_c(some\ n\ t'^c) = vars_c(all\ t'^c) = vars_c(t'^c)$

Let $t^c \in \mathcal{T}^c$ be a construct term. The free Variables *of t^c, $freevars_c(t^c)$ are recursively defined as follows.*

- $freevars_c(\text{“}text\text{”}) = freevars_c(l\{\}) = freevars_c(l\{\{\}\}) = \emptyset$
- $freevars_c(l\{t_1^c, \ldots, t_n^c\}) = freevars_c(l[t_1^c, \ldots, t_n^c]) = \bigcup_{i=1}^{n} freevars_c(t_i^c)$
- $freevars_c(X) = \{X\}$
- $freevars_c(\uparrow id) = freevars_c(deref(\uparrow id))$
- $freevars_c(id@t'^c) = freevars_c(t'^c)$
- $freevars_c(some\ n\ t'^c) = freevars_c(all\ t'^c) = \emptyset$

Let $t^c \in \mathcal{T}^c$ be a construct term. The bound Variables *of t^c, $boundvars_c(t^c)$ are recursively defined as follows.*

- $boudvars_c(\text{“}text\text{”}) = boundvars_c(l\{\}) = boundvars_c(l\{\{\}\}) = \emptyset$
- $boundvars_c(l\{t_1^c, \ldots, t_n^c\}) = boundvars_c(l[t_1^c, \ldots, t_n^c]) = \bigcup_{i=1}^{n} boundvars_c(t_i^c)$

- $boundvars_c(X) = \emptyset$
- $boundvars_c(\uparrow id) = boundvars_c(deref(\uparrow id))$
- $boundvars_c(id@t'^c) = boundvars_c(t'^c)$
- $boundvars_c(some\ n\ t'^c) = boundvars_c(all\ t'^c) = freevars_c(t'^c)$

Let $Q \in QFormula$ be a query formula. The variables *of Q, $vars_Q(Q)$ are recursively defined as follows.*

- $vars_Q(t^q) = vars_q(t^q)$ *as defined above for query terms.*
- $vars_Q(Q_1 \vee \ldots \vee Q_n) = vars_Q(Q_1) \cap \ldots \cap vars_Q(Q_n)$
- $vars_Q(Q_1 \wedge \ldots \wedge Q_n) = vars_Q(Q_1) \cup \ldots \cup vars_Q(Q_n)$

Informally, variable well-formedness indicates that every variable which occurs in the head of a rule must occur in each disjunction of the disjunctive normal form of its body. This must hold if the body is both in disjunctive normal form and arbitrary forms. Furthermore, every variable that is bound by an *all* or *some n* is not allowed to be free nor bound by another *all* or *some n*.

To formalize this requirement, it is necessary to introduce the definition of *variable well-formedness*. In the following all definitions are restricted to variable well formed rules and goals and only the cases in which rules and goals are variable well formed are studied.

Definition 10 (Variable Well-formedness) *Let t^c be a construct term. t^c is* variable well formed *iff $freevars_c(t^c) \cap boundvars_c(t^c) = \emptyset$ and all of the following holds.*

- *if $t^c = \uparrow id$, then for $t'^c := deref(\uparrow id)$ it must hold that $freevars_c(t'^c) \cap boundvars_c(t'^c) = \emptyset$ and t'^c must be variable well formed.*
- *if $t^c = l\{t_1^c, \ldots, t_n^c\}$ or $t^c = l[t_1^c, \ldots, t_n^c]$ then for all t_i^c it must hold that $freevars_c(t_i^c) \cap boundvars_c(t_i^c) = \emptyset$ and all t_i^c must be variable well formed.*
- *if $t^c = id@t'^c$, $t^c = some\ n\ t'^c$ or $t^c = all\ t'^c$, then t'^c must be variable well formed.*

Let $r = t^c \leftarrow Q$ be a rule or a goal. r is variable well formed *iff*

- *t^c is variable well formed and*
- *$vars_c(t^c) \subseteq vars_Q(Q)$ (r is* range restricted*).*

Example 6 (Variable Well-formedness)

- *$f[all\ X] \leftarrow g[[Y]] \wedge h[[X]]$ is variable well-formed. The occurrence of an additional variable Y in the body that never occur in the head is allowed.*

- $f[all\ X] \leftarrow g[[Y]] \vee (h[[X]] \wedge k[[Y]])$ *is not variable well-formed. The variable* X *does not occur in every conjunction of the DNF of the body.*
- $f[all\ X, X] \leftarrow h[[X]]$ *is not variable well-formed. The variable* X *is both bound and free in the head of the rule.*
- $f[all\ X, Y] \leftarrow g[[Y]] \wedge h[[X]]$ *is variable well-formed.*
- $f[all\ a[X, all\ b[Y]\ ]\ ] \leftarrow g[[Y]] \wedge h[[X]]$ *is variable well-formed.*
- $f[all\ a[X, all\ b[X]\ ]\ ] \leftarrow h[[X]]$ *is not variable well formed. The variable* X *is both bound and free in the subterm* $a[X, all\ b[X]\ ]$ *of the head.*

In the following, *Rules* denotes the set of all variable well formed rules and *Goals* denotes the set of all variable well formed goals. Rules and goals with grouping heads or with non-grouping construct terms are denoted by $Rules_g, Rules_{ng}, Goals_g$ and $Goals_{ng}$ respectively. A formal definition of an Xcerpt program is given as follows.

Definition 11 (Program) *A* Program *is a finite set of (variable well formed) rules, goals and data terms that contains at least one goal. Furthermore, for a program* P

- $goals(P) := P \cap Goals$ *denotes the set of all goals of* P*,*
- $rules(P) := P \cap Rules$ *denotes the set of all rules of* P*,*
- $rules_g(P) := P \cap Rules_g$ *denotes the set of all grouping rules of* P*,*
- $rules_{ng}(P) := P \cap Rules_{ng}$ *denotes the set of all non-grouping rules of* P*,*
- $goals_g(P) := P \cap Goals_g$ *denotes the set of all grouping goals of* P*,*
- $goals_{ng}(P) := P \cap Goals_{ng}$ *denotes the set of all non-grouping goals of* P *and*
- $terms(P) := P \cap \mathcal{T}^d$ *denotes the set of all data terms of* P*.*

A program is hence a finite subset of $Rules \cup Goals \cup \mathcal{T}^d$.

The definition of query formulas and goals presented in this thesis omits the resource concept present in the real Xcerpt language as it would complicate the definition of the calculus. It is thus assumed that every data term queried by a program is present in the program itself. An analysis that determines this set of resources is simple to perform, and a transformation of this set and the program rules that preserve the "segmentation" of different resources and the query terms against specific resources is also simple: a new data term with an outermost label naming a unique resource identifier and containing the data term accessible by the specified resource is added to the program and each query term querying the resource is modified in the same way.

What is missing in such an approach is the possibility to provide a resource by a variable that is bound by a previous query. Such a flexibility can easily be introduced once the rule chaining is defined, and is therefore left aside in this paper.

In this section, the term sets have been introduced: data, query and construct terms. Furthermore, the rules and goals of which an Xcerpt program can contain besides data terms were defined and restrictions for well formedness were introduced. Since the upcoming sections are somewhat theoretical, a short description on the process of querying is given in the following that should be kept in mind while reading further on.

3 An Intuitive Description of Program Evaluation

The last section already gave an intuition about how chaining and querying in Xcerpt works. However, there are still some preliminaries missing in order to present chaining calculi. Basically, the unification mechanism must be defined before any presentation of chaining. Since the unification used in Xcerpt to unify query terms (of a rule or goal body) and construct terms (of a rule head) or data terms (of a database or simply a web page) is rather new, the room this definition takes is quite large. Some aspects of the simulation unification and the search for a simulation unifier affects the chaining calculi. Especially the incompleteness of the simulation unification algorithm regarding grouping construct terms has interesting consequences for a chaining calculus.

The aim of a chaining calculus is the formalization of the evaluation of an Xcerpt program. Basically, the evaluation of a program works as follows. First, the goals of the program are triggered, and their bodies are evaluated. The query terms of which these bodies consists are matched against a data term or rule head to find a data term or construct term which unifies with the query term. If the query term matches with (i.e. unifies with) a data term, then this match yields constraints (lower and upper bounds) for the variables in the form of a *simulation unifier*. If the query term matches a rule head, the simulation unifier from this unification is collected in a constraint store and the body of the rule is evaluated. These new constraints affect the possible matches of the rule body. These chain of repeated query term evaluation end with the matching of a unifiable data term. An infinite loop is hence programmable in Xcerpt by specifying a rule with a query term that matches its body and produces no further constraints on the variable bindings to store which could ensure termination. The following rule for example would produce an infinite loop, if it is evaluated:

$$loop\{\} \leftarrow loop\{\}$$

Or with simplified syntax, omitting the curly brackets:

$$loop \leftarrow loop$$

The intuition behind a query term is a pattern that is matched against a data term or construct term as a "substructure" of a larger term. A query term offers a pattern that is matched with data terms or rule heads. A construct term on the other hand, describes how the bindings emerging from the body evaluation should be rearranged to produce new data terms.

These simple concepts build a powerful, yet simple to understand language for querying and transforming semistructured data. The aim of the following sections is to define the simulation unification, that is, the "matching" of query terms with construct terms and data terms.

4 Simulation Unification

As semistructured data is very heterogeneous, Xcerpt uses a non-standard unification algorithm for unifying Xcerpt terms. This section introduces this method of unification. The start of this introduction is to define unification of ground terms and then extend the simulation unification to terms with variables based upon the first approach. Together with simulation unification for non-ground terms, the notion of "simulation unifier"is defined.

However, the simulation unification is not the main emphasis of this paper. The presentation of the simulation unification is a preliminary for the discussion of chaining calculi. Most of the material presented in this section is based on the work done in [BS02e] and on internal Notes of the Xcerpt team.

4.1 Simulation Unification for Ground Query Terms

This section discusses the core concepts of simulation unification. The ground query terms fit well for the purpose of a first approach to simulation unification, because aspects involving variables, constraints and substitutions are omitted, but compared to data terms, they offer the partial query specifications $\{\{\}\}, [[]]$. The resulting changes from the introduction of variables are discussed in Section 4.2.

4.1.1 Graphs for Ground Query Terms

Term simulation is a notion based on graphs. Intuitively, a graph G_1 simulates in another graph G_2 if there exist a mapping of the nodes of G_1 into the nodes of G_2 that respect the edges. Therefore, G_1 simulates in G_2 if its structure can be found as a subgraph of G_2. By introducing a relation on nodes, the possible simulations can be restricted to those that also respect this node relation.

For the purpose of this thesis, graphs, edges and nodes are defined as follows.

Definition 12 (Rooted Graph) *A Rooted Graph is a triple $G = (V, E, r)$ of nodes V, edges $E \subseteq V \times \mathbb{N} \times V$ and root $r \in V$.*

Let $v \in V$ be a node. The set of outgoing edges *of v is the set $\{e_i \mid e_i \in E, e_i = (v, i, w)\}$ with v on the left side of the relation. $e_i = (v, i, w)$ is called* i-th outgoing edge *of v.*

The natural number of the edges of a graph $e_i = (v, i, w)$ are understood as numbering the outgoing edges of a node. A total order over the outgoing edges is induced by this notation.

The simulation of ground query terms presented here defines the matching of a query term into another. Since every data term is a ground query term, this definition is a core concept of Xcerpt. In order to apply the notion of simulation to ground query terms, it is necessary to define a graph representation for ground query terms. The graph $G = (V, E, r)$ representing a ground query term $t \in \mathcal{T}^q_{\text{ground}}$ is the *graph induced by* t, $G = G(t)$.

The graph induced by a ground query term is built as follows. The nodes of the induced graph are adorned with the labels of the query term together with their type of subterm grouping (i.e. {}, {{}}, [] or [[]]). The defining occurrences of identifiers are omitted on the induced

graph, and references to identifiers are replaced by edges to the respective nodes. The edges of a query term are directed from the node with the parent label to the node with the labels of the direct subterms, and the edge order is the same as the order of these subterms. The root of a query term is the node with the left outermost label. The labels of a graph node are noted here after the node name within angled brackets: $node\langle label\rangle$. For the retrieval of a label from a node $n = n_{name}\langle n_{label}\rangle$, a function $label(n) = n_{label}$ is used. Note that within a labeled graph, there might be distinct nodes with the same label.

Example 7 (Graph induced by Ground Query Term) *In this example, the graph $G(t) = (V, E, r)$ induced by the following ground query term*

$$t := a\{\{1@b[\uparrow 1, c\{\}], c[[\text{“}text\text{”}]]\}$$

is studied. Each of the three components of $G(t)$ is presented separately.

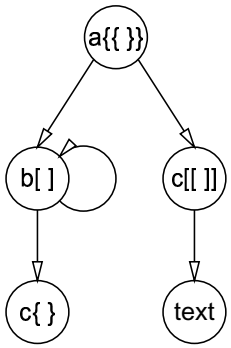
The set of all nodes, V, consists of the following elements:

$$V = \{v_1\langle a\{\{\}\}\rangle, v_2\langle b[]\rangle, v_3\langle c\{\}\rangle, v_4\langle c[[]]\rangle, v_5\langle\text{“}text\text{”}\rangle\}$$

The set of all edges $E \subset V \times \mathbb{N} \times V$ of the induced graph from t is then as follows:

$$\begin{aligned} E = \quad & \{(v_1\langle a\{\{\}\}\rangle, 1, v_2\langle b[]\rangle), (v_1\langle a\{\{\}\}\rangle, 2, v_4\langle c[[]]\rangle), (v_2\langle b[]\rangle, 1, v_2\langle b[]\rangle), \\ & (v_2\langle b[]\rangle, 2, v_3\langle c\{\}\rangle), (v_4\langle c[[]]\rangle, 1, v_5\langle\text{“}text\text{”}\rangle)\} \end{aligned}$$

Finally, the root of $G(t)$ is the node $r = v_1\langle a\{\{\}\}\rangle$. A graphical representation of this graph can be found below. In this representation, the labels are drawn within the nodes, while the node names are left aside. These node names are of no importance for simulation unification of terms.



4.1.2 Term Simulation

Based on the graphs of ground query terms, the simulation unification of terms, written $t_1 \preceq t_2$, is defined as a mapping of the nodes of t_1 in the nodes of t_2 that respects the edges and the labels of the nodes in a certain way. Intuitively, a term t_1 simulates into t_2 if it can be found as a substructure of t_2.

For the restriction of possible node matches, a relation called *Node Label Match ($\sim_{NLM}$)* is defined. A mapping of the nodes must respect the node label match relation to be a valid simulation unification.

Definition 13 (Node Label Match) *The relation $\sim_L$, called* Node Label Match, *is a relation that fulfills the following conditions.*

1. *for all strings s containing no quotation marks it holds that*

$$\text{“}s\text{”} \quad \sim_L \quad \text{“}s\text{”}.$$

2. *for all term labels l it holds that*

$$\begin{array}{lrcl}
2.1 & l\{\} & \sim_L & l\{\} \\
2.2 & l\{\} & \sim_L & l[] \\
2.3 & l[] & \sim_L & l[] \\
2.4 & l\{\{\}\} & \sim_L & l\{\{\}\} \\
2.5 & l\{\{\}\} & \sim_L & l[[]] \\
2.6 & l\{\{\}\} & \sim_L & l\{\} \\
2.7 & l\{\{\}\} & \sim_L & l[] \\
2.8 & l[[]] & \sim_L & l[[]] \\
2.9 & l[[]] & \sim_L & l[].
\end{array}$$

A node v_1 is said to match a node v_2 by node label match iff

$$label(v_1) \sim_L label(v_2).$$

Example 8 (Node Label Match) *Considering the matching of the following pairs of nodes:*

$$\begin{array}{lllll}
pair\ 1: & v_1\langle a\{\}\rangle & and & v_2\langle a[[]]\rangle \\
pair\ 2: & v_1\langle a\{\{\}\}\rangle & and & v_2\langle a[[]]\rangle \\
pair\ 3: & v_1\langle a\{\}\rangle & and & v_2\langle a[]\rangle \\
pair\ 4: & v_1\langle a[]\rangle & and & v_2\langle a\{\}\rangle
\end{array}$$

The matching of the presented pairs is as follows.

- *pair* 1 *does not match. $a\{\}$ occurs on the left of $\sim_L$ in case 2.1 and 2.2. In those cases, $a[[]]$ does not occur on the right hand side: a total subterm term specification does never match a partial term specification.*
- *pair* 2 *matches. This match is specified by case* 2.4. *An unordered subterm specification matches always its ordered counterpart.*
- *pair* 3 *matches. The match is specified in case* 2.2 *with the same reason as for pair* 2
- *pair* 4 *does not match. $a[]$ occurs on the left hand side only in case 2.3. An ordered subterm specification matches never an unordered one. The subterm order specified by $[]$ is not preserved on the right hand side.*

After the restriction of the nodes was defined, the simulation of ground query terms can be defined. Note that this definition includes restriction on the edges as well. These restrictions are explained in details afterwards.

Definition 14 (Ground Query Term Simulation) *Let $t_1, t_2 \in \mathcal{T}^q_{ground}$, and let $G_1 = (V_1, E_1, r_1) = G(t_1), G_2 = (V_2, E_2, r_2) = G(t_2)$. A relation $\preceq \subseteq V_1 \times V_2$ is a* ground query term simulation *iff all of the following hold.*

1. $r_1 \preceq r_2$
2. *If $v_1 \preceq v_2$, then $label(v_1) \sim_L label(v_2)$ (node label match).*
3. *Let $\{e_1, \ldots, e_m\} \subseteq E_1, \{f_1, \ldots, f_n\} \subseteq E_2$ be the outgoing edges of the nodes v_1 and v_2. If $v_1 \preceq v_2$ then for each edge $e_i = (v_1, i, v_1^i)$ there exists an edge $f_j = (v_2, j, v_2^j)$ such that $v_1^i \preceq v_2^j$ and all of the following holds.*
   - *if $label(v_1)$ is of the form $l[]$ or $l\{\}$, then $n = m$ (total term specification)*
   - *if $label(v_1)$ is of the form $l[]$ or $l[[]]$, then for all $v_1^k \preceq v_2^l$ it holds that $k < i$ implies $l < j$ (order)*
   - *if $label(v_1)$ is of the form $l\{\}$ or $l\{\{\}\}$, then $e_i = (v_1, i, v_1^i), e_k = (v_1, k, v_1^k)$ with both $v_1^i \preceq v_2^j$ and $v_1^k \preceq v_2^j$ implies that $i = k$, and therefore $v_1^i = v_1^k$ (injectivity)*
   - *if $e_i = (v_1, i, v_1^i), f_j = (v_2, j, v_2^j)$ and $f_k = (v_2, k, v_2^k)$ with both $v_1^i \preceq v_2^j$ and $v_1^i \preceq v_2^k$ then $j = k$, and therefore $v_2^j = v_2^k$ (minimality)*

To ease the understanding of this definition, the requirements can be described informally as follows.

Order. Given two pairs of mappings, say $v_1 \preceq v_2$ and $v_1' \preceq v_2'$, of two children with a ordered specification, the *order* restriction specifies that if v_1 is less than v_1' by the total order given by the indices of the edges form the parent of these children to them, the mapped nodes must preserve that order (i.e. v_2 must be less than v_2').

Injectivity. Injectivity ensures that no two distinct edges of $G(t_1)$ are simulated by a single edge in $G(t_2)$. This requirements must not be given for the ordered braces case, since the order constraint implies injectivity. That the injectivity is welcome for intuitive programs was already proposed above.

Minimality. Minimality enforces in fact that the mapping of an edge is a mapping in the mathematical sense, i.e. each node v_1 is mapped at most one node v_2. Such a relation is a minimal simulation in the sense of the subset order.

To use the defined relation as a binary predicate, one must use the somewhat cumbersome notation $G(t_1) \preceq G(t_2)$. As a convenient abbreviation for this, $t_1 \preceq t_2$ is used in the following.

Example 9 (Ground Query Term Simulation) *First, considering a simple example of ground query term simulation with the following ground query terms t_1 and t_2:*

$$\begin{aligned} t_1 &:= a\{\{b, b\}\} \\ t_2 &:= a[b, c, b] \end{aligned}$$

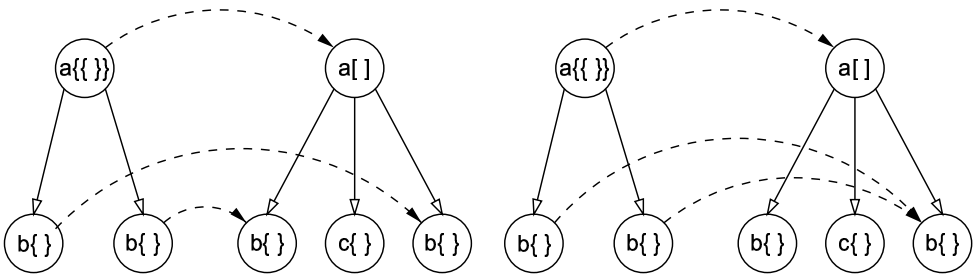


Figure 1: legal and illegal simulation due to injectivity

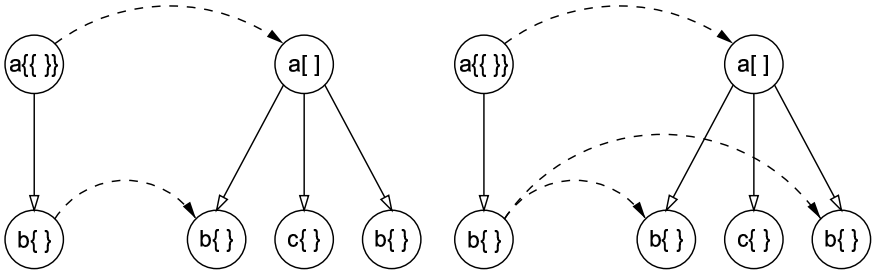


Figure 2: legal and illegal simulation due to minimality

Obviously, t_1 simulates in t_2, and the simulation is

$$S = \{(v_1\langle a\{\{\}\}\rangle, w_1\langle a[]\rangle), (v_2\langle b\{\}\rangle, w_2\langle b\{\}\rangle), (v_3\langle b\{\}\rangle, w_4\langle b\{\}\rangle)\}.$$

But the following simulation is illegal because of the injectivity constraint (see Figure 1).

$$S' = \{(v_1\langle a\{\{\}\}\rangle, w_1\langle a[]\rangle), (v_2\langle b\{\}\rangle, w_2\langle b\{\}\rangle), (v_3\langle b\{\}\rangle, w_2\langle b\{\}\rangle)\}$$

The second example illustrates the minimality constraint. Let t_1 and t_2 be the following ground terms.

$$\begin{aligned} t_1 &:= a\{\{b\}\} \\ t_2 &:= a[b, c, b] \end{aligned}$$

A valid simulation of t_1 in t_2 would be

$$S = \{(v_1\langle a\{\{\}\}\rangle, w_1\langle a[]\rangle), (v_2\langle b\{\}\rangle, w_2\langle b\{\}\rangle)\}$$

But the following simulation is illegal due to the minimality constraint, since $S \subset S'$ (See also Figure 2):

$$S' = \{(v_1\langle a\{\{\}\}\rangle, w_1\langle a[]\rangle), (v_2\langle b\{\}\rangle, w_2\langle b\{\}\rangle), (v_2\langle b\{\}\rangle, w_4\langle b\{\}\rangle)\}$$

A third example illustrates the order constraint. Again, let t_1 and t_2 be the following ground query terms.

$$\begin{aligned} t_1 &:= a[[b, c]] \\ t_2 &:= a[c, b, c] \end{aligned}$$

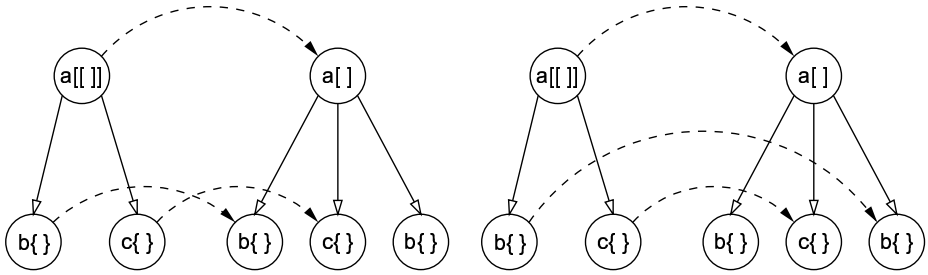


Figure 3: legal and illegal simulation due to order

The single possible simulation of t_1 *in* t_2 *is the following.*

$$S = \{(v_1\langle a[[]]\rangle, w_1\langle a[]\rangle), (v_2\langle b\{\}\rangle, w_3\langle b\{\}\rangle), (v_3\langle c\{\}\rangle, w_4\langle c\{\}\rangle)\}.$$

Whereas S' *is illegal due to the order constraint, since the order of the edges mapped in* t_2 *does not respect the order of* t_1 *(see also Figure 3):*

$$S' = \{(v_1\langle a[[]]\rangle, w_1\langle a[]\rangle), (v_2\langle b\{\}\rangle, w_3\langle b\{\}\rangle), (v_3\langle c\{\}\rangle, w_2\langle c\{\}\rangle)\}.$$

After the simulation unification of ground terms, the non-ground terms are handled next.

4.2 Variables

In this section the consequences from the introduction of variables to simulation unification are discussed. This introduction requires extensions to the ground query term simulation. First, substitutions and grounding substitutions for construct and query terms are defined. Then, the simulation unification of non-ground terms is discussed.

4.2.1 Grounding Substitutions

If variables are introduced, the first important notion is how to replace them and how to transform terms containing variable into ground terms containing no variables. For this, the notion of substitution, the replacement of variables, is introduced. It is useful here because the definition of the semantics of rules can easily be defined once substitutions and their application are declared for both query and construct terms.

Definition 15 (Substitution) *A* Substitution *is a set of variable bindings, written* t/X*, where* X *is a variable and* $t \in \mathcal{T}^d$ *is a data term. For every substitution* σ *it must furthermore hold that there are no two bindings* $t_1/X_1, t_2/X_2 \in \sigma$ *with* $X_1 = X_2$*.*

The notation t/X is understood as standing for "replace every occurrence of X by t" and is best read "t for X".

In the following Greek letters σ, ϕ, ρ are used to name substitutions. For sets of substitutions, uppercase Greek letters Σ, Φ are used.

Definition 16 (Substituted Variables) *Let σ be a substitution. The substituted variables, $vars_s(\sigma)$ are defined as $vars_s(\sigma) := \{X \mid \exists t.\ t/X \in \sigma\}$.*

It is important to separate valid substitutions from other. Here, the name *grounding substitution* is used to state that a substitution is valid for a term in the sense that it replaces all occurring variables of the term. The definition of grounding substitutions for construct terms is given first, since it is simpler than for query terms.

Definition 17 (Grounding Substitution for Construct Term) *Let $t \in \mathcal{T}^c$ be a variable well formed construct term, let σ be a substitution. σ is a* Grounding Substitution *for t, iff $vars_c(t) \subseteq vars_s(\sigma)$.*

The definition of grounding substitution for a query term is a bit more complex, and requires the notion of subterms of a term for a proper handling of the *desc* construct.

Definition 18 (Subterms of Query Term) *Let $t \in \mathcal{T}^q$ be a query term. The* Subterms *of t, $subterms(t)$ is defined as follows.*

- $subterms(\text{“}text\text{”}) = \{\text{“}text\text{”}\}$
- $subterms(l\{\}) = \{l\{\}\}$
- $subterms(l\{\{\}\}) = \{l\{\{\}\}\}$
- $subterms(l[]) = \{l[]\}$
- $subterms(l[[]]) = \{l[[]]\}$
- $subterms(\uparrow id) = subterms(deref(\uparrow id))$
- $subterms(id@t) = subterms(t)$
- $subterms(X) = \{X\}$
- $subterms(X \rightsquigarrow t) = \{X \rightsquigarrow t\}$
- $subterms(l\{t_1, \ldots, t_n\}) = \{l\{t_1, \ldots, t_n\}\ \} \cup \bigcup_{i=1}^{n} subterms(t_i)$
- $subterms(l\{\{t_1, \ldots, t_n\}\}) = \{l\{\{t_1, \ldots, t_n\}\}\ \} \cup \bigcup_{i=1}^{n} subterms(t_i)$
- $subterms(l[t_1, \ldots, t_n]) = \{l[t_1, \ldots, t_n]\} \cup \bigcup_{i=1}^{n} subterms(t_i)$
- $subterms(l[[t_1, \ldots, t_n]]) = \{l[[t_1, \ldots, t_n]]\} \cup \bigcup_{i=1}^{n} subterms(t_i)$

It is important to note that in the cases of $X \rightsquigarrow t$, t is not a subterm of any term. It is just a restriction pattern for the variable X, while only $X \rightsquigarrow t$ is a subterm. This case can be considered as similar to having a single variable X.

With the definition of the subterms, a grounding substitution for a query term can be characterized as follows.

Definition 19 (Grounding Substitution for Query Term) *Let σ be a substitution. In the following, let l denote a term label, let id denote a unique identifier, let X denote a variable and let $t_i \in \mathcal{T}^q$ be $1 \leq i \leq n \in \mathbb{N}$ query terms. σ is a* Grounding Substitution *for a query term $t \in \mathcal{T}^q$ iff the following holds.*

- $vars_q(t) \subseteq vars_s(\sigma)$.
- *if t is of the form $X \rightsquigarrow s$, it must hold that σ is a grounding substitution for s.*
- *if t is of the form $X \rightsquigarrow desc\ s$, then σ is a grounding substitution for s and the existing $t'/X \in \sigma$ has the property that there exists a subterm $t'' \in subterms(t')$ such that $s\sigma \preceq t''$.*
- *if t is of the form $\uparrow id$, then σ is a grounding substitution for $deref(\uparrow id)$.*
- *if t is of the form $id@s$, then σ is a grounding substitution for s.*
- *if t is of the form $l\{t_1, \ldots, t_n\}$ or $l\{\{t_1, \ldots, t_n\}\}$ or $l[t_1, \ldots, t_n]$ or $l[[t_1, \ldots, t_n]]$, then σ is a grounding substitution for $t_1, \ldots, t_n$.*

Example 10 (Grounding Substitution for Query Term) *Let t be the following query term.*

$$t := a[X \rightsquigarrow b[[c]], Y, Z \rightsquigarrow desc\ d]$$

Considering the following three substitutions:

$$\begin{aligned} \sigma &= \{b[b,c]/X, a/Y, a[a[a[a,d]\ ]\ ]/Z\} \\ \rho &= \{b\{b,c\}/X, a[b,c]/Y, d/Z\} \\ \phi &= \{b[b[d],c]/X, b/Y, d[a,d]/Z\} \end{aligned}$$

σ is obviously a grounding substitution for t. It holds that $b[[c]] \preceq b[b,c]$ and $desc\ d \preceq a[a[a[a,d]\ ]\ ]$.
ρ is not a grounding substitution for t, since it does not hold that $b[[c]] \preceq b\{b,c\}$.
ϕ is a grounding substitution. Again, $b[[c]] \preceq b[b[d],c]$ holds as well as $desc\ d \preceq d[a,d]$ (the latter with subterm d, not with $d[a,d]$).

4.2.2 Substitution Application

The *application* of grounding substitutions to construct- and query terms is addressed in the remainder of this section. Because of the grouping construct terms containing *all* or *some n*, it may not be enough to apply a single substitution to a construct term, since all possible bindings of variables resulting from the evaluation of the rule body must be considered. Therefore, the application of substitutions to construct terms is defined with sets of grounding substitutions. For query terms, a single substitution is enough. For this reason the definition of substitution application is simpler for query terms and is presented first.

Definition 20 (Substitution Application) *Let $t \in \mathcal{T}^q$ be a query term, σ a grounding substitution for t. In the following, let l denote a term label, id a unique identifier, X a variable and $t_i \in \mathcal{T}^q$ be $1 \leq i \leq n \in \mathbb{N}$ query terms.*

The application of σ to t, $t\sigma$, is defined as follows.

- *if t is a variable X and $t'/X \in \sigma$, then $t\sigma := t'$.*
- *if t is of the form $X \rightsquigarrow s$ and $t'/X \in \sigma$, then $t\sigma := t'$.*
- *if t is of the form $X \rightsquigarrow desc\ s$ and $t'/X \in \sigma$, then $t\sigma := t'$.*
- *if t is of the form $\uparrow id$, then $t\sigma := id'$, where id' is a new identifier and $\uparrow id'$ points to $id'@deref(\uparrow id)\sigma$.*
- *if t is of the form $id@s$, then $t\sigma := id@s\sigma$.*
- *if t is of the form $l\{t_1, \ldots, t_n\}$ then $t\sigma := l\{t_1\sigma, \ldots, t_n\sigma\}$.*
- *if t is of the form $l\{\{t_1, \ldots, t_n\}\}$ then $t\sigma := l\{\{t_1\sigma, \ldots, t_n\sigma\}\}$.*
- *if t is of the form $l[t_1, \ldots, t_n]$ then $t\sigma := l[t_1\sigma, \ldots, t_n\sigma]$.*
- *if t is of the form $l[[t_1, \ldots, t_n]]$ then $t\sigma := l[[t_1\sigma, \ldots, t_n\sigma]]$.*

Note that the application of a grounding substitution on a query term does not generate a data term but a ground query term.

Example 11 (Substitution Application) *Let t be the following query term.*

$$t := a[X \rightsquigarrow b[[c]], Y, Z \rightsquigarrow desc\ d]$$

The following three substitutions σ, ρ and ϕ are grounding for t. Two of them, namely σ and ϕ, were presented in the last example.

$$\begin{aligned} \sigma &= \{b[b,c]/X, a/Y, a[a[a[a,d]]]/Z\} \\ \rho &= \{b[c]/X, a[b,c]/Y, d/Z\} \\ \phi &= \{b[b[d],c]/X, b/Y, d[a,d]/Z\} \end{aligned}$$

The application of the three substitutions to t is as follows.

$$\begin{aligned} t\sigma &= a[b[b,c], a, a[a[a[a,d]]]] \\ t\rho &= a[b[c], a[b,c], d] \\ t\phi &= a[b[b[d],c], b, d[a,d]] \end{aligned}$$

Definition 21 (Grounding Set) *Let t be a construct or query term. A set Σ of substitutions is a* Grounding Set *for t, iff for all $\sigma \in \Sigma$ it holds that σ is a grounding substitution for t.*

The application of a grounding set to a query term is straightforward: the result of the application of a grounding set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ to t^q is simply the set of ground query terms $\{t^q\sigma_1, \ldots, t^q\sigma_n\}$.

The definition of a grounding set application for grouping construct terms needs more work, since a definition for replacement of the *all* and *some n* constructors needs a grouping of substitutions within a grounding set. The substitutions are always grouped such that those substitutions with the same free variables bindings are applied together to an *all* or *some n*

constructor (i.e. the substitution set is divided in several equivalence classes). If there are more substitution groups with different bindings for the free variables, they are applied in different "branches" of the term - either as children of an ancestor *all* construct or as different data terms in a set. But first, the mechanism of grouping needs to be explained. The substitution set is divided in equivalence classes such that in every equivalence class, the bindings for all variables in a set of variables V are the same:

Definition 22 (Substitution Grouping) *Let Σ be a grounding set, V be a variable set such that $\forall \sigma \in \Sigma.\ V \subseteq vars_s(\sigma)$.*

- *The equivalence relation $\simeq_V \subseteq \Sigma \times \Sigma$ is defined as $\sigma_1 \simeq_V \sigma_2 \iff \forall X \in V.\ X\sigma_1 = X\sigma_2$ for all σ_1, σ_2 in Σ.*
- *The* grouping of Σ on V *is the set of all equivalence classes $\Sigma/\simeq_V$*
- *The equivalence classes of the grouping are accordingly defined as $[\![\sigma]\!] = \{\tau \in \Sigma \mid \tau \simeq_V \sigma\}$*

Example 12 (Substitution Grouping) *Let Σ be the following substitution set.*

$$\Sigma := \{\{a/X, a/Y, a/Z\}, \{a/X, b/Y, b/Z\}, \{b/X, b/Y, b/Z\}, \{b/X, b/Y, a/Z\}\}$$

Considering different groupings, $\Sigma_1 := \Sigma/\simeq_{\{X\}}$, $\Sigma_2 := \Sigma/\simeq_{\{Z\}}$ and $\Sigma_3 := \Sigma/\simeq_{\{X,Y\}}$, the explicit notation of these sets is as follows.

$$\begin{aligned}
\Sigma_1 &= \{ \quad \{\{a/X, a/Y, a/Z\}, \{a/X, b/Y, b/Z\}\}, \quad \{\{b/X, b/Y, b/Z\}, \{b/X, b/Y, a/Z\}\} \quad \} \\
&= \{ \quad [\![\sigma_{a/X}]\!], \quad [\![\sigma_{b/X}]\!] \quad \}
\end{aligned}$$

$$\begin{aligned}
\Sigma_2 &= \{ \quad \{\{a/X, a/Y, a/Z\}, \{b/X, b/Y, a/Z\}\}, \quad \{\{a/X, b/Y, b/Z\}, \{b/X, b/Y, b/Z\}\} \quad \} \\
&= \{ \quad [\![\sigma_{a/Z}]\!], \quad [\![\sigma_{b/Z}]\!] \quad \}
\end{aligned}$$

$$\begin{aligned}
\Sigma_3 &= \{ \quad \{\{a/X, a/Y, a/Z\}\}, \quad \{\{a/X, b/Y, b/Z\}\}, \quad \{\{b/X, b/Y, b/Z\}, \{b/X, b/Y, a/Z\}\} \quad \} \\
&= \{ \quad [\![\sigma_{a/X,a/Y}]\!], \quad [\![\sigma_{a/X,b/Y}]\!], \quad [\![\sigma_{b/X,b/Y}]\!] \quad \}
\end{aligned}$$

For the insertion of several subterms specified by the resolution of an *all* or *some n* constructor, term sequences and their insertion into construct terms are introduced.

Definition 23 (Term Sequences) *Let $t_i \in \mathcal{T}^c$ be $1 \leq i \leq n \in \mathbb{N}$ construct terms. A* Term Sequence *is denoted by $\langle t_1, \ldots, t_n \rangle$.*

Let $t_i \in \mathcal{T}^c$, be $1 \leq i \leq n \in \mathbb{N}$ construct terms and $t \in \mathcal{T}^c$ be a construct term of the form $l\{t_1, \ldots, t_n\}$ or $l[t_1, \ldots, t_n]$. The insertion of a term sequence $seq := \langle s_1, \ldots, s_m \rangle$ as child of $t, insert(t, i, seq)$ is defined as

$$\begin{aligned}
&insert(t, i, seq) := l\{t_1, \ldots, t_i, s_1, \ldots, s_m, t_{i+1}, \ldots, t_n\} \\
or \quad &insert(t, i, seq) := l[t_1, \ldots, t_i, s_1, \ldots, s_m, t_{i+1}, \ldots, t_n] \textit{respectively.}
\end{aligned}$$

Of course, $insert(t, i, seq)$ is a partial function only applicable if $0 \leq i \leq n$. The notion of term sequences is formally not compatible with the subterm grouping: it is illegal to write $t = l\{t_1, \ldots, t_i, \langle s_1, \ldots, s_m \rangle, t_{i+1}, \ldots, t_n\}$. Nevertheless, for simplicity it is assumed that t is legal and stands for $t' = insert(l\{t_1, \ldots, t_n\}, i, \langle s_1, \ldots, s_m \rangle)$ (for subterm grouping [] respectively).

Definition 24 (Grounding Set Application) *First, a restricted version of* Grounding Set Application *is defined. Let $t^c \in \mathcal{T}^c$ be a construct term such that $freevars_c(t^c) = \emptyset$. Let Σ be a grounding set for t^c, i.e. $\Sigma / \simeq_\emptyset = \{[\![\sigma]\!]\}$. The application of Σ to t^c, is defined as $t^c[\![\sigma]\!]$ while the following rules apply.*

Let $t, s \in \mathcal{T}^c$ be construct terms, let l denote a term label, X a variable, id a unique identifier and let $t_i \in \mathcal{T}^c$ be $1 \leq i \leq n \in \mathbb{N}$ construct terms.

1. *if t is of the form $l\{t_1, \ldots, t_n\}$, then $t[\![\sigma]\!] := l\{t_1[\![\sigma]\!], \ldots, t_n[\![\sigma]\!]\}$.*
2. *if t is of the form $l[t_1, \ldots, t_n]$, then $t[\![\sigma]\!] := l[t_1[\![\sigma]\!], \ldots, t_n[\![\sigma]\!]]$.*
3. *if t is of the form $\uparrow id$, then $t[\![\sigma]\!] :=\uparrow id'$ while id' points to the term $id'@deref(id)[\![\sigma]\!]$.*
4. *if t is of the form $id@s$, then $t[\![\sigma]\!] := id'@s\Sigma$ with a new identifier id'.*
5. *if t is of the form all s and $V = freevars_c(s)$, then $t[\![\sigma]\!] := \langle t[\![\tau_1]\!], \ldots, [\![\tau_k]\!] \rangle$ where $\{\tau_i, \ldots, \tau_k\} = [\![\sigma]\!] / \simeq_V$*
6. *if t is of the form some m s and $V = freevars_c(s)$, then $t[\![\sigma]\!] := \langle t[\![\tau_1]\!], \ldots, [\![\tau_k]\!] \rangle$ where $\{\tau_i, \ldots, \tau_k\} \subseteq [\![\sigma]\!] / \simeq_V$ and $k = min(index([\![\sigma]\!] / \simeq_V), m)$.*
7. *if t is of the form X, then $X[\![\sigma]\!] = X\sigma$.*

The unrestricted Grounding Set Application *is as follows. Let t^c be a construct term. Let $V = freevars_c(t^c)$. Let Σ be a grounding set for t^c.*

$$t^c\Sigma := \{t^c[\![\sigma]\!] \mid [\![\sigma]\!] \in \Sigma / \simeq_V\}$$

Due to the restriction of construct terms to those where $freevars(t^c) = \emptyset$ holds, it is ensured that in case 7, where the variables are finally replaced, any substitution can be taken from the given set $[\![\sigma]\!]$, since these substitutions have been grouped in cases 5 or 6.

In a second approach, it is simple to lift this restriction, but by lifting it, the application of a grounding set to a construct term yields in the general case a set of data terms, not a single data term.

Example 13 (Grounding Set Application) *Considering the two following construct terms t_1 and t_2*

$$\begin{aligned} t_1 &:= a\{b\{X\}, c[X, Y]\ \} \\ t_2 &:= a\{all\ b\{X, all\ c\{Y\}\ \}\ \}, \end{aligned}$$

the following substitution sets are grounding for t_1 as well as for t_2.

$$\begin{aligned} \Sigma &= \{\{f/X, h/Y\}\} \\ \Theta &= \{\{f[a]/X, h/Y\}, \{f/X, h[b]/Y\}\} \\ \Phi &= \{\{f/X, h/Y\}, \{f[a]/X, h/Y\}, \{f/X, h[b]/Y\}\} \end{aligned}$$

Now, the application of the grounding sets to t_1 results in

$$\begin{array}{lll} t_1\Sigma & = \{ & a\{b\{f\}, c[f,h]\ \}\ \} \\ t_1\Theta & = \{ & a\{b\{f[a]\}, c[f[a],h]\ \}, \\ & & a\{b\{f\}, c[f,h[b]\ \}\ \} \\ t_1\Phi & = \{ & a\{b\{f\}, c[f,h]\ \}, \\ & & a\{b\{f[a]\}, c[f[a],h]\ \}, \\ & & a\{b\{f\}, c[f,h[b]\ ]\ \}\ \} \end{array}$$

But considering the application of the same grounding sets to t_2, we get

$$\begin{array}{lll} t_2\Sigma & = \{ & a\{b\{f, c\{h\}\ \}\ \}\ \} \\ t_2\Theta & = \{ & a\{b\{f[a], c\{h\}\ \}, b\{f, c\{h[c]\}\ \}\ \}\ \} \\ t_2\Phi & = \{ & a\{b\{f, c\{h\}, c\{h[b]\}\ \}, b\{f[a], c\{h\}\ \}\ \} \end{array}$$

The groupings of the substitution sets in the third application is calculated as follows. At the beginning, Φ is grouped by the free variables of t_2:

$$\Phi/\simeq_\emptyset = \{[\![\Phi']\!]\}$$

At the occurrence of the first all, the set $[\![\Phi']\!]$ is grouped as

$$\begin{array}{rcl} \Phi'/\simeq_{\{X\}} & = & \{\{\{f/X, h/Y\}, \{f/X, h[b]/Y\}\}, \{\{f[a]/X, h/Y\}\}\} \\ & = & \{[\![\Phi_1]\!], [\![\Phi_2]\!]\} \end{array}$$

since X is the only free variable in the term $b\{X, all\ c\{Y\}\ \}$. Then, a sequence is inserted with $[\![\Phi_1]\!]$ and $[\![\Phi_2]\!]$ applied to each sequence element:

$$a\{\langle b\{X, all\ c\{Y\}[\![\Phi_1]\!], b\{X, all\ c\{Y\}[\![\Phi_2]\!]\rangle\}$$

Now, it is obviously not of any importance which substitution is taken from $[\![\Phi_1]\!]$ to look up a binding for X, because all substitutions in this set contains the same binding f/X. The term can be simplified as

$$a\{b\{h, all\ c\{Y\}llbracketPhi_1]\!], b\{h[a], all\ c\{Y\}[\![\Phi_2]\!]\}$$

again, the substitution sets Φ_1 and Φ_2 has to be grouped by the bindings of the only free variable in $c\{Y\}$, namely Y:

$$\begin{array}{rcl} [\![\Phi_1]\!]/\simeq_{\{Y\}} & = & \{\{\{f/X, h/Y\}\}, \{\{f/X, h[b]/Y\}\}\} \\ & = & \{[\![\Phi_{11}]\!], [\![\Phi_{12}]\!]\} \\ [\![\Phi_2]\!]/\simeq_{\{Y\}} & = & \{\{\{f[a]/X, h/Y\}\}\} \\ & = & \{[\![\Phi_{21}]\!]\} \end{array}$$

And again, a term sequence is inserted to replace the all construct.

$$a\{b\{h, \langle c\{Y\}[\![\Phi_{11}]\!], c\{Y\}[\![\Phi_{12}]\!]\rangle, b\{h[a], \langle c\{Y\}[\![\Phi_{21}]\!]\rangle\}$$

The resolving of the sequences and the application of the substitution finally results in the term

$$a\{b\{f, c\{h\}, c\{h[b]\}\ \}, b\{f[a], c\{h\}\ \}.$$

4.2.3 Term Simulation and Simulation Unifier

With the grounding substitutions and substitution application defined, it is now possible to formalize the notion of matching a non-ground query term against a non-ground, possibly grouping construct term by simulation unification. Intuitively, a query term t^q simulates into a construct term t^c if there exist at least one substitution set Σ that is a grounding set for both t^q and t^c and the ground query terms resulting from the application of Σ to t^q simulates into one of the ground construct terms resulting from the application of Σ to t^c. Note that the statement is plural since application of a substitution set yields more than one construct and query term. A grounding substitution set that fulfills these condition is called *Simulation Unifier*.

Definition 25 (Term Simulation and Simulation Unifier) *Let $t^q \in \mathcal{T}^q$ be a query term, let $t^c \in \mathcal{T}^c$ be a construct term. The simulation $t^q \preceq t^c$ is valid with* Simulation Unifier Σ *iff Σ is a grounding substitution set for t^q and for t^c and the following holds.*

$$\forall s^q \in t^q\Sigma \; \exists s^c \in t^c\Sigma.\; s^q \preceq_{su} s^c$$

Example 14 (Term Simulation and Simulation Unifier) *Consider the following simulation problem:*

$$a\{b, X, c\{\{Y\}\}\ \} \preceq a[b, c\{a\}, c\{Z\}]$$

The following substitution set presents a possible solution to this problem.

$$\Sigma = \{\sigma\} = \{\{c\{a\}/X, e/Y, e/Z\}\}$$

Because

$$a\{b, X, c\{\{Y\}\}\ \}\sigma = a\{b, c\{a\}, c\{\{e\}\}\ \} =: t_1$$
$$a[b, c\{a\}, c\{Z\}\Sigma = \{a[b, c\{a\}, c\{e\}]\} =: \{t_2\}$$

And obviously $t_1 \preceq t_2$.

Considering another example with grouping construct terms,

$$a\{\{b, c[[X]]\ \}\} \preceq a\{all\ Y, Z\}$$

the substitution set Σ is one of infinite possible solutions.

$$\Sigma = \{\sigma_1, \sigma_2\} = \{\{e/X, a/Y, b/Z\}, \{d/X, c[d, e]/Y, b/Z\}\}$$

Because

$$a\{\{b, c[[X]]\ \}\}\sigma_1 = a\{\{b, c[[e]]\ \}\} =: u_1$$
$$a\{\{b, c[[X]]\ \}\}\sigma_2 = a\{\{b, c[[d]]\ \}\} =: u_2$$
$$a\{all\ Y, Z\}\Sigma = a\{a, c[d, e], b\} =: u_3$$

And obviously $u_1 \preceq u_3$ and $u_2 \preceq u_3$.

In the next section, the simulation unification calculus is presented as a part of an algorithm to find a simulation unifier if one exists.

4.3 A Calculus for Simulation Unification

The formal definition of simulation unification above defines which conditions a substitution set must fulfill to be a simulation unifier. In this section, an algorithm in the form of a calculus of logical equivalences is presented for the computation of a simulation unifier. In the current state of research however, this algorithm cannot be applied to grouping construct terms, that is, construct terms containing *all* or *some*. Therefore, it is mandatory to resolve every grouping construct term before applying the unification of query and construct term. In other words, before the simulation unifier of a query and a grouping construct term can be determined, the body of the respective rule must be evaluated and the instances of the rule head must be created.

The calculus for simulation unification is a constraint calculus which returns a constraint store. If there is a solution to the simulation problem, this returned constraint store can be transformed into a substitution set which is a simulation unifier for the simulation problem. Otherwise, if there is no solution to the simulation problem, the returned constraint store is $False$ and the simulation unifier becomes the empty set, representing a failed unification. Note however that the calculus does not require this transformation to occur. This is important since the chaining calculus extends the simulation unification calculus such that simulation steps and chaining steps might be intertwined as needed.

4.3.1 Constraint Stores

The simulation unification calculus operates on constraints to find a solution to the problem $t^q \preceq t^c$. These constraints are solved by decomposing them into smaller constraints until a solution is found, that is, a conjunction of non-decomposable constraints is found in the constraint store whose satisfaction alone would satisfy the whole constraint store. Since the solution must be a grounding set, the next step is to present a transformation of a constraint store into a substitution set form. The given definition for constraint stores includes more constraint types than required for simulation unification alone, since the chaining calculi and algorithms introduced in Section 5 operate on the same constraints as well.

Definition 26 (Constraint Stores) *Let $t^q \in \mathcal{T}^q$ be a query term, let $t^c \in \mathcal{T}^c$ be a construct term, let $Q \in QFormula$ be a query formula, let $C_i \in CStore$ be $1 \leq i \leq n \in \mathbb{N}$ Constraint Stores.*

A Constraint Store *$cs \in CStore$ is build from the atomic constraints.*

- $t^q \preceq_{su} t^c \in CStore$
- $\langle Q \rangle \in CStore$
- $False \in CStore$
- $True \in Cstore$

and the following non atomic constraints.

- $(C_1 \mid C_2) \in CStore$
- $\bigvee_{i=1}^{n} C_i \in CStore.$
- $\bigwedge_{i=1}^{n} C_i \in CStore.$

As before, an infix notation for constraints ($C_1 \wedge C_2, C_1 \vee C_2$) is used where convenient. The meaning of each constraint constructor can be described informally as follows.

Boolean Values and Connectives. The simple boolean values *True* and *False* as well as the boolean connective $\wedge$ and $\vee$ are understood as expected. The standard logic equivalences on boolean formulas are also valid on constraint stores (i.e. $True \vee a \Leftrightarrow True$ and so on). Furthermore, the empty connectives are valid with the following interpretation.

$$\bigwedge_{i=1}^{0} \Leftrightarrow True \quad \bigvee_{i=1}^{0} \Leftrightarrow False$$

Simulation Constraint. A simulation constraint, written $t^q \preceq_{su} t^c$, requires that every variable occurring in t^q or t^c is bound such that a ground query simulation of t^q in t^c can be found. Note the index in $\preceq_{su}$ that is *not* the same as the term simulation $\preceq$ defined in Section 4.2.3. It is a syntax notation for a constraint that might be further decomposed in order to calculate a valid substitution Σ for term simulation.

Query Constraint. A query constraint written $\langle Q \rangle$ is a constraint consisting of a query formula that has to be evaluated with respect to the current program. It describes a folded query that might be unfolded at a later time. This constraint is needed for chaining.

Dependency Constraint. A dependency constraint $(C_1 \mid C_2)$ influences the constraint resolution order. It can be read as "C_1 after C_2" or "C_1 depends on C_2" and states that C_1 may only be evaluated if the evaluation of C_2 did not fail in the sense that it is not equivalent to $False$. How the dependency constraint is processed is discussed in the dependency resolution rule below. This constraint type is also needed for chaining.

The following function *disjuncts* constructs a set of all disjuncts of the disjunctive normal form of a constraint store:

$$\begin{array}{lcl}
disjuncts(t^q \preceq_{su} t^c) & = & \{t^q \preceq_{su} t^c\} \\
disjuncts(\langle Q \rangle) & = & \{\langle Q \rangle\} \\
disjuncts((C \mid D)) & = & \{(C \mid D)\} \\
disjuncts(False) & = & \{False\} \\
disjuncts(True) & = & \{True\} \\
disjuncts(\bigvee_{i=1}^{0} C_i) & = & \{False\} \\
disjuncts(\bigvee_{i=1}^{n} C_i) & = & \bigcup\{disjuncts(C_i)\} \text{ for } n > 0 \\
disjuncts(\bigwedge_{i=1}^{0} C_i) & = & \{True\} \\
disjuncts(\bigwedge_{i=1}^{n} C_i) & = & \{\bigwedge_{i=1}^{n} C_i' \mid C_i' \in disjuncts(C_i)\} \text{ for } n > 0
\end{array}$$

With this constraint store definition and the *disjunct* functions, the substitution induced by a constraint store is defined as follows.

Definition 27 (Substitution Set Induced by Constraint Store) *Let $C \in CStore$ be a constraint store. The substitution set induced by C, $subst(C)$ is defined as follows.*
At first, let

$$disjuncts(C) = \{\bigwedge_{j=1}^{m_1} C_{1j}, \ldots, \bigwedge_{j=1}^{m_n} C_{nj}\}$$

be the disjuncts of C. Then, let

$$S_i := \{t/X \mid \exists 1 \leq j \leq m_i.\ C_{ij} = X \preceq_{su} t\}.$$

The substitution set induced by C is then

$$subst(C) := \{S_1, \ldots, S_n\}.$$

An attentive reader will note that it may be possible by this definition to have a constraint store inducing an invalid substitution. This may happen if there are two constraints of the form $X \preceq_{su} t_1$ and $X \preceq_{su} t_2$ in the same disjunct. For those cases however, the simulation unification calculus specifies a consistency rule resolving inconsistencies to $False$ in Section 4.3.3. Hence, the case where $X \preceq_{su} t_1$ and $X \preceq_{su} t_2$ occur in the same disjunct can never emerge as a result of the calculus if the rules are applied as long as possible.

4.3.2 Decomposition Rules

There is a set of rules that are applicable to constraint stores in order to find a solution. A part of these rules are decomposition rules. The aim of these rules is to simplify a constraint into smaller constraints. For the definition of the rules, the following mappings are needed.

Definition 28 (Subterm Mappings) *Given two natural numbers, $n, m \in \mathbb{N}$, we define the following set of mappings.*

- $\Pi(n, m)$ *is the set of all total injective mappings* $\pi : \{1, \ldots, n\} \to \{1, \ldots, m\}$
- $\Pi_m(n, m) := \{\pi \in \Pi(n, m) \mid \pi \text{ is monotonic}\}$
- $\Pi_s(n, m) := \{\pi \in \Pi(n, m) \mid \pi \text{ is surjective}\}$
- $\Pi_{m+s}(n, m) := \Pi_m(n, m) \cap \Pi_s(n, m)$

Note that if $n > m$, there are no monotonic mappings. If $n < m$, then there are no surjective mappings. Thus, if $n \neq m$, there are no mappings that are at the same time surjective and monotonic.

The rules are presented here as equivalences applicable to small portions of the constraint store. The decision which rule to apply and where to find the portion of the constraint store where the rule applies must be made by a constraint resolution algorithm.

Brace Incompatibility. The first set of rules presented here handles the failure of unification due to incompatibility of braces (i.e. subterm grouping). In the cases of brace incompatibility, the constraint can be simplified to $False$.

$$\frac{l_1[t_1, \ldots, t_n] \preceq_{su} l_2\{s_1, \ldots, s_m\}}{False} \qquad \frac{l_1[[t_1, \ldots, t_n]] \preceq_{su} l_2\{s_1, \ldots, s_m\}}{False}$$

Left Term Empty. This set of rules considers unifications where the left term is empty in the sense that it does not specify any children. These cases can all be resolved to either $True$ or $False$, depending on the emptiness of the right term and the subterm grouping of the left term. Depending on whether the left term specifies partial or total subterm grouping the simulation constraint can be reduced to merely $False$ or a Term Label Match test.

Let $m \geq 0$ and $l \geq 1$:

$$\frac{l\{\{\}\} \preceq_{su} l\{s_1, \ldots, s_m\}}{True} \qquad \frac{l\{\{\}\} \preceq_{su} l[s_1, \ldots, s_m]}{True} \qquad \frac{l[[]] \preceq_{su} l[s_1, \ldots, s_m]}{True}$$

$$\frac{l\{\} \preceq_{su} l\{s_1, \ldots, s_l\}}{False} \qquad \frac{l\{\} \preceq_{su} l[s_1, \ldots, s_l]}{False} \qquad \frac{l[] \preceq_{su} l[s_1, \ldots, s_l]}{False}$$

$$\frac{l\{\} \preceq_{su} l\{\}}{True} \qquad \frac{l\{\} \preceq_{su} l[]}{True} \qquad \frac{l[] \preceq_{su} l[]}{True}$$

Decomposition. This rule set handles the general case where no simple resolution to $True$ or $False$ can be found directly. The root label is eliminated by the rules if the labels are the same. Other cases cannot be decomposed. The decomposition rules consider all possible matchings of subterms placed in an *or*. Note that if there is no mapping $\Pi_x(n, m)$, then the result of the decomposition is an empty *or*, which is interpreted as $False$ (see above).

Let $n, m \geq 1$.

$$\frac{l\{\{t_1, \ldots, t_n\}\} \preceq_{su} l\{s_1, \ldots, s_m\}}{\bigvee_{\pi \in \Pi(n,m)} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}} \qquad \frac{l\{\{t_1, \ldots, t_n\}\} \preceq_{su} l[s_1, \ldots, s_m]}{\bigvee_{\pi \in \Pi(n,m)} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}}$$

$$\frac{l\{t_1, \ldots, t_n\} \preceq_{su} l\{s_1, \ldots, s_m\}}{\bigvee_{\pi \in \Pi_s(n,m)} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}} \qquad \frac{l\{t_1, \ldots, t_n\} \preceq_{su} l[s_1, \ldots, s_m]}{\bigvee_{\pi \in \Pi_s(n,m)} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}}$$

$$\frac{l[[t_1, \ldots, t_n]] \preceq_{su} l[s_1, \ldots, s_m]}{\bigvee_{\pi \in \Pi_m(n,m)} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}} \qquad \frac{l[t_1, \ldots, t_n] \preceq_{su} l[s_1, \ldots, s_m]}{\bigvee_{\pi \in \Pi_{m+s}(n,m)} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}}$$

Label Mismatch. In the case that $l_1 \neq l_2$, the unification fails in any case left open. Let $n, m \in \mathbb{N}$.

$$\frac{l_1\{\{t_1,\ldots,t_n\}\} \preceq_{su} l_2\{s_1,\ldots,s_m\}}{False} \qquad \frac{l_1\{t_1,\ldots,t_n\} \preceq_{su} l_2\{s_1,\ldots,s_m\}}{False}$$

$$\frac{l_1\{\{t_1,\ldots,t_n\}\} \preceq_{su} l_2[s_1,\ldots,s_m]}{False} \qquad \frac{l_1\{t_1,\ldots,t_n\} \preceq_{su} l_2[s_1,\ldots,s_m]}{False}$$

$$\frac{l_1[[t_1,\ldots,t_n]] \preceq_{su} l_2[s_1,\ldots,s_m]}{False} \qquad \frac{l_1[t_1,\ldots,t_n] \preceq_{su} l_2[s_1,\ldots,s_m]}{False}$$

$\rightsquigarrow$ Elimination. The standard decomposition rules are not applicable in the case of a $\rightsquigarrow$ construct. The $\rightsquigarrow$ construct was said to constrain the possible bindings of a variable X. This is done by introducing an additional simulation constraint that specifies every binding of X must simulate in the term on the left hand side of $\rightsquigarrow$.

$$\frac{X \rightsquigarrow t \preceq_{su} s}{t \preceq_{su} s \wedge t \preceq_{su} X \wedge X \preceq_{su} s}$$

Descendant Elimination. The *desc* construct specifies a pattern that may match at arbitrary depth. Hence, every subterm of a term matching *desc* t must be tested. Let $m \geq 0$.

$$\frac{desc\ t \preceq_{su} l\{s_1,\ldots,s_m\}}{t \preceq_{su} l\{s_1,\ldots,s_m\} \vee \bigvee_{i=1}^{m} desc\ t \preceq_{su} s_i}$$

$$\frac{desc\ t \preceq_{su} l[s_1,\ldots,s_m]}{t \preceq_{su} l[s_1,\ldots,s_m] \vee \bigvee_{i=1}^{m} desc\ t \preceq_{su} s_i}$$

4.3.3 Consistency Verification

Some combinations of constraints makes it necessary to test whether the sum of them is still satisfiable. This is the case if two simulation constraints contain common variables. The consistency rules are applied at the insertion of new constraints into a nonempty constraint store.

Consistency. The consistency rule implies that every substitution induced by a satisfiable constraint store never contains two bindings for a same variable. This rule is applicable if in a conjunction of constraint a variable must simulate into two terms.

$$\frac{X \preceq_{su} t_1 \ \wedge \ X \preceq_{su} t_2}{X \preceq_{su} t_1 \wedge t_1 \preceq_{su} t_2 \wedge t_2 \preceq_{su} t_1}$$

Transitivity. The transitivity rule guarantees that the binding of a variable stays the same over all constraints of a conjunction. If a variable is given an explicit upper bound, this bound is propagated through all other right hand terms of simulation constraints containing this variable. There, the notation $t[t'/X]$ denotes the replacement of the variable X by t'. Note that this replacement notation is used on the free variables of construct terms only and the replacement is hence safe.

$$\frac{t_1 \preceq_{su} t_2 \wedge X \preceq_{su} t_3 \qquad X \in freevars_c(t_2)}{t_1 \preceq_{su} t_2[t_3/X] \wedge X \preceq_{su} t_3}$$

5 Rule Chaining and Program Evaluation

The three chaining calculi are building upon the rules from Section 4.3. These rules specified the decomposition of terms within a simulation constraint $\preceq_{su}$. What is missing for the evaluation of an Xcerpt program are constraint solving rules for chaining with program rules. The chaining calculi were designed to resemble the SLD resolution ([Av82]), which is itself a restriction of SL resolution ([KK71]) to definite horn clauses.[1] Nevertheless, some special properties of the language Xcerpt influenced the design of the chaining calculi as well.

5.1 Particularities of Xcerpt and Consequences

Altogether, three particular properties can be identified which have strong influence on the evaluation calculi of Xcerpt.

Grouping constructs *all* and *some*. Since the grouping constructs are an integral part of the calculus, they should be supported within the evaluation calculus. An external support like Prolog's *setof* and *bagof* predicates could be unfavorable, since Xcerpt programs make extensive use of the grouping constructs.

High branching level. In traditional logic programming, there is one element of nondeterminism that imply a branching of a proof: the selection of the program rule to unfold a query with. The other element of nondeterminism, the selection order of the queries or predicates within a rule body to unfold, does not lead to a branching.

The simulation unification of Xcerpt, with its simulation constraint calculus for partial and unordered query specifications introduces another element of nondeterminism that branches a proof. The unification of terms in general leads to several distinct unifiers where classical unification yields only one possible unification.

This makes it necessary for chaining to rely on constraint solving techniques which can handle such a high level of branching more efficiently.

Constraint calculus. The algorithm for simulation unification is formulated as an incomplete constraint solving calculus. It is hence desirable to have a formalization of chaining that is consistent with the simulation unification calculus and fits well into it.

To summarize, Xcerpt's model of semistructured data and the nonstandard unification mechanism based on constraint solving makes a classical approach to chaining in Xcerpt like collecting unifiers to find a solution difficult.

[1] The SL resolution itself is a variant of *Model Elimination* ([Lov68, Lov69])

5.2 "All at Once" or "One at Once"?

In the following, two different calculus approaches are presented, named "All at Once" and "One at Once". A goal driven forward chaining variant of "All at Once" calculus is presented here as well. This variant arises from a slight modification of the "All at Once" chaining rule.

The "One at Once" and "All at Once" calculi basically differ in that "One at Once" does not permit disjunctions to appear in constraint stores. The aim is just like with SLD resolution to follow only one proof path at a time. "All at Once" on the other hand, allows the occurrence of disjunctions and offers hence the possibility to follow more than one proof path and up to all proof paths at a time.

On a first glance, it could appear that potentially following all proof paths is the same as a breadth first search in the tree of all proof paths. However, this is not the case: while the "One at Once" calculus formalization supports well the depth-first search strategy within the tree of all proof paths, the "All at Once" calculus is able to integrate all search strategies equally well. By unfolding query constraints within one conjunction, a depth-first search can be performed, and by switching the conjunction in which query constraints are unfolded every time, a breadth-first search is performed: the selection algorithm for query constraint unfolding determines the search strategy. It is hence easy to use nontrivial search strategy such as A* with an "All at Once" calculus.

In addition to this, the occurrence of grouping constructs *all* and *some* make it necessary to collect all proof paths of a certain subtree. This task is easier performed in an "All at Once" formalization than in an "One at Once" approach. A "One at Once" style needs a meta rule that is "external" to the calculus, since only one calculus path is visible at a time within the calculus itself. There is no possibility to reason on sibling or child proof paths within a "One at Once" approach.

5.2.1 Program Evaluation

The calculi presented here for simulation unification and rule chaining aim to present a formalization of program evaluation. However, the evaluation of a program depends on the type of chaining calculus. More precisely, the constraint solving algorithms differ between the "One at Once" and "All at Once" chaining calculi, while the algorithm for program evaluation, once given the algorithms for constraint solving, is the same and can be specified for a given program P and calculus $\mathcal{C}$ as follows.

procedure *eval*(Program P, Calculus $\mathcal{C}$)
  foreach $(t^c \leftarrow Q)$ **in** $goals(P)$ **do**
    if t^c is grouping **then**
      SubstitutionSet $\Sigma = subst(\mathcal{C}.solveall(\langle Q \rangle))$
    else
      SubstitutionSet $\Sigma = subst(\mathcal{C}.solve(\langle Q \rangle))$
    Set(Term) $S = t^c\Sigma$
    if $S \neq \emptyset$ **then**
      Term t **in** S
      out t

A calculus for this program evaluation algorithm must define two constraint solving functions *solve* and *solveall*. These function must return a constraint store that either represents one solution to the constraint problem (*solve*) or all possible solutions (*solveall*) to a constraint problem. The remainder of this section investigates the definitions of the functions *solve* and *solveall* for each calculus type.

5.2.2 Common Calculus Rules

For each of the three calculi, different unfolding rules must be specified. Besides these style-specific rules, there is a set of chaining rules that is common for all calculi.

For the specification of the chaining rules, the following convenient abbreviations are made:

- $R := rules(P)$ for the currently processed program P,
- $R_n := rules_{ng}(P)$ for the same program P,
- $R_g := rules_g(P)$ and
- $T := Terms(P)$.

Likewise, the following conventions are introduced for the rule notation.

- t denotes a data term,
- t^c denotes a construct term,
- t^q denotes a query term,
- Q denotes a query formula and
- C denotes a constraint store.

In the following rules, it is always assumed that the variables in the current constraint store and the variables of the selected program rules are disjunct. Where this is not the case, it is implicitly assumed that a consistent variable renaming on the program rules or the constraint store takes place to make the variable sets disjunct.

Boolean Connectives of Query Formulas. The folded query constraint is linear. To match a conjunction of query formulas $Q_1 \wedge \ldots \wedge Q_n$ is the same as a conjunction of the matching of each query formula Q_i. The same holds of course for disjunctions. Let $Q_i \in QFormula$ be $1 \leq i \leq n \in \mathbb{N}$ query formulas.

$$\frac{\langle Q_1 \wedge \ldots \wedge Q_n \rangle}{\langle Q_1 \rangle \wedge \ldots \wedge \langle Q_n \rangle} \qquad \frac{\langle Q_1 \vee \ldots \vee Q_n \rangle}{\langle Q_1 \rangle \vee \ldots \vee \langle Q_n \rangle}$$

all **and** *some* **Incompleteness.** The current implementation of the simulation unification is incomplete in the case that the term on the right hand side of $\preceq_{su}$ has the form *all* t or *some* n t in the sense that there are no decomposition rules handling such terms. The *all* and *some* constructs are commonly avoided in substitution generating simulation unification by a forward chaining step such that a grounding substitution set can be applied to the respective grouping construct term such that no more grouping constructs occur.

Nevertheless, at least a refutation test can be achieved by simulation unification with grouping construct terms on the right hand side. If t_g^c is a grouping construct term, the simulation constraint $t^q \preceq_{su} t_g^c$ can be seen as a test such that the rule is not triggered if no simulation of the query term in the construct term is possible. By this, a costly evaluation of a grouping rule that have no possibility to fulfill the query is avoided. When an grouping construct is encountered in a simulation constraint, the simulation constraint is reduced to $True$ or some satisfiable constraints on the variables outside the scope of grouping constructs. The unfolding then passed the pre-test, and the grouping rule body is evaluated.

$$\frac{t_1 \preceq_{su} all\ t_2}{True} \qquad \frac{t_1 \preceq_{su} some\ n\ t_2}{True}$$

Together with this test, the dependency constraints can be resolved as follows.

Dependency Resolution. The resolution of dependency uses the incomplete decomposition of *all* and *some* to test if the unfolding is valid. If the unfolding is valid, it starts recursively a full constraint solving with the constraint store on the right hand side of "|". The collected answer substitutions are then applied as substitution set to instantiate the simulation unification test.

$$\frac{(t^q \preceq_{su} t^c \mid \langle Q \rangle)}{\bigvee_{t \in t^c \Sigma} t^q \preceq_{su} t}$$

The substitution set Σ is defined as follows.

$$\Sigma := subst(solveall(t^q \preceq_{su} t^c \wedge \langle Q \rangle)$$

It is nevertheless only evaluated if $t^q \preceq_{su} t^c$ can not be reduced to $False$. The result of this test can be inserted in the call of *solveall* instead of the simulation constraint $t^q \preceq_{su} t^c$ such that it is not reduced twice. If the test fails, the substitution set Σ is set as $\Sigma = \emptyset$, such that $\bigvee_{t \in t^c \Sigma} t^q \preceq_{su} t$ is equivalent to the empty or connective, itself equivalent to $False$.

The double line emphasizes that between the premise and the conclusion of the rule there are (a considerable amount of) intermediate calculation steps, caused by the recursive call to *solveall*. How *solveall* is defined is discussed in the following sections 5.3.2 and 5.4.2 and depends on the calculus variant used.

5.3 "One at Once"

The "One at Once" calculus is similar to SLD resolution. The calculus follows one proof path at a time in the formalization and leaves open several nondeterministic choices. The implementation uses choice points and backtracking just as SLD.

"One at Once" has the advantage that it only needs to consider a single conjunctive path at a time. On the other hand, occurrences of grouping constructs externally interrupt the evaluation by requiring an auxiliary application of the calculus.

5.3.1 "One at Once" Calculus Rules

The specification of the calculus rules for "One at Once" consists of three additional rules for each possible rule type and for data terms. The folded query can be unfold with a simple data term or with a non-grouping or grouping rule; these two cases must be considered separately. Note that the separate handling of the different chaining types is made possible by the restriction of the calculus to single proofs. For the "All at Once" calculus style, all possible chainings must be handled at once in one rule.

Query unfolding with data term This rules unfolds a query with a data term, i.e. it tries to match a query against a given data term in the program P.

$$\frac{\langle t^q \rangle \qquad t \in T}{t^q \preceq_{su} t}$$

Query Unfolding with non-grouping Rule The query term is matched against a rule head. The resulting constraint is the simulation constraint of query and rule head together with the constraint that the body must be unfold (i.e. must be fulfilled) as well.

$$\frac{\langle t^q \rangle \qquad (t^c \leftarrow Q) \in R_n}{t^q \preceq_{su} t^c \wedge \langle Q \rangle}$$

Query unfolding with Grouping Rule In the case that a folded query constraint and a grouping Xcerpt rule is selected, a dependency constraint is introduced that continues the evaluation of the body under the condition that the simulation constraint on the left hand side can be satisfied. The evaluation of the rule body is handled in a separate calculation, as specified by the dependency resolution rule.

$$\frac{\langle t^q \rangle \qquad (t^c \leftarrow Q) \in R_g}{(t^q \preceq_{su} t^c \mid \langle Q \rangle)}$$

Disjunctive Split The "One at Once" calculus follows only one proof at a time. Constraints stores with disjunctions represent several proof paths to follow. The peculiarity is here that the outcome of a rule application may also contain disjunctions, as most prominent in the case of the decomposition rules. It is hence necessary in the "One at Once" approach to split constraint stores with disjunctions into constraint stores with conjunctions only. By this, the additional branching of Xcerpt's simulation unification compared to classical logic programming is introduced into a single proof search calculus.

Let $C_1 \vee \ldots \vee C_n$ be the disjunctive normal form of a constraint store C. The following rule is called "Disjunctive Split" and creates one proof path per disjunct C_i.

$$\frac{C_1 \vee \ldots \vee C_n}{C_1 \mid \ldots \mid C_n}$$

5.3.2 "One at Once" Evaluation Algorithms

To understand the constraint solving functions *solve* and *solveall*, the notion of a calculus tree is introduced. The branches of this tree are the single proofs. To follow an edge in the calculus tree is equivalent to follow an application of the rules of the calculus to a constraint, itself leading into one or several constraints.

Definition 29 (Calculus Tree) *Let P be a program and C be a constraint store. A* Calculus Tree for C in P *is build as follows.*

1. *The tree is initialized with C as root node.*
2. *If D is the label of a tree node, then a child node with label E is attached to this node for every rule of the calculus $\frac{D' \quad P_1 \ldots P_n}{E'}$ if $P_1 \ldots P_n$ are valid premises and if D' is a sub formula of D. The formula E is $D[E'/D']$, that is D with the sub formula D' replaced by E'.*

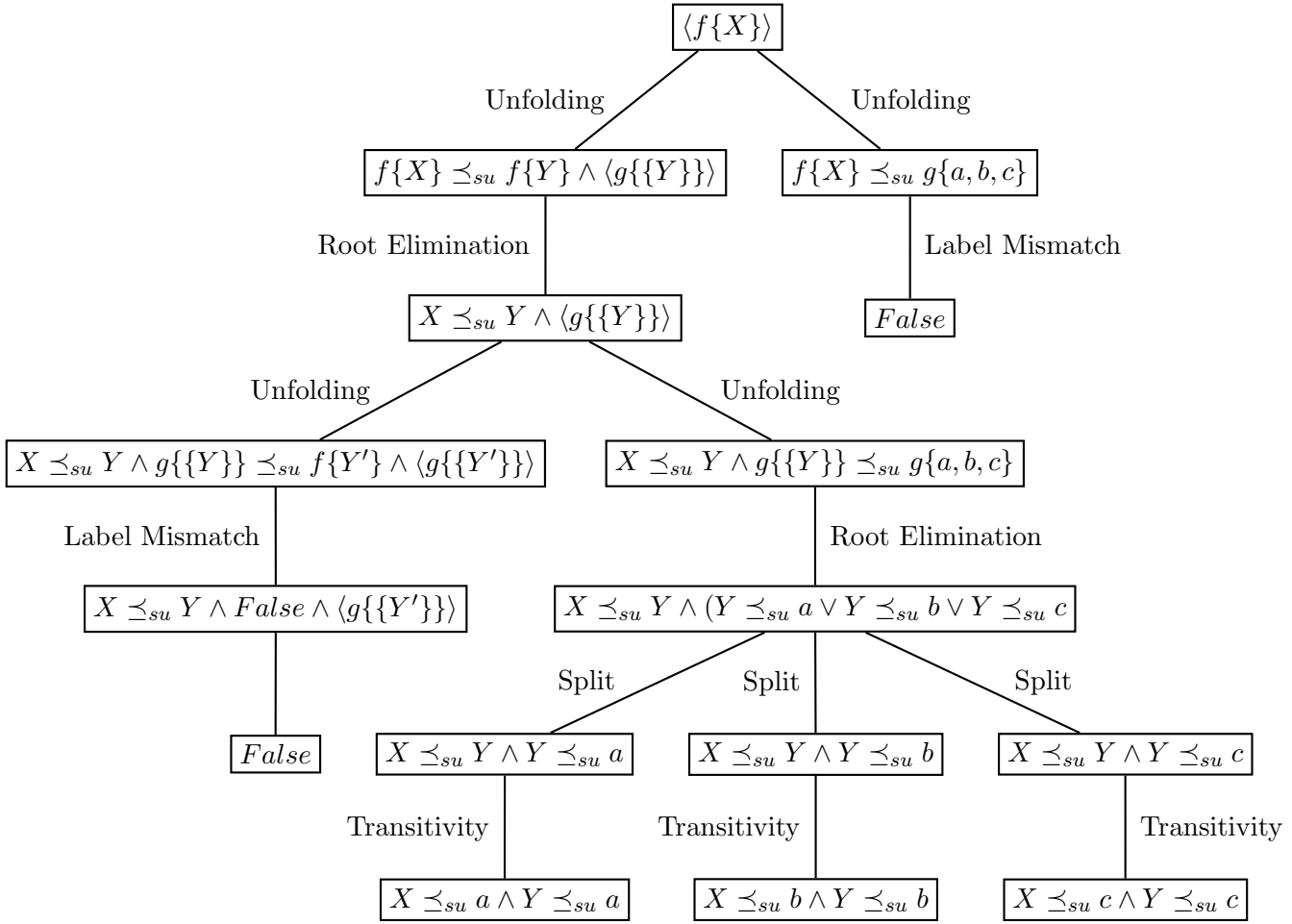
In the following examples, the confluence (or Church-Rosser property) of the calculus for constraint selection order is assumed without proof. That is, for all paths having the same result but different constraint selection order, the examples shows only one representative path. Otherwise, the examples would be too large for visualization.

Example 15 (Calculus Tree) *Assume the following Xcerpt program against which the constraint $\langle g\{\{Y\}\}\rangle$ is evaluated.*

Program 1

$f\{Y\} \leftarrow g\{\{Y\}\}$
$g\{a, b, c\}$

A calculus tree for $\langle f\{X\}\rangle$ in the above program would be the following.


$\langle f\{X\}\rangle$
Unfolding
Unfolding
$f\{X\} \preceq_{su} f\{Y\} \wedge \langle g\{\{Y\}\}\rangle$
$f\{X\} \preceq_{su} g\{a,b,c\}$
Root Elimination
Label Mismatch
$X \preceq_{su} Y \wedge \langle g\{\{Y\}\}\rangle$
$False$
Unfolding
Unfolding
$X \preceq_{su} Y \wedge g\{\{Y\}\} \preceq_{su} f\{Y'\} \wedge \langle g\{\{Y'\}\}\rangle$
$X \preceq_{su} Y \wedge g\{\{Y\}\} \preceq_{su} g\{a,b,c\}$
Label Mismatch
Root Elimination
$X \preceq_{su} Y \wedge False \wedge \langle g\{\{Y'\}\}\rangle$
$X \preceq_{su} Y \wedge (Y \preceq_{su} a \vee Y \preceq_{su} b \vee Y \preceq_{su} c$
Split
Split
Split
$False$
$X \preceq_{su} Y \wedge Y \preceq_{su} a$
$X \preceq_{su} Y \wedge Y \preceq_{su} b$
$X \preceq_{su} Y \wedge Y \preceq_{su} c$
Transitivity
Transitivity
Transitivity
$X \preceq_{su} a \wedge Y \preceq_{su} a$
$X \preceq_{su} b \wedge Y \preceq_{su} b$
$X \preceq_{su} c \wedge Y \preceq_{su} c$

With the notion of a calculus tree and with the assumption of confluence of the calculus, the needed *solve* and *solveall* functions for the "One at Once" calculus are defined as follows.

The *solve* function performs a depth first search and returns the first constraint that is not equivalent to $False$ and cannot be simplified further.

function *solve*(Constraint CS) : Constraint
    Boolean $noRuleApplies = True$
    foreach Rule R applicable to CS **do**
        $noRuleApplies = False$
        Set(Constraint) CSS =apply R to CS
        foreach CS' **in** CSS **do**
            Constraint $CS'' = solve(CS')$
            if not $CS'' = False$ **then return** CS''
    if $noRuleApplies$
        then return CS
        else return $False$

If no rule is applicable to the input constraint store CS, then *solve* returns CS. If there is a rule that applies to CS, then the rule is applied to CS. If every constraint store resulting from the application of the rule does not lead to a solution, then *solve* returns $False$.

The *solveall* function performs a depth first search for the input constraint and then reassembles the different proof paths into one constraint in disjunctive normal form.

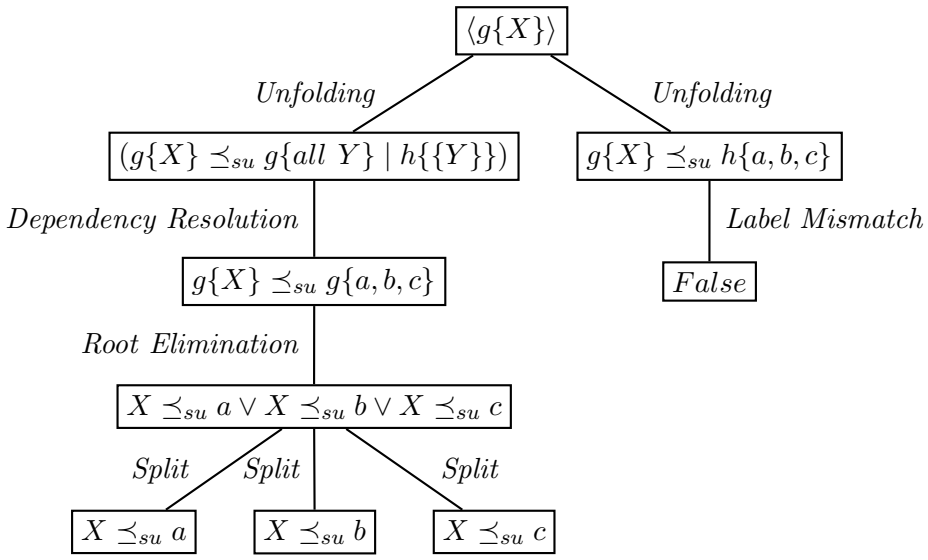
function *solveall*(Constraint CS) : Constraint
    Set(Constraint) $CSS = \emptyset$
    Constraint $CS' = False$
    foreach Rule R applicable to CS **do**
        Set(Constraint) CSS' = apply R to CS
        foreach CS'' **in** CSS' **do**
            Constraint $CS'' = solveall(CS'')$
            $CSS = CSS \cup CS''$
    foreach Constraint CS'' in CSS **do**
        $CS' = CS' \vee CS''$
    return CS'

Example 16 (Program Evaluation) *Assume the following program.*

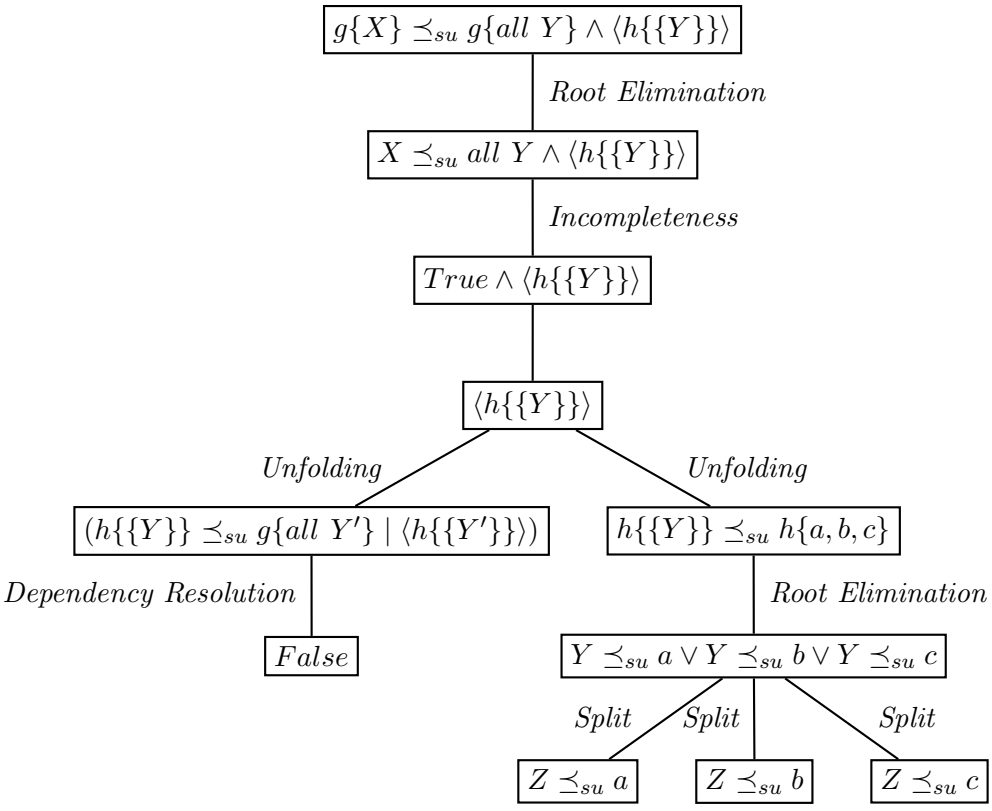
Program 2

$f\{X\} \leftarrow g\{\{X\}\}$
$g\{all\ Y\} \leftarrow h\{\{Y\}\}$
$h\{a, b, c\}$

And let $f\{Y\} \leftarrow g\{\{Y\}\}$ be a goal of the program. Inspecting the program evaluation algorithm, the expression $subst(solve(\langle g\{\{Y\}\}\rangle))$ must be evaluated to calculate the goal result. The full calculus tree for the constraint $\langle g\{\{Y\}\}\rangle$ is as follows.



Where the application of the dependency resolution rule lead to the following auxiliary calculation. Obviously, it holds that $g\{X\} \preceq_{su} g\{all\ Y\}$*. Hence, the calculus tree is build for the constraint* $g\{X\} \preceq_{su} g\{all\ Y\} \wedge \langle h\{\{Y\}\}\rangle$ *which is the following:*



The second application of the dependency resolution rule failed because $h\{\{Y\}\} \preceq_{su} g\{all\ Y'\}$ *reduces to* $False$ *due to label mismatch. The returned constraint store from solveall is now*

$Y \preceq_{su} a \vee Y \preceq_{su} b \vee Y \preceq_{su} c$. *The substitution for this constraint store is then*

$$\Sigma = subst(Y \preceq_{su} a \vee Y \preceq_{su} b \vee Y \preceq_{su} c) = \{\{a/Y\}, \{b/Y\}, \{c/Y\}\}.$$

Applied to the head of the grouping rule, the result is $g\{all\ Y\}\Sigma = \{g\{a, b, c\}\}$. *The result of the overall calculation could be hence with the substitution* $\Psi = subst(Y \preceq_{su} a) = \{\{a/Y\}\}$ *the term* $f\{a\}$, *since* $f\{Y\}\Psi = \{f\{a\}\}$.

Note that the nondeterminism involved in the calculus would allow for a short representation of program evaluation, since the right path could be always selected where only one path is followed.

5.4 "All at Once"

"All at Once", in contrast to "One at Once" follows all proof paths at a time. It preserves the disjunctions produced by the calculus rules for a compact representation of the multiple proof paths. At the same time, the "All at Once" approach allows for an easy integration of different search strategies than depth-first-search such as a complete cost based A*-Search strategy. It is hence the implementation of choice for the Xcerpt language processor.

5.4.1 "All at Once" Calculus Rules

The "All at Once" approach merges all three chaining rules in one rule. If a folded query constraint is to be resolved, the contained query term is tested against every term in the program as well as against every rule head (grouping or non-grouping). The chaining is hence reduced to the single following rule.

	$\langle t^q \rangle$	
$\bigvee_{t \in T}$	$t^q \preceq_S t$	$\vee$
$\bigvee_{(t^c \leftarrow Q) \in R_{ng}}$	$t^q \preceq_S t^c \wedge \langle Q \rangle$	$\vee$
$\bigvee_{(t^c \leftarrow Q) \in R_n}$	$(t^q \preceq_S t^c \mid \langle Q \rangle)$	

A dependency constraint is only introduced in cases where grouping rules are involved. In the other cases, the calculus is able to "chain through the rule".

5.4.2 "All at Once" Evaluation Algorithms

In contrast to "One at Once", the "All at Once" calculus does not really need the notion of a proof tree to understand its constraint solving approach. The pseudocode for *solveall* in an "All at Once" approach simply applies the rules of the calculus until no more rule is applicable and returns the resulting constraint store.

The pseudocode of *solve* is more difficult here, because the application of the constraint solving should stop once a conjunction of constraints is found that alone represents a valid solution to the constraint problem. This conjuction of constraints is characterizable by the following two properties: First, it is not equal to $False$ and second, there are no applicable rules of the calculus to the conjunction of constraints.

The pseudocode of *solve* and *solveall* is as follows.

function *solveall*(Constraint CS) : Constraint
    while (a Rule is applicable to CS) **do**
        Rule R = select a Rule applicable to CS
        CS = apply R to CS
    return CS

solveall loops as long as there is a calculus rule that is applicable to the constraint store and returns the completely solved constraint store. *solve* can be specified similarly, but with a more complex break condition.

function *solve*(Constraint CS) : Constraint
    while (a Rule is applicable to CS) **do**
        if exists Constraint cs **in** $disjuncts(CS)$:
                $cs \neq False$ **and** no Rule applies to cs
            then return cs
        Rule R = select a Rule applicable to CS
        CS = apply R to CS
    return $False$

solve loops as long as there is a rule applicable to the whole constraint store CS. If there is a conjunction of the DNF of CS that is a solution of the constraint problem, then this conjunction is returned. If no such conjunction was ever encountered in the loop and no rule applies to the constraint store, then there is no solution to the initial constraint problem and $False$ is returned.

A proof search strategy can be implemented into *solve* and *solveall* by the selection of the calculus rule to use and the part c of the complete constraint store CS to which the rule applies. By selecting c within the same conjunction of constraints every time, a depth first search is performed with a similar space overhead as backtracking strategies. By changing the conjunctions within the constraint store in which the constraint c is selected, a breadth first search is performed. Other search algorithms can build upon this freedom of choice to perform the search.

Example 17 (Calculus application) *The application of the calculus is again demonstrated with the same program as for the "One at Once" calculus.*

Program 1
$f\{Y\} \leftarrow g\{\{Y\}\}$
$g\{a, b, c\}$

Unlike the "One at Once" calculus, this calculus type has no need for a disjunction elimination. The integration of the simulation unification rules is hence simper with this type of calculus.

Here again, the folded query $\langle f\{X\}\rangle$ is the beginning of the calculus.

$$\frac{\langle f\{X\}\rangle}{(f\{X\} \preceq_{su} f\{Y\} \wedge \langle g\{\{Y\}\}\rangle) \vee (f\{X\} \preceq_{su} g\{a,b,c\}} \textit{ Unfolding}$$

$$\frac{(f\{\{X\}\} \preceq_{su} f\{Y\} \wedge \langle g\{\{Y\}\}\rangle) \vee False}{} \textit{ Label Mismatch}$$

$$\frac{f\{\{X\}\} \preceq_{su} f\{Y\} \wedge \langle g\{\{Y\}\}\rangle}{X \preceq_{su} Y \wedge \langle g\{\{Y\}\}\rangle} \textit{ Root Elimination}$$

$$\frac{X \preceq_{su} Y \wedge ((g\{\{Y\}\} \preceq_{su} f\{Y'\} \wedge \langle g\{\{Y'\}\}\rangle) \vee (g\{\{Y\}\} \preceq_{su} g\{a,b,c\})}{} \textit{ Unfolding}$$

$$\frac{X \preceq_{su} Y \wedge ((False \wedge \langle g\{\{Y'\}\}\rangle) \vee (g\{\{Y\}\} \preceq_{su} g\{a,b,c\})}{} \textit{ Label Mismatch}$$

$$\frac{X \preceq_{su} Y \wedge g\{\{Y\}\} \preceq_{su} g\{a,b,c\}}{X \preceq_{su} Y \wedge (Y \preceq_{su} a \vee Y \preceq_{su} b \vee Y \preceq_{su} c)} \textit{ Root Elimination}$$

$$\frac{(X \preceq_{su} Y \wedge Y \preceq_{su} a) \vee (X \preceq_{su} Y \wedge Y \preceq_{su} b) \vee (X \preceq_{su} Y \wedge Y \preceq_{su} c)}{(X \preceq_{su} a \wedge Y \preceq_{su} a) \vee (X \preceq_{su} b \wedge Y \preceq_{su} b) \vee (X \preceq_{su} c \wedge Y \preceq_{su} c)} \textit{ Transitivity(3 times)}$$

5.4.3 "All at Once" - Goal Driven Forward Chaining

The goal driven forward chaining variant of "All at Once" is similar to the magic sets methods known from deductive databases and logic programming. The variation of the chaining rule handles both the non-grouping and grouping rules with dependency constraints. This implies, as is visible in the examples, a goal driven forward chaining character. The advantage of this approach is that the constraint store on which the rules work contain far less simulation constraints with variables on both sides and the constraint store contains less constraints.

$$\frac{\langle t^q\rangle}{\begin{array}{lll} \bigvee_{t \in T} & t^q \preceq_S t & \vee \\ \bigvee_{(t^c \leftarrow Q) \in R} & (t^q \preceq_S t^c \mid \langle Q\rangle) & \end{array}}$$

Nevertheless, this variant also has disadvantages. The simulation unification of the query term contained in the query constraint with a given rule head is done twice. The first unification is the pre-test of the dependency constraint resolution rule and the second unification is performed within the conclusion of the dependency resolution rule, once the instances of the rule head were created (see Page 37). In fact, this double evaluation demonstrates the goal driven and forward chaining character of this calculus.

An example of the calculus application cannot be given here, since this calculus builds completely upon the dependency resolution rule. Examples for program evaluations using this dependency resolution rule is given for each calculus in the next section.

Example 18 (Program Evaluation) *For the program evaluation example of the two variants of "All at Once", assume the following two programs.*

Program 2	**Program 3**
$f\{X\} \leftarrow g\{\{X\}\}$	$f\{X\} \leftarrow g\{X\}$
$g\{all\ Y\} \leftarrow h\{\{Y\}\}$	$g\{Y\} \leftarrow h\{\{Y\}\}$
$h\{a, b, c\}$	$h\{a, b, c\}$

The evaluation of Program 2 with "All at Once" is as follows. The program evaluation algorithm starts with calling solve($\langle g\{\{Y\}\}\rangle$). This leads to the following calculation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle g\{\{X\}\}\rangle}{(g\{\{X\}\} \preceq_{su} g\{all\ Y\} \mid \langle h\{\{Y\}\}\rangle) \vee g\{\{X\}\} \preceq_{su} h\{a, b, c\}}\ \textit{Unfolding}}{(g\{\{X\}\} \preceq_{su} g\{all\ Y\} \mid \langle h\{\{Y\}\}\rangle) \vee False}\ \textit{Label Mismatch}}{(g\{\{X\}\} \preceq_{su} g\{all\ Y\} \mid \langle h\{\{Y\}\}\rangle)}}{g\{\{X\}\} \preceq_{su} g\{a, b, c\}}\ \textit{Dependency Resolution}}{X \preceq_{su} a \vee X \preceq_{su} b \vee X \preceq_{su} c}\ \textit{Root Elimination}$$

The substitution set resulting from the initial program evaluation call is $\Psi = \{\{a/X\}, \{b/X\}, \{c/X\}\}$, *and the result of the program evaluation is hence one of the three terms*

$$f\{X\}\Psi = \{f\{a\}, f\{b\}, f\{c\}\}.$$

The recursive call was triggered since $g\{\{X\}\} \preceq_{su} g\{all\ Y\}$ *reduces to* $True$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{True \wedge \langle h\{\{Y\}\}\rangle}{\langle h\{\{Y\}\}\rangle}}{(h\{\{Y\}\} \preceq_{su} g\{all\ Y'\} \mid \langle h\{\{Y'\}\}\rangle) \vee h\{\{Y\}\} \preceq_{su} h\{a, b, c\}}\ \textit{Unfolding}}{False \vee h\{\{Y\}\} \preceq_{su} h\{a, b, c\}}\ \textit{Dependency Resolution}}{h\{\{Y\}\} \preceq_{su} h\{a, b, c\}}}{Y \preceq_{su} a \vee Y \preceq_{su} b \vee Y \preceq_{su} c}\ \textit{Root Elimination}$$

The substitution set resulting from the call is $\Sigma = \{\{a/Y\}, \{b/Y\}, \{c/Y\}\}$, *and the initial calculation continued.*

The evaluation of Program 2 does not differ between the two variants of the "All at Once" calculus, while the evaluation of Program 3 does. For both calculi, the program evaluation

calls solve with the constraint $\langle g\{\{Y\}\}\rangle$. The application of the pure "All at Once" calculus leads to the following computation.

$$\langle g\{\{X\}\}\rangle$$

$$\frac{}{(g\{\{X\}\} \preceq_{su} g\{Y\} \wedge \langle h\{\{Y\}\}\rangle) \vee g\{\{X\}\} \preceq_{su} h\{a,b,c\}} \textit{ Unfolding}$$

$$\frac{}{(g\{\{X\}\} \preceq_{su} g\{Y\} \wedge \langle h\{\{Y\}\}\rangle) \vee False} \textit{ Label Mismatch}$$

$$\frac{}{g\{\{X\}\} \preceq_{su} g\{Y\} \wedge \langle h\{\{Y\}\}\rangle}$$

$$\frac{}{X \preceq_{su} Y \wedge \langle h\{\{Y\}\}\rangle} \textit{ Root Elimination}$$

$$\frac{}{X \preceq_{su} Y \wedge ((h\{\{Y\}\} \preceq_{su} g\{Y'\} \wedge \langle h\{\{Y'\}\}\rangle) \vee h\{\{Y\}\} \preceq_{su} h\{a,b,c\})} \textit{ Unfolding}$$

$$\frac{}{X \preceq_{su} Y \wedge ((False \wedge \langle h\{\{Y'\}\}\rangle) \vee h\{\{Y\}\} \preceq_{su} h\{a,b,c\})} \textit{ Label Mismatch}$$

$$\frac{}{X \preceq_{su} Y \wedge h\{\{Y\}\} \preceq_{su} h\{a,b,c\}}$$

$$\frac{}{X \preceq_{su} Y \wedge (Y \preceq_{su} a \vee Y \preceq_{su} b \vee Y \preceq_{su} c)} \textit{ Root Elimination}$$

$$\frac{}{(X \preceq_{su} Y \wedge Y \preceq_{su} a) \vee (X \preceq_{su} Y \wedge Y \preceq_{su} b) \vee (X \preceq_{su} Y \wedge Y \preceq_{su} c)} \textit{ Transitivity}$$

The returned constraint store is the one conjunction to which no more calculus rules are applicable: $X \preceq_{su} a \wedge Y \preceq_{su} a$. The program result is hence

$$f\{X\}\{\{a/X, a/Y\}\} = \{f\{a\}\}.$$

In contrast to this, the use of goal driven forward chaining calculus variant leads to the following computation.

$$\langle g\{\{X\}\}\rangle$$

$$\frac{}{(g\{\{X\}\} \preceq_{su} g\{Y\} \mid \langle h\{\{Y\}\}\rangle \vee g\{\{X\}\} \preceq_{su} h\{a,b,c\}} \textit{ Unfolding}$$

$$\frac{}{(g\{\{X\}\} \preceq_{su} g\{Y\} \mid \langle h\{\{Y\}\}\rangle \vee False} \textit{ Label Mismatch}$$

$$\frac{}{(g\{\{X\}\} \preceq_{su} g\{Y\} \mid \langle h\{\{Y\}\}\rangle}$$

$$\frac{}{g\{\{X\}\} \preceq_{su} g\{a\} \vee g\{\{X\}\} \preceq_{su} g\{b\} \vee g\{\{X\}\} \preceq_{su} g\{c\}} \textit{ Dependency Resolution}$$

$$\frac{}{X \preceq_{su} a \vee g\{\{X\}\} \preceq_{su} g\{b\} \vee g\{\{X\}\} \preceq_{su} g\{c\}} \textit{ Root Elimination}$$

The initial call to solve ends here and the constraint $X \preceq_{su} a$ is returned. The recursive call to solveall was performed within the dependency Resolution since $g\{\{X\}\} \preceq_{su} g\{Y\}$ reduces

to $X \preceq_{su} Y$ *by Root Elimination.*

$$\frac{X \preceq_{su} Y \wedge \langle h\{\{Y\}\}\rangle}{\begin{array}{l} X \preceq_{su} Y \wedge ((h\{\{Y\}\} \preceq_{su} g\{Y'\} \mid \langle h\{\{Y'\}\}\rangle \\ \vee h\{\{Y\}\} \preceq_{su} h\{a,b,c\}) \end{array}} \textit{ Unfolding}$$

$$\frac{X \preceq_{su} Y \wedge (False \vee h\{\{Y\}\} \preceq_{su} h\{a,b,c\})}{X \preceq_{su} Y \wedge h\{\{Y\}\} \preceq_{su} h\{a,b,c\}} \textit{ Dependency Resolution}$$

$$\frac{X \preceq_{su} Y \wedge (Y \preceq_{su} a \vee Y \preceq_{su} b \vee Y \preceq_{su} c)}{(X \preceq_{su} Y \wedge Y \preceq_{su} a) \vee (X \preceq_{su} Y \wedge Y \preceq_{su} b) \vee (X \preceq_{su} Y \wedge Y \preceq_{su} c)} \textit{ Root Elimination}$$

$$\frac{(X \preceq_{su} a \wedge Y \preceq_{su} a) \vee (X \preceq_{su} b \wedge Y \preceq_{su} b) \vee (X \preceq_{su} c \wedge Y \preceq_{su} c)}{} \textit{ Transitivity (3 times)}$$

The result from the program evaluation is the same as above for the pure "All at Once" calculus.

$$f\{X\}subst(X \preceq_{su} a) = f\{X\}\{\{a/X\}\} = \{f\{a\}\}$$

The important difference between the pure and the goal driven forward chaining variant of "All at Once" is obviously that the calculated substitution sets are "cleaner" in the sense that there are no additional variable bindings present in them besides the needed ones.

5.5 Comparison of the Calculi

This section compares the three calculi regarding chaining behavior, specification clarity and fitness.

Intuitive Chaining behavior The chaining behavior of calculi for Xcpert may differ even if the results are the same. Some calculi may involve a certain overhead or show a counterintuitive processing of the program while still being correct.

Investigating the goal driven forward chaining variation of the "All at Once" calculi, it can be stated that it has the disadvantage of calculating some constraint simplifications twice. This happens both in the case of chaining with non-grouping rules and with grouping rules. If the query simulates into the rule head, this simulation gets calculated twice. Once in the resolution of the dependency constraint, and then again in the final simulation of the query against the generated data terms. This double unification involves obviously some time overhead in the calculation. Fortunately, this effect can be eliminated by using a global rule application mechanism or a pointer mechanism in the implementation of the dependency constraint resolution.

The "One at Once" calculus just like the "All at Once" calculus has no further impact on chaining behavior, since the behavior is mainly controlled by the search strategy used.

Clear Specification The specification of the chaining calculi may use different mathematical concepts. A specification may be hence easier to use or may emphasize a certain aspect of chaining more than other specifications. These points are discussed in the following.

The "One at Once" calculus with the notion of the calculus tree shows the impact of the grouping constructs on chaining. The the calculus trees make the chaining in Xcerpt easier to grasp because of the emphasis on the accumulation of proofs. The "All at Once" calculi on the other hand make no use the calculus trees but content with the notion of constraint store operation. It could be said that the second approach conceal the impact of the grouping constructs on chaining. However, different search strategies can be easily integrated into the "All at Once" calculi. Hence, the "All at Once" offers a clear formalization of chaining that can be easily used for the analysis of search strategies.

Fitness Different calculi have different approaches and concepts for chaining. The language Xcerpt has requirements and properties that may lead to the preference of a calculus formalization over the others. This is investigated in the following.

The "One at Once" formalization leans strongly on the existing mathematical specifications of backward chaining for Prolog. Nevertheless, it has become evident that this SLD-like style of program evaluation does not fit well with the simulation unification rules because of the required disjunction handling.

An argument for an SLD-like evaluation could be that the rules for simulation unification could be rewritten instead of using the "All at Once" formalizations. Considering this possibility, it becomes soon evident that this would simply combine the "Disjunctive Split" rule with every disjunction generating calculus rule. The difference is hence small between such an approach and the presented "One at Once" calculus. The "One at Once" calculus already did the closest approach to SLD-resolution feasible. The "All at Once" calculi on the other hand integrate well the simulation unification rules.

Nevertheless, the "One at Once" formalization is handy for the human to calculate Xcerpt's rule chaining by hand just because of the feasible selection in the disjunctive split. It allows to concentrate on one single proof and use intuition to select the right rules of the calculus to apply and the rules of the Xcerpt program to use for chaining. At the same time, failing calculation paths can be simply left away, where the "All at Once" calculi require to consider all possible program rules for chaining. Furthermore, the notion of the proof tree introduced for the "One at Once" calculus clarifies the processing of an Xcerpt program in the presence of grouping construct rules.

Both formalizations, "All at Once" and "One at Once" have hence their fields of application.

6 Conclusion

In this thesis, terms for semistructured data were introduced together with Xcerpt's variations of them, the construct and query terms. Second, a notion of simulation unification was defined first on ground query terms and then extended to non-ground terms. In the following, a partial calculus for simulation unification was defined which is incomplete for grouping construct terms. This incompleteness was then lifted by the introduction of special chaining

mechanism building upon the dependency constraint and its resolution. Afterwards, this work presented three possible formalizations for chaining by calculi. The properties of each of these three variants were investigated in regards to their fitness for Xcerpt. The outcome of this investigation was that the first calculus ("One at Once") is suitable for computation of chaining by hand and for understanding chaining processing, while the second and third calculi ("All at Once" and its goal driven forward chaining variant) fits well for the purpose of an automated language processing.

What is left apart in this thesis is the formal proof of correctness and completeness of the presented calculi, since the aim of this thesis was to consider possible formalizations of chaining and to formalize the evaluation process as a whole. Future papers may present a complete formalization of the semantics of Xcerpt and prove the correctness and completeness of the calculi.

This thesis has shown some interesting further fields of research. In the field of Xcerpt program evaluation, parallelization and distributed evaluation could be the investigated. The revealing of the evaluation ordering interdependencies in an Xcerpt program could lead to interesting performance advantages in program evaluation, since for example the waiting time that occurs in the requesting of web resources could be then filled with the evaluation of a parallelizable part of the Xcerpt program.

Another field of research could be the investigation of search strategies for the evaluation calculi. Here, a complete search strategy like A*-search or similar nontrivial search strategies could help to improve termination and result finding of Xcerpt program evaluation in various scenarios.

To summarize, the evaluation of Xcerpt programs have shown to have some salient nonstandard properties that are associated with the nature of the web which renders difficult not to depart from standard evaluation techniques; The semistructured nature of data on the web leads to the need of partial query specification which again leads to the notion of simulation unification. This unification, relying on constraint solving techniques, made the application of SLD-like calculi rather difficult. Furthermore, the grouping constructs *all* and *some* led to a preference of non-SLD calculi for use in an Xcerpt language processor. It became evident that for the chaining in Xcerpt, it is better to depart from SLD calculi rather than staying close to them in the formalization.

References

[Av82] Kristoff R. Apt and Maarten H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[BBSW03] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Demonstrations Track, Proceedings of 29th Intl. Conference on Very Large Databases, Berlin, Germany (9th–12th September 2003)*, 2003.

[BS02a] François Bry and Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Sardinia, Italy (14th June 2002)*, 2002.

[BS02b] François Bry and Sebastian Schaffert. Pattern Queries for XML and Semistructured Data. Forschungsbericht/research report PMS-FB-2002-5, Institute of Computer Science, LMU, Munich, 2002.

[BS02c] François Bry and Sebastian Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Web, Web-Services, and Database Systems, Proceedings of 2nd Annual International Workshop "Web and Databases", Erfurt, Germany (9th–10th October 2002)*, volume 2593 of *LNCS*. German Informatics Society (GI), 2002.

[BS02d] François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. Forschungsbericht/research report PMS-FB-2002-2, Institute of Computer Science, LMU, Munich, 2002.

[BS02e] François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of International Conference on Logic Programming, Copenhagen, Denmark (29th July–1st August 2002)*, volume 2401 of *LNCS*, 2002.

[BS03] François Bry and Sebastian Schaffert. An Entailment Relation for Reasoning on the Web. In *Proceedings of Rules and Rule Markup Languages for the Semantic Web, Sanibel Island (Florida), USA (20th October 2003)*, LNCS, 2003.

[KK71] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3–4):227–260, 1971.

[Lov68] Donald W. Loveland. Mechanical theorem proving by model elimination. *Journal of the ACM*, 15(2):236–251, 1968.

[Lov69] Donald W. Loveland. A simplified format for the model elimination procedure. *Journal of the ACM*, 16(3):349–363, 1969.