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

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München
LMU

An Efficient Evaluation of Xcerpt

An Algebra and Optimization Techniques for Simulation Unification

Andreas Schroeder

Diplomarbeit

Beginn der Arbeit:	22.06.2005
Abgabe der Arbeit:	21.12.2005
Betreuer:	Prof. Dr. François Bry
	Dr. Sebastian Schaffert
	Tim Furche

Erklärung

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

München, den 21.12.2005 Andreas Schroeder

Abstract

Xcerpt is a rule based query language for semistructured data offering salient new aspects to rule based querying. Xcerpt uses a pattern based approach to data querying. In this thesis, a higher level algebra for Xcerpt's unification method, simulation unification, is presented. Furthermore, the matrix method which is an implementation method for simulation unification is presented and possible optimizations are investigated. Both the matrix method and the algebra aims at improving the performance of Xcerpt query evaluation. Throughout this work, the focus is be set on optional and negated query sub-patterns. The interplay of these salient Xcerpt features with the basic query concepts is the main thread throughout this work. As this thesis reveals, both optional and negated query sub-patterns are beneficial concepts for both the programmer and the evaluation.

Zusammenfassung

Xcerpt ist eine regelbasierte Anfragesprache für semistrukturierte Daten, die besondere neue Aspekte beinhaltet. Xcerpt verwendet einen musterbasierten Ansatz für das Anfragen von Daten. In dieser Diplomarbeit wird eine logische Algebra für Xcerpt's Unifikationsmethode, der Simulationsunifikation, behandelt. Daneben wird die Matrixmethode erläutert, die eine Implementierungstechnik für die Simulationsunifikation ist. Darüber hinaus werden Optimierungen dieser Matrixmethode untersucht. Sowohl die Matrixmethode als auch die Algebra sollen die Performanz der Anfrageauswertung in Xcerpt verbessern. Der Schwerpunkt dieser Arbeit wird dabei auf die Behandlung von optionalen und negierten Teilmustern liegen. Das Zusammenspiel dieser besonderen Konstrukte mit anderen grundlegenden Konzepten der Anfragebearbeitung von Xcerpt ist der zentrale Leitfaden dieser Arbeit. Wie hier gezeigt wird, sind beide Arten von Teilmustern, die optionale wie auch die negierten, sowohl für den Programmierer wie auch für die Anfragebearbeitung von Vorteil.

Acknowledgments

There were astonishingly many persons who supported me during this work. I do not know what this work would look like if they did not helped me the way they did. Of these persons, I thank before all:

- *François Bry*, for asking the right questions, pointing me to the right directions, and providing a fruitful atmosphere throughout all discussions.
- *Tim Furche*, with whom I had many discussions about the algebra and the algorithmic improvements. I must before all thank him for having a sympathetic ear.
- *Sebastian Schaffert*, for creating an interesting programming language to investigate, and supporting me in the beginning of this work.

Inhaltsverzeichnis

1 Introduction

With the rise of XML technologies and XML based technologies the need for querying and transforming XML data emerged quickly. The W3C has recommended two languages, namely XSLT [Cla99] and XQuery [BCF+05], for these tasks. XSLT is a functional style transformation language using templates dedicated to XML transformation tasks, while XQuery is a functional style query language to query XML Data. Both languages use XPath [CD99] for structure navigation.

Alongside with the (still unfinished) development of XQuery, a rule based query and transformation language, Xcerpt [Sch04, BBSW03, BS03, BSS04, BFB+04, BFB+05, BM05], is under development in Munich and is further refined to become a base technology within the REWERSE network of excellence.[1] While Xcerpt is a fullfledged query and transformation language, its goals differ notably from those of XQuery. Xcerpt is aimed to be a more declarative language than XQuery and is designed to fit tasks of automated reasoning on XML. A further difference between Xcerpt and XQuery is that Xcerpt is not only able to deal with XML, but also with data that is compliant to a broader model of semistructured data, including e.g. RDF [MB04, MBG04, MKC04, MH04] and TopicMaps [BBN99] data. Xcerpt hence not only aims to be an XML technology, but is also a more general approach to data access.

This has obviously positive and negative aspects: while being more general than a pure XML query language, Xcerpt in its current, prototypical version lacks native support for XML specifics such as processing instructions, comments and entities. Xcerpt can support all these specifics by certain transformation conventions. At the same time, as languages get more and more general, implementation and optimization issues become more and more challenging. However, this also means that Xcerpt brings up interesting questions for research. As this work shows, the design of Xcerpt and its special constructs led us to investigate manyfold aspects of query evaluation and optimization with surprising results.

1.1 Motivation

Some readers may understand the words “declarative” and “reasoning capable” as “performing bad” and “performing even worse”. This is true to some extent, but not entirely. First, it must be clear that Xcerpt is a programming language designed for easy data retrieval, transformation and reasoning. It is obvious that programs for other purposes than reasoning on semistructured data or semistructured data transformation and access should be written in other languages. A shader program for a video game for example should definitely not be programmed in Xcerpt, but rather in C and assembly languages. The performance of a programming language must be considered in its intended and realistic field of application. For the tasks of reasoning, data transformation and data access, Xcerpt is performing quite well and has the potential to perform even better.

Xcerpt offers very high level constructs such as term optionality and term negation. These constructs, especially the optional construct, ease program development since they allow to write more declarative programs, that is, programs that reflect more the intend of the programmer in their structure. The salient aspect of such constructs from the perspective

[1] See www.rewerse.net

of optimization is that they may allow a more efficient program execution (and the optional construct indeed has this effect; queries using optional constructs may be evaluated more efficiently than equivalent queries without optional constructs). This is also an interesting result of optimization research: constructs that are handy for the programmer often also lead to efficient evaluation. Since the program structure reflects the user intent in a better way, the choice of an appropriate algorithm may be easier for the optimizer. However, high level constructs do not always have a positive effect on the overall evaluation performance. In cases where the high level constructs do not lead to faster programs, it is still the aim of this work to find algorithms and data structures allowing a reasonable query evaluation performance.

Improving things is a basic need of humanity. Science and research demonstrate this clearly: the aim of research is to find methods to improve techniques. Technology itself again is a means to improve and ease the life of all. Program optimization carries this over to computer science: we strive to find better ways of doing things that we need the computer to do, that is, we look for better algorithms and data structures for programming tasks. But as often experienced with technology, improving things often enough fails or leads to unexpected side effects. In computer science, failing to improve means that certain optimizations may have only marginal or even negative overall effects on performance. Hence, careful steps steps must be made when optimizing programs; the results must be verified by either investigating algorithm complexity or testing and measuring the execution speed of the algorithms on adequate data and queries.

1.2 Optimization Techniques

The two aspects of query evaluation optimized in this work are processing time and memory footprint of query processing. A good algorithm will reduce both, but often enough, memory footprint and low processing time are conflicting objectives in optimization. Sometimes however, low memory footprint coincides with execution speed. This is the case when the memory footprint is mainly determined by the size of the processed intermediate data. In such cases, reducing the amount of processed intermediate data reduces both memory footprint and execution time.

Basically, two kinds of optimizations are considered in this thesis. The first kind is to investigate improvements of algorithms and data structures. Again, there are different ways to do so. We could try to find better algorithms and data structures for the general case of query evaluation. Since simulation unification itself is NP-hard[2], these optimizations could be very hard to find, lead to only marginal improvement or win the Alan Turing Award for proving the famous equation "P = NP", which is definitely not the aim of this work. The second way of algorithmic query optimization is more promising in the context of a diploma thesis: it may be possible to identify cases allowing an efficient (hopefully polynomial) algorithmic treatment.

Besides searching for an improved algorithmic handling, heuristics can be used to improve performance. Here again, two kinds of heuristics are investigated for query evaluation. The first kind of heuristics are memoization techniques: a data structure (mostly some kind of table) stores intermediate results as they are computed for the first time, anticipating their future reuse. This optimization technique must be used with hindsight, since extensive mem-

[2]Simulation unification allows to perform graph queries against graph data, which is known to be NP-hard.

ory usage leads to bad program performance and even bad overall system performance due to trashing effects.[3] The matrix method for simulation unification developed by Schaffert is an example for memoization techniques leading to a tremendous reduction of query evaluation time. Memoization techniques are sensible if the memoization lookup is cheap, if the memoized data is used often and if amount of avoided computation is high.

The second kind of heuristics is pruning techniques: if a part of the query will fail on a given data term without completely evaluating it, its evaluation can be omitted. These techniques are recommended only when the pruning test is cheap, the probability of pruning is high and the avoided computation amount is high. If the pruning test is too expensive and the pruning probability is too low, using pruning heuristics may lead to an overall slow down of the query evaluation.

1.3 Algebras for Query Languages

Algebras are widespread query optimization tools. The definition of the relational algebra by Codd in 1970 [Cod70] has been a revolutionary step in relational database research. The relational algebra has a rich set of equivalence rules which allow query optimization through rewritings. For SQL queries for example, the direct translation of a query into relational algebra expressions leads to the so-called canonical query plan. The fastest query plans that can be generated through query rewritings starting from the canonical query plan often enough outclasses the initial plan by order of magnitudes in both execution speed and memory savings.

The relational algebra furthermore divides the query evaluation process into small composable steps, which can be optimized and implemented independently. The relational database management system can then choose between different implementations for the single operators. The selection operator for example may be realized as a straightforward full table scan or as a sophisticated index search. For the join operator alone, there are a multitude of possible realizations starting with the canonical nested loop join to such sophisticated methods such as the hybrid hash join.

The relational algebra is a core concept of query optimization for relational databases, since it offers two important aspects. First, it offers a frame for query analysis and query rewriting, which is often called logical optimization. The algebraic expression stemming from the logical optimization is also called logical query plan. Second, the relational algebra decomposes the evaluation process into small composable steps. The choice of the algorithms for each single operator in a logical query plan constitutes the physical query plan.

In the world of semistructured data, there are already several algebras with varying operators and algebra domain, as well as expressivity and power. Defining a query algebra for the simulation unification of Xcerpt is hence an obvious step in the development of an efficient evaluation method for the language.

Besides the classical purpose of optimization by rewriting, the goal of the simulation unifi-

[3] "Trashing" denotes a state of the virtual memory management of the operating system, but this term is also applicable to any caching system. A trashing memory management moves the memory pages to the secondary storage device (such as the hard disk) a running process will need in the near future. This effect occurs if the primary storage device is not large enough to provide an adequate memory window for the current running processes so that they "steal" memory pages from each other or from themselves.

cation algebra (which is called Xcerpt query algebra in the following) is to introduce a more deterministic approach to simulation unification than the so far existing constraint calculus (see section 2.2). Furthermore, the algebraic approach to simulation unification reveals further optimization possibilities beyond the current (still very efficient) implementation.

This thesis investigates both algorithmic optimization techniques and a query algebra for Xcerpt. It is structured into four sections. Section 2 gives a brief introduction into the rule based query language. Following the definition of the syntactic form of Xcerpt programs, section 2.2 focuses on the operational semantics of Xcerpt programs. The constraint calculus that is used to formalize the simulation unification is defined. This novel unification method specifies how query terms match data or construct terms when Xcerpt programs are evaluated.

Section 3 introduces a higher level algebra for simulation unification. Some already existing algebras for semistructured data and XML are investigated (section 3.1) before the Xcerpt query algebra principles are explained (section 3.2). Section 3.3 defines the Xcerpt query algebra, starting with a basic set of operators and extending it step by step until all (acyclic) query terms can be translated into algebraic expressions. The following section 3.4 defines rewriting rules for the Xcerpt query algebra. These rules allow query optimization based on static data and program analysis. After the rewriting rule definitions, the restrictions of the Xcerpt query algebra is discussed in section 3.5 as well as possibilities to lift them.

Section 4 investigates the complexity of the query algebra optimizations and of further optimizations. The optimizations presented in section 4 are not yet all built into the formal definition of the Xcerpt query algebra, but they are expected to be incorporated in the future. Section 4.1 first introduces the so-called matrix method that was developed by Sebastian Schaffert [Sch04] and is the basic implementation technique for the existing prototypic implementations. The complexity of the basic matrix method is investigated, especially in the presence of the special Xcerpt constructs `without` and `optional`. The following sections 4.2 and 4.3 present optimizations of the basic matrix method algorithm, especially focusing on the efficient handling of the `without` and `optional` constructs. Section 4.4 demonstrates the performance gain achieved by the improved algorithms.

2 Xcerpt: a Short Introduction

This section gives a quick overview of Xcerpt's syntax and part of its semantics. A more detailed treatment of the syntax and semantics of Xcerpt can be found in [Sch04] and in the Xcerpt use cases presented in [BBF+05].

In this section, the syntax of Xcerpt programs is presented first. Section 2.1 defines data, construct and query terms as well as rules and goals.

Section 2.2 focuses on the semantics of simulation unification. The semantics is defined with the help of a constraint calculus.

Section 2.3 finally investigates the semantics of the special query constructs `without` and `optional`. The semantics of these constructs is explained with the key concepts of simulation unification.

2.1 Xcerpt Syntax

The syntax of Xcerpt is based on a straightforward syntax of semistructured data and extends it with some new concepts for data querying and transformation. There are three term types which form the basis of Xcerpt, which are the data terms, query terms and construct terms. Data terms are used to represent semistructured data, while query terms are used to query data terms. They specify data patterns that the unification of Xcerpt, the simulation unification, tries to embed into the data. The process of embedding a query term into a data term is called simulation unification. A peculiarity of simulation unification is that for a given query and data term, the number of embeddings is not limited to one. There might be several possible embeddings of a query term into a data term. The results that these embeddings yield are variable bindings. The third kind of terms, the construct terms, are used to build new data terms from these variable bindings. They may create one data term for each variable binding, or group several variable bindings together in one data term. Note that this is symmetric to the fact that there are multiple embeddings of a query term into a data term.

The basic semistructured data syntax is given first. A word in the following given grammar is called a data term, as it represents semistructured data.

The grammar for data terms makes use of undefined nonterminal symbols that are `oid`, `label`, `iri`, `string`, and `number`. Since the grammars for these nonterminals are intuitive and very verbose, they are left aside.

Definition 1 (Data Term) *The set of all data terms is denoted with $\mathcal{T}^d$ in the following. A data term is a word produced by the nonterminal* `<data-subterm>`[4] *and the following grammar.*

```
1       <data-term> := ( oid "@" )? <ns-label> <attributes>? <subterms> .
2        <ns-label> := (<ns-prefix> ":")? label .
3       <ns-prefix> := label | '"' iri '"' .
4      <attributes> := "(" <attribute> ("," <attribute>)* ")" .
5        <attribute> := <ns-label> "=" '"' string '"' .
6         <subterms> := <list> | <set> .
7             <list> := "[" (<data-subterm> ( "," <data-subterm> )*)? "]" .
8              <set> := "{" (<data-subterm> ( "," <data-subterm> )*)? "}" .
9    <data-subterm> := (<data-term> | '"' string '"' | number | "^" oid).
```

The square brackets `[ ]` define an ordered sequence of terms, while the curly braces { } define an unordered set of terms.

An XML document is translated into a single semistructured data term. However, the IDREF and IDREFS attributes of XML must be translated into Xcerpt attributes. The referencing mechanism of Xcerpt can not be used, since it is transparent. That is to say, an occurrence of a reference in a term is not distinguishable from a parent-child relationship. Bare in mind that the case is different in XML: here, a reference to a term with IDREF type attribute is defined, which is clearly distinguishable from a parent-child relationship.

[4] This is counterintuitive at first, but the constraint calculus in section 2.2 needs more than just the data terms as domain.

Example 1 (Data Term) *The following example represents an address book containing persons. Each person entry consists of a name and possibly a phone number, email address or icq number.*

```
addressbook{ person[ name{ "Francois Bry" },
                     phone{ "0892110123" },
                     email{ "bry@lmudomain.de" }
                   ],
             person[ name{ "Sebastian Schaffert" },
                     phone{ "0892110124" },
                     email{ "schaffert@lmudomain.de" }
                   ],
             person[ name{ "Tim Furche" },
                     email{ "furche@lmudomain.de" }
                   ],
             person[ name{ "Andreas Schroeder" },
                     icq{ "12341234" }
                   ]
           }
```

The first extension of the data term grammar are query terms. A query term is intended to query a data term and provides patterns that define only the part of the structure relevant for the query. Query terms use different forms of ellipsis, one of which is called partial subterm specification in the following, denoted by the use of double braces. It indicates that there might be other data terms besides the specified ones that are not relevant for the retrieval of the data that the programmer is interested in.

Definition 2 (Query Term) *The set of all query terms is denoted by $\mathcal{T}^q$ in the following. A query term is a word produced by the nonterminal* <query-subterm> *in the following grammar.*

```
1         <query-term> := ( oid "@" )? <ns-label> <attributes>? <subterms> .
2           <ns-label> := (<ns-prefix> ":")? (label | "var" label | regexp) .
3          <ns-prefix> := label | '"' iri '"' | "var" label | regexp.
4         <attributes> := <total-attributes> | <partial-attributes>.
5    <total-attributes> :=  "(" <attribute> ( "," <attribute> )* ")" .
6  <partial-attributes> := "((" <attribute> ( "," <attribute> )* "))" .
7           <attribute> := <ns-label> "=" ('"' string '"' | "var" label | regexp)
8                        | "optional" <attribute> | "without" <attribute>
9                        | "var" label .
10           <subterms> := <total-list> | <partial-list>
11                       | <total-set>  | <partial-set> .
12         <total-list> :=  "[" <subterms-list>? "]" .
13       <partial-list> := "[[" <subterms-list>? "]]" .
14          <total-set> :=  "{" <subterms-list>? "}" .
15        <partial-set> := "{{" <subterms-list>? "}}" .
16      <subterms-list> := <query-subterm> ("," <query-subterm>)* .
17      <query-subterm> := ("position" ("var" label | number))?
```

```
18                    ( "var" label ("as" <query-subterm>)?
19                    | "desc" <query-subterm>
20                    | "optional" <query-subterm>
21                    | "without" <query-subterm>
22                    | regexp | '"' string '"' | number | "^" oid
23                    | <query-term>) .
```

Beyond the incomplete subterm specifications, there are other concepts introduced by query terms. One natural extension over data terms is the use of regular expressions instead of labels or strings. If a label is in the language induced by a regular expression, then the match is successful.

Most importantly, variables in query terms are allowed. When a query term is unified with a data term, the variables are bound to the data terms the unification allows. The data terms bound to a variable can be further restricted by a so-called as-pattern. This pattern is again a query term, and this query term must unify with the data term bound by the restricted variable. The unification is successful only when the data term and the as-pattern query term unify.

A further extension is the `desc` query construct. It specifies a subterm at an arbitrary depth below the parent term. This is also a form of ellipsis or incomplete specification, but in depth instead of breadth.

The `position` construct allows to specify or query the position of a subterm in the parent term. Position constructs might be inconsistent with the order constraint of a query term list. For example, the query `f[position 2 a[ ], position 1 b[ ] ]` is not satisfiable.

Another new concept are optional and negated subterms. Optional subterms allow to specify a query part that is not mandatory in a successful match, but must be matched if possible. Satisfying the last requirement needs a considerable computational effort. This is further investigated in the sections about the semantics (section 2.2) and about the algebraic treatment of optional terms (section 3.3.6). Subterm negation, expressed with the `without` keyword, is only sensible in partial query lists since negated terms prohibit the existence of terms in the data. On the other hand, the totality requirement of total subterm lists already prohibits the existence of any terms besides those specified in the query. A peculiarity about negation is that a variable within a negated term does not produce a binding, but merely imposes constraints on bindings. This means that if the matching allows to bind a negated variable to a part of a data term, the positive occurrence of the same variable may not match any such data term. Therefore, each variable occurring within a negated term must also occur outside of any negation. This is an important aspect for the formalization of the query evaluation.

Example 2 (Query Term) *Assume that the persons without phone number must be retrieved from the address book, and furthermore their name and their email must be retrieved where possible. The query for this task is the following:*

```
addressbook{{
    person[[
        name{ var Name },
        without phone{{ }},
        optional email{ var Email }
    ]]
}}
```

Note that an optional query term is necessary, since the email address may be missing as well.

Construct terms are a second kind of data term extension. Construct terms define how to use the variable bindings produced by the query terms to create new data terms.

Definition 3 (Construct Term) *The set of all construct terms is denoted by $\mathcal{T}^c$ in the following. A construct term is a word produced by the nonterminal* `<cons-subterm>` *in the following grammar.*

```
1        <cons-term> := ( oid "@" )? <ns-label> <attributes>? <list> .
2         <ns-label> := ( <ns-prefix> ":" )? (label | "var" label) .
3        <ns-prefix> := label | '"'iri'"' | "var" label .
4       <attributes> := "(" <attribute> ( "," <attribute> )* ")" .
5        <attribute> := <ns-label> "=" ('"' string '"' | "var" label) .
6             <list> := <ordered-list> | <unordered-list> .
7     <ordered-list> := "[" <cons-subterms>? "]" .
8   <unordered-list> := "{" <cons-subterms>? "}" .
9    <cons-subterms> := <cons-subterm> ("," <cons-subterm>)* .
10    <cons-subterm> := ("position" (strvar label | number))?
11                      ( "var" label | '"' string '"' | number | "^" oid
12                      | "optional" <cons-subterm> ("with default" <cons-subterm>)?
13                      | <agg-subterm> | <cons-term>) .
14     <agg-subterm> := "all" <cons-subterm> <group-spec>? <order-spec>?
15                      | "some" number <cons-subterm> <group-spec>? <order-spec>?
16      <group-spec> := "group by" <var-list> .
17       <oder-spec> := "order by" <var-list> ("ascending" | "descending")? .
18        <var-list> := "[" ("var" label (, "var" label)* "]" .
```

Construct terms introduce grouping, ordering and aggregation constructs. These constructs are necessary to group several answers into one single term. Although being very useful, these constructs have deep implications for the semantics and the evaluation process. Especially, the aggregation constructs imply that there is no minimal model semantics for Xcerpt programs.

Example 3 (Construct Terms) *Assume that a data term must be built with all the persons without a phone number. The variables* `Email` *and* `Name` *are retrieved in the query example. All matchings are aggregated in one single data term.*

```
missingphone{
    all person[
        name{ var Name },
        email{ optional var Email with default "missing" }
    ]
}
```

The sets of all data terms ($\mathcal{T}^d$), query terms ($\mathcal{T}^q$) and construct terms ($\mathcal{T}^c$) have two interesting properties. First, it holds that $\mathcal{T}^d \subset \mathcal{T}^q$, that is, every data term is also a query term. Second, $\mathcal{T}^d \subset \mathcal{T}^c$, but also note that $\mathcal{T}^q \cap \mathcal{T}^c \neq \mathcal{T}^d$ due to the possible variable occurrence in both query and construct terms.

The three basic term types were defined, but there is still no connection between construct and query terms. Rules and goals are entities specifying transformations, and associate query terms and construct terms. A rule consists of a rule body, which is a formula of query terms, and a rule head. The rule body specifies the data retrieval, while the rule head specifies how the query results are composed and transformed to create a new data term.[5] Goals are special rules that specify the output of a program and the entry points for a backward chaining evaluation of the program.

Definition 4 (Rules and Goals) *A program is a word produced by the nonterminal* `<program>` *in the following grammar.*

```
1    <program> := <goal> (<goal> | <rule>)* .
2       <goal> := "GOAL" <cons> ("FROM" <query>)? ("WHERE" <clause>)? "END" .
3       <rule> := "CONSTRUCT" <cons> ("FROM" <query>)? ("WHERE" <clause>)? "END" .
4       <cons> := <cons-term> | "out" "{" "resource" '"' iri '"' <cons-term> "}".
5      <query> := <and-query> | <or-query> | <not-query>
6                | <ress-query> | <query-term>.
7  <and-query> := "and" "{" <query> ("," <query>)+ "}".
8   <or-query> := "or" "{" <query> ("," <query>)+ "}".
9  <not-query> := "not" <query> .
10<ress-query> := "in" "{" "resource" '"' iri '"' <query> "}" .
11    <clause> := <and-clause> | <or-clause> | <not-clause> | <comparison> .
12<and-clause> := "and" "{" <clause> ("," <clause>)* "}".
13 <or-clause> := "or" "{" <clause> ("," <clause>)* "}".
14<not-clause> := "not" <clause> .
15<comparison> := "var" label <op> (regexp | '"' string '"' | number).
16        <op> := "=" | "<" | ">" | "<=" | ">=" | "~".
```

The allowed comparison operators are regular expression matching (`~`) and order comparisons (`=`, `<`, `>`, `<=`, `>=`) on number and strings.

As the reader may have noticed, the grammar above allows a too large set of programs. An Xcerpt program in fact must satisfy additional constraints besides just being a word in the presented grammar.

[5] This is a forward chaining view of a rule. Viewed from a backward chaining approach, a rule specifies what the transformation may produce and how to achieve it.

Position Consistency. A position specification must be satisfiable. Since position specifications may not occur in total and ordered subterm specifications, it must be checked whether two position specifications within one subterm specify one and the same position, or if they are conflicting with the order constraint of partial and ordered specifications as in e.g.

```
f[[
    position 3 /*/{{ }},
    position 1 /*/{{ }}
]]
```

Without and Totality. As mentioned before, without may not occur within a total subterm specification.

Variable Pattern Compatibility. If a variable is restricted by a pattern more than once in the normal form of the rule body, the patterns must be compatible. That is to say, it must be possible to satisfy both patterns by a single data term.

Range Restriction. A simple version of range restriction would require that each variable in the construction part of a rule must occur in the body of the same rule. This simple version of range restrictedness is insufficient, because it does not consider variable negation, optionality and disjunctions. All non-optional variables occurring in the rule head must occur in all disjuncts of the body DNF. The same holds for negated variables: for each negated variable in a query term, there must be a positive occurrence of the same variable that is in conjunction with this negative occurrence. Similarly, if a variable occurs only within optional query subterms in any disjunct of the body DNF, then it also must occur within an optional term in the rule head.[6]

Example 4 (Xcerpt Program) *The query, construct and data term examples constitute the following program:*

```
GOAL
    missingphone{ all person[ name{ var Name },
                                    email{ optional var Email
                                           with default "missing" }
                                ]
                }
FROM
    addressbook{{ person [[ name{ var Name },
                              without phone{{ }},
                              optional email{ var Email }
                           ]]
                }}
END
```

[6] For the detailed definition of range restriction, see [Sch04].

```
CONSTRUCT
    addressbook{ person[ name{ "Francois Bry" },
                         phone{ "0892110123" },
                         email{ "bry@lmudomain.de" }
                       ],
                 person[ name{ "Sebastian Schaffert" },
                         phone{ "0892110124" },
                         email{ "schaffert@lmudomain.de" }
                       ],
                 person[ name{ "Tim Furche" },
                         email{ "furche@lmudomain.de" }
                       ],
                 person[ name{ "Andreas Schroeder" },
                         icq{ "12341234" }
                       ]
               }
END
```

Executing the program above will output the following term.

```
  missingphone{ person[ name{ "Tim Furche" },
                        email{ "furche@lmudomain.de" }
                      ],
                person[ name{ "Andreas Schroeder" },
                        email{ "missing" }
                      ]
              }
```

This section did not provide examples for every query or construct term feature. The presentation of the syntax was very condensed, even to the point of leaving out the formal definition for range restrictedness, variable well formedness and other syntactic restrictions. Some of these syntactic restrictions are context free, and it is possible to give an EBNF-grammar expressing them. In this case, a comprehensive and simpler grammar that allows too many terms was favored over a more complete but complex grammar.

Xcerpt is a rule based query language influenced from logic programming. It offers several constructs tailor-made for querying semistructured data. The unification method presented in the following section is also an adaptation of classical unification to the requirements of semistructured data querying. This section presented the three base term types of Xcerpt: data terms, query terms and construct terms. Query and construct terms extend data terms with variables and special constructs for semistructured data querying and construction. Furthermore, rules and goals were introduced consisting of a construct term, the rule head, and a query formula, the rule body. Goals are special rules that produce the output of a program.

2.2 Xcerpt Semantics: Simulation Unification

The semantics of Xcerpt programs is specified by a constraint substitution calculus. Within this calculus, the unification of query terms with construct terms is defined as rules that can be applied in arbitrary order. The calculus operates on constraints. The rules are constraint replacement rules which specify that if a certain combination of constraints is in conjunction within the store, these constraints can be removed from the store and replaced by the constraints in the conclusion. A calculus rule has the following form:

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

The rule specifies that the constraint $C_1 \wedge \ldots \wedge C_n$ can be replaced by D.

In the following constraint calculus, meta-variables are used to specify labels, terms, and variables. The variables used for terms are s, t, c, and q. Labels are denoted with l, ns, and *label* with possible adornments. The meta-variables used for variables are X and Y with possible adornments.

Definition 5 (Constraints) *The set of constraints, Constraint, is the smallest set that fulfills the following conditions:*

- $True \in Constraint$. *(Boolean primitive)*
- $False \in Constraint$. *(Boolean primitive)*
- $\forall t \in (\mathcal{T}^q \cup \mathcal{T}^c)\ \forall c \in \mathcal{T}^c.\ t \preceq_{su} c \in Constraint$. *(Simulation constraint)*
- $\forall c \in Constraint.\ \neg c \in Constraint$. *(Negation)*
- $\forall \{c_1, \ldots, c_n\} \subseteq Constraint.\ \bigwedge_{i=1}^{n} c_i \in Constraint$ *and* $(\bigvee_{i=1}^{n} c_i) \in Constraint$. *(Boolean connectives)*

It holds that $\bigwedge_{i=1}^{0} = True$ *and* $\bigvee_{i=1}^{0} = False$

Furthermore, constraints of the form var $X \preceq_{su} t$ *or* $t \preceq_{su}$ *var* X *are called atomic simulation constraints. All constraints built without use of the negation and connective construction rules are called atomic constraints.*

The constraint $t \preceq_{su} c$ expresses that t must simulate into c. The symbol used, $\preceq_{su}$, is the simulation relation.

The constraint calculus usually starts with a simulation constraint $q \preceq_{su} c$. With the application of the calculus rules, it is possible to deduce a formula of atomic simulation constraints representing the variable constraints that must hold in order to solve the initial simulation problem. When a formula of atomic simulation constraints is reached, this formula represents all possible embeddings of the query q in the construct term c.

A substitution can be deduced from a conjunction of atomic simulation constraints of the form *var* $X \preceq_{su} t$. A formula of atomic simulation constraints can be transformed into a

set of substitutions. Substitutions are used to build the ground instances of the associated construct terms. When grouping or aggregation occurs in the construct term, however, a single substitution is not sufficient to ground the term. Grounding aggregating or grouping construct terms requires to gather all embeddings of the rule body and create a set of substitutions from all such embeddings. The set of constraints computed from all embeddings is used to ground the aggregating or grouping rule head.

Furthermore, due to the complex nature of grouping, the constraint calculus fails on construct terms containing grouping constructs. For this reason, grouping and aggregating construct terms must be grounded before the embeddings of a query term can be determined. For details on substitution generation and term grounding see also [Sch04].

The operational semantics of an Xcerpt program is specified using four kinds of rules. The first kind is constraint simplification rules. Second, so-called decomposition rules are needed. These rules specify how a single non-atomic simulation constraint is decomposed to reach atomic simulation constraints. The third kind of rules, the consistency verification rules, is necessary to check the validity of the atomic simulation constraints. They are applied if one variable occurs within an as-pattern of another variable, or if a variable occurs several times within a query. Finally, the rule chaining requires a fourth type of calculus rules, which are the chaining rules. This thesis focuses on optimization of single queries, so that chaining rules are not discussed. Interested readers can refer to [BSS04], [Sch04] and [BS03] for these rules.

The simulation unification calculus uses a normalized form of terms in order to reduce the number of necessary calculus rules, and this term normalization is introduced first.

2.2.1 Term Normalization

The constraint calculus considers a normalized form of query and construct terms. This allows to keep the following set of rules small and simple. The algebra for simulation unification (section 3), however, do not need the following normalization rules.

Namespace Distribution. Namespaces are completely distributed over all term labels. If a term has no namespace associated, the default namespace is used as term namespace. Hence, each term label has a namespace.

Attribute Normalization. The query attributes are handled as special terms nested within the term with attributes. Term attributes are translated as follows:

An attribute list with single braces is translated into an unordered and total query term list. This translation applies to both query and construct terms (and hence to data terms as well).

$$ns : label(\ ns_1 : label_1 = value_1, \ldots, ns_n : label_n = value_n\ )\{\ t_1, \ldots, t_m\ \} = \\ ns : label\{\ \texttt{attributes}\{\ ns_1 : label_1\{value_1\}, \ldots ns_n : label_n\{value_n\}\ \}, t_1, \ldots, t_m\ \}$$

An attribute query list with double braces is translated into an unordered and partial query term list. This translation applies to query terms only.

$$ns : label((\ ns_1 : label_1 = value_1, \ldots, ns_n : label_n = value_n\ ))\{\ t_1, \ldots, t_m\ \} = \\ ns : label\{\ \texttt{attributes}\{\{\ ns_1 : label_1\{value_1\}, \ldots ns_n : label_n\{value_n\}\ \}\}, t_1, \ldots, t_m\ \}$$

The translation rule also applies to terms with ordered or partial child specifications, preserving the braces of the subterm list.

Furthermore, an empty `attributes` element with double braces is added to query terms specifying no attributes, and an empty `attributes` element with single braces is added to construct and data terms without attributes.

Optional Substitution. Queries containing optional constructs are transformed in the following way: For each `optional q` subterm, the entire query term containing the optional term into a new disjunct in the query. In the original query term, `optional q` is replaced by `q`. In the duplicate, `optional q` is replaced by `without q`, if the query term list is partial (i.e. has double braces as delimiters). In total query term lists, the optional subterm is removed in the copy without replacing it, since `without` is forbidden within total query term lists.

Example 5 (Optional Transformation) *Assume that the following query term is normalized by optional substitution.*

```
        persons{{ optional person[ name{{ var Name }},
                                   optional email{{ var email }},
                                   phone{{ var phone }}
                                 ]
                }}
```

Replacing the first optional leads to the query

```
    or{ persons{{ person[ name{{ var Name }},
                          optional email{{ var email }},
                          phone{{ var phone }}
                        ]
                }},
        persons{{ without person[ name{{ var Name }},
                                  optional email{{ var email }},
                                  phone{{ var phone }}
                                ]
                }}
}
```

Replacing the second optional term yields the following:

```
or{ persons{{ person[ name{{ var Name }},
                      email{{ var email }},
                      phone{{ var phone }}
                    ]
            }},
    persons{{ without person[ name{{ var Name }},
                              email{{ var email }},
                              phone{{ var phone }}
                            ]
            }},
    persons{{ person[ name{{ var Name }},
                      phone{{ var phone }}
                    ]
            }},
    persons{{ without person[ name{{ var Name }},
                              phone{{ var phone }}
                            ]
            }}
}
```

This substitution rule leads to an exponential growth of the query in the number of optional subterms. Optimization techniques to overcome this exponential blowup are found in section 4. The formal treatment however lacks rules for the handling of optional terms since they can be substituted as the rule specifies above.

The query normalization guarantees that every composite term has a namespace and `attributes` element. Hence, the calculus rules do not need to consider the case where the composite terms lack attributes or namespaces. Furthermore, the normalization eliminates optional query terms. The constraint calculus hence do not need to handle optional query terms, and the set of rules get smaller because of this normalization.

The following section introduces equivalence rules that are known from the classical boolean algebra.

2.2.2 Constraint Simplification and Negation

In order to handle negation appropriately, the calculus needs constraint simplification rules for negated constraints. The de-Morgan rules and the negation rules for boolean values hold in the constraint calculus.

$$\frac{\neg(C \wedge D)}{\neg C \vee \neg D} \qquad \frac{\neg(C \vee D)}{\neg C \wedge \neg D} \qquad \frac{\neg False}{True} \qquad \frac{\neg True}{False}$$

However, the double negation of a constraint is not equivalent to its positive form. The calculus rule only allows the following conclusion rule (from intuitionistic logic).

$$\frac{\neg\neg\neg C}{\neg C}$$

The reason for this is that unnegated atomic simulation constraints create variable bindings. A doubly negated simulation constraint is not supposed to do so, since it stems from subterms with `without` constructs. As stated before, variables within a negation does not produce any binding.

To handle negated variable bindings, a "negation as failure" rule must be introduced. Whenever a negated binding of a variable occurs without being in conjunction with any positive binding of the same variable, it may be reduced to false.

$$\frac{\neg(var\ X \preceq_{su} c) \text{ and no other positive binding for X exists}}{False}$$

This rule also applies to double negated atomic simulation constraints: the single negated simulation constraint can be replaced by false, and then the single negated false constraint can be replaced by true:

$$\frac{\frac{\neg\neg(X \preceq_{su} a)}{\neg False}}{True}$$

This is the way the calculus resolves doubly negated simulation constraints without creating variable bindings.

The most important deviation from classical boolean algebra is the negation as failure rule and the negation canceling rule that allows to cancel two negations if the constraint is triply negated.

The following section introduces the decomposition rules. These are the core rules of the constraint calculus for simulation unification. They decompose non-atomic simulation constraints into smaller simulation constraints. That is to say, the depth of the terms on the left hand and right hand side of the constraints is reduced.

2.2.3 Decomposition Rules

The decomposition rules can decompose a single non-atomic simulation constraint into several simulation constraints on the subterms of both terms. The aim of such constraint simplification rules is to reach atomic simulation constraints or to reduce the simulation constraint to boolean primitives.

The decomposition rules reducing the simulation constraint to boolean primitives are the "brace incompatibility" and the "left term empty" rules presented in the following.

Brace Incompatibility. If the query term specifies an ordered list and the data term has an unordered set of subterms, the simulation constraint can be reduced to $False$.

$$\frac{ns_1 : l_1[t_1, \ldots, t_n] \preceq_{su} ns_2 : l_2\{s_1, \ldots, s_m\}}{False} \qquad \frac{ns_1 : l_1[[t_1, \ldots, t_n]] \preceq_{su} ns_2 : l_2\{s_1, \ldots, s_m\}}{False}$$

Left Term Empty. If the left term is empty, the simulation constraint can be either reduced to term label matchings or $False$, depending on whether the right term is also empty and

whether the left term allows unification with a non-empty term. Depending on whether the left term specifies partial or total subterm grouping, the simulation constraint can either be reduced to $False$ or to two label matching tests, one for the namespace and one for the label of the node. Let $m \geq 0$.

$$\frac{ns_1 : l_1\{\{\ \}\} \preceq_{su} ns_2 : l_2\{s_1, \ldots, s_m\}}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2} \qquad \frac{ns_1 : l_1\{\{\ \}\} \preceq_{su} ns_2 : l_2[s_1, \ldots, s_m]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2}$$

$$\frac{ns_1 : l_1[[\ ]] \preceq_{su} ns_2 : l_2[s_1, \ldots, s_m]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2} \qquad \frac{ns_1 : l_1\{\ \} \preceq_{su} ns_2 : l_2\{s_1, \ldots, s_m\}}{False}$$

$$\frac{ns_1 : l_1\{\ \} \preceq_{su} ns_2 : l_2[s_1, \ldots, s_m]}{False} \qquad \frac{ns_1 : l_1[\ ] \preceq_{su} ns_2 : l_2[s_1, \ldots, s_m]}{False}$$

$$\frac{ns_1 : l_1\{\ \} \preceq_{su} ns_2 : l_2\{\}}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2} \qquad \frac{ns_1 : l_1\{\ \} \preceq_{su} ns_2 : l_2[]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2}$$

$$\frac{ns_1 : l_1[] \preceq_{su} ns_2 : l_2[]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2}$$

Label Simulation. There are three cases for label simulations such as $label_1 \preceq_{su} label_2$ that must be considered. The first case is that both labels are strings. This is the most common case and is handled with the left rule below. Here, the rule verifies whether both labels are the same. If the left label is a regular expression, the rule must ensure that the right label is a word in the language produced by the regular expression. If one of the two labels is a variable, the decomposition rules leave this simulation constraint unchanged, as it represents an atomic simulation constraint and the consistency verification rules handle these constraints. In cases where the right hand label is a variable and the left hand label is a regular expression is a most interesting one: in such a case, the regular expression matching test is deferred until an actual binding of the variable is present in the constraint store. Here, the transitivity rule applies and replace the variable by its binding, allowing to apply the second of the following three rules:

$$\frac{label_1 \preceq_{su} label_2}{label_1 = label_2} \qquad \frac{regexp \preceq_{su} label}{label \in L(regexp)} \qquad \frac{label \preceq_{su} var\ X}{var\ X \preceq_{su} label}$$

As Elimination. To eliminate an "as" construct, a simulation constraint requiring that the pattern query term simulates into the actual construct term is introduced. Furthermore, the rule adds the construct term as upper bound to the variable of the "as" construct. Additionally, the rule adds the query pattern as lower bound to the variable. This latter constraint may seem surprising at first, and that is related to the incompleteness of the simulation constraint calculus for grouping and aggregation. The lower bound to the variable can be ignored, if the right hand side term does not contain any grouping constructs.

$$\frac{X\ as\ q \preceq_{su} c}{q \preceq_{su} c \wedge q \preceq_{su} X \wedge X \preceq_{su} c}$$

Descendant Elimination. To eliminate descendant constructs, a simulation unification is introduced that tests whether the actual right hand side term unifies with the query pattern. The descendant query is furthermore satisfied if any of the construct term children unify with the descendant construct. The descendant elimination rule will then apply again with each child term of the current construct term.

$$\frac{desc\ q \preceq_{su} ns : l\{c_1, \ldots, c_m\}}{q \preceq_{su} ns : l\{c_1, \ldots, c_m\} \vee \bigvee_{i=1}^{m} desc\ q \preceq_{su} c_i}$$

$$\frac{desc\ q \preceq_{su} ns : l[c_1, \ldots, c_m]}{q \preceq_{su} ns : l[c_1, \ldots, c_m] \vee \bigvee_{i=1}^{m} desc\ q \preceq_{su} c_i}$$

Term Elimination. Term elimination is one of the more complicated decomposition rules. The term elimination rule decomposes a non-atomic simulation constraint into label simulations for both namespace and label and simulation constraints on the query term children and the construct term children.

To create every possible combination, mappings on terms with special properties are used in order to conform to all four kinds of composed query terms (i.e. ordered and total, ordered and partial, unordered and total, unordered and partial).

Definition 6 (Mapping Properties) *Let $N = \langle t_1, \ldots, t_n \rangle$ and $M = \langle s_1, \ldots, s_m \rangle$ be term sequences. Let furthermore $index : \{t_1, \ldots, t_n\} \cup \{s_1, \ldots, s_m\} \rightarrow Integer$ be an index function such that $index(t_i) = i$ and $index(s_i) = i$. Let G be the set of words producable by starting with the* `<agg-subterm>` *nonterminal in the construct term grammar (that is, all construct terms starting with an aggregation or grouping:* `all` *cterm,* `some` *n cterm,). A mapping $\pi : N \rightarrow M$ is called*

- index injective, *if for all $t_i, t_j \in N$ it holds that if $i \neq j$ and $\pi(t_i) \notin G$, then $\pi(t_i) \neq \pi(t_j)$.*
- index surjective, *if for all $t_i \in M$ there is a $s_j \in N$ such that $\pi(t_i) = s_j$.*
- index bijective, *if π is index injective and index surjective.*
- index monotonic, *if for all $t_i, t_j \in N$ it holds that if $i < j$ and $\pi(t_i) \notin G$, then $index(\pi(t_i)) < index(\pi(t_j))$.*
- position respecting, *if for all* `position` *$j\ t \in N$ it holds that $index(\pi($* `position` *$j\ t)) = j$.*
- position preserving, *if for all* `position` *$j\ t \in N$ it holds that $\pi($* `position` *$j\ t) =$* `position` *$j\ s$ and for all* `position` *var $X\ t \in N$ it holds that $\pi($* `position` *$X\ t) =$* `position` *$j\ s$ or $\pi($* `position` *var $X\ t) =$* `position` *var $Y\ s$.*

A mapping $\pi : N \rightarrow M$ is called completable if there is a mapping $\pi' : N \rightarrow M$ such that

- $\pi \subseteq \pi'$
- *π' is a real superset of π*

Note that the injectivity and monotonicity requirement is lifted for grouping and aggregating terms, since they may (and mostly will) produce more than one subterm. Hence, a mapping may assign more than one query term to the same construct term, if it is grouping or aggregating.

In the case of ordered subterm specifications, the decomposition rule must use index monotonic mappings, and for total subterm specifications index surjective mappings. Every mapping that is considered must be index injective; this restriction is imposed in order to guarantee that every query subterm matches its own data subterm, and to make sure that no two query subterms map to the same data subterm (except for aggregating and grouping construct terms as mentioned above). This constraint is useful to ensure that in the case the programmer specifies two identical subterms, data terms containing only one subterm does not satisfy the query.

Position respecting and position preserving mappings are necessary to handle position specification appropriately.

The following sets of mappings are used in the term elimination rules. Let $N = \langle t_1, \ldots, t_n \rangle$ and $M = \langle s_1, \ldots, s_m \rangle$.

- Π_{pp} is the set of all index injective and position preserving mappings from N to M.
- Π_{pp-bij} is the set of all index bijective and position preserving mappings from N to M.
- Π_{pr} is the set of all index injective and position respecting mappings from N to M.
- Π_{pr-bij} is the set of all position respecting and index bijective mappings from N to M.
- Π_{pr-mon} is the set of all position respecting and index monotonic mappings from N to M.
- $\Pi_{bij-mon}$ is the set of all index monotonic and index bijective mappings from N to M.

Note that the defined mapping sets depend on the two term sequences N and M. These term sequences will be $N = \langle q_1, \ldots, q_n \rangle$ and $M = \langle c_1, \ldots, c_m \rangle$ in the term decomposition rules below.

First, the case of partial and unordered query term unification with unordered construct terms is considered. The decomposition rule uses the set Π_{pp} of position preserving mappings in the case of a partial and unordered query term, because the valid mappings must respect injectivity and must map all position specifications of the query to a respective position specifying construct subterm.

$$\frac{ns_1 : l_1\{\{q_1, \ldots, q_n\}\} \preceq_{su} ns_2 : l_2\{c_1, \ldots, c_m\}}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 \wedge \bigvee_{\pi \in \Pi_{pp}} \bigwedge_{i=1}^{n} q_i \preceq_{su} \pi(q_i)}$$

When mapping an unordered and partial query term list to an ordered construct term list, the set of position respecting mappings Π_{pr} is used, because the construct subterm positions are meaningful and may hence be used to satisfy positional query specifications.

$$\frac{ns_1 : l_1\{\{q_1, \ldots, q_n\}\} \preceq_{su} ns_2 : l_2[c_1, \ldots, c_m]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 \wedge \bigvee_{\pi \in \Pi_{pr}} \bigwedge_{i=1}^{n} q_i \preceq_{su} \pi(q_i)}$$

In the case of a total and unordered query list, the surjectivity of the mapping must be imposed additionally. Position preserving and position respecting mappings must be used for the same reasons as for partial and unordered query term lists.

$$\frac{ns_1 : l_1\{q_1, \ldots, q_n\} \preceq_{su} ns_2 : l_2\{c_1, \ldots, c_m\}}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 \wedge \bigvee_{\pi \in \Pi_{pp-bij}} \bigwedge_{i=1}^{n} q_i \preceq_{su} \pi(q_i)}$$

$$\frac{ns_1 : l_1\{q_1, \ldots, q_n\} \preceq_{su} ns_2 : l_2[c_1, \ldots, c_m]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 \wedge \bigvee_{\pi \in \Pi_{pr-bij}} \bigwedge_{i=1}^{n} q_i \preceq_{su} \pi(q_i)}$$

For partial and ordered query term lists, index monotonic mappings must be used to respect the subterm order. Furthermore, position respecting mappings are used to map query subterms with position specifications.

$$\frac{ns_1 : l_1[[t_1, \ldots, t_n]] \preceq_{su} ns_2 : l_2[s_1, \ldots, s_m]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 \wedge \bigvee_{\pi \in \Pi_{pr-mon}} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}}$$

Total and ordered query term lists require index surjective and index monotonic mappings. Since a total and ordered query term list may not contain any subterm with position specification, they can be ignored.

$$\frac{ns_1 : l_1[t_1, \ldots, t_n] \preceq_{su} ns_2 : l_2[s_1, \ldots, s_m]}{ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 \wedge \bigvee_{\pi \in \Pi_{bij-mon}} \bigwedge_{i=1}^{n} t_i \preceq_{su} s_{\pi(i)}}$$

Term Elimination with Negation. The term elimination rule must be extended to handle `without` constructs on the left hand side. Negated terms have a lower priority than positive terms, since they do not need to match a construct term. For this reason, the mappings are first restricted on the positive query terms. A mapping must be surjective on the positive query terms. For a given mapping π, mapping extensions π' to the negative query are considered. The mapping π together with the mapping extension π' must satisfy the mapping properties required by the query type. This means for example that the mapping π extended with π' must be monotonic for a partial and ordered query.

The mapping extensions do not need to be total on the negative query terms, since not every negative query term may be mapped onto a construct term. Because of this lack of totality, it is not sufficient to consider a single mapping extension π' to a positive mapping. It is necessary to consider *all* mapping extension $E(\pi)$ of a given positive mapping π and add all constraints of these extensions to the positive mapping constraints. This is necessary for a sensible negation semantics (see section 2.3), but also has deep implications for the handling of optional and negated constructs, as section 3.3.5, 3.3.6 and 4 demonstrate.

The query term sequence on the left hand side was $N = \langle q_1, \ldots, q_n \rangle$. For this sequence, the positive subterms of N are defined as $Pos = \langle q_+ \mid q_+ \in N, q_+ \neq without\ q \rangle$ as the and $Neg = \langle q_- \mid q_- \in N, q_- = without\ q \rangle$ is defined as the negative subterms of N.

A positive mapping is a total mapping on the positive terms, $\pi : Pos \to M$, while a mapping extension is a possibly partial mapping on the negative terms: $\pi' : Neg \to M$.

The set of considered mapping extensions differ for each query type. The following mapping extension sets to a given positive mapping π exists:

- $E_{pp}(\pi)$ is the set of all index injective and position preserving mapping extensions of π.
- $E_{pr}(\pi)$ is the set of all index injective and position respecting mapping extensions of π.
- $E_{pr-mon}(\pi)$ is the set of all position respecting and index monotonic mapping extensions of π.

With these extension sets defined, the decomposition rules can be defined as follows:

$$\frac{ns_1 : l_1\{\{q_1, \ldots, q_n\}\} \preceq_{su} ns_2 : l_2\{c_1, \ldots, c_m\}}{\begin{array}{rcl} ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 & \wedge & \bigvee_{\pi \in \Pi_{pp}} (\bigwedge_{(q_+,c)\in\pi} q_+ \preceq_{su} c \\ & \wedge & \bigwedge_{\pi' \in E_{pp}(\pi)} \bigwedge_{(q_-,c)\in\pi'} q_- \preceq_{su} c) \end{array}}$$

$$\frac{ns_1 : l_1\{\{q_1, \ldots, q_n\}\} \preceq_{su} ns_2 : l_2[c_1, \ldots, c_m]}{\begin{array}{rcl} ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 & \wedge & \bigvee_{\pi \in \Pi_{pr}} (\bigwedge_{(q_+,c)\in\pi} q_+ \preceq_{su} c \\ & \wedge & \bigwedge_{\pi' \in E_{pr}(\pi)} \bigwedge_{(q_-,c)\in\pi'} q_- \preceq_{su} c) \end{array}}$$

$$\frac{ns_1 : l_1[[t_1, \ldots, t_n]] \preceq_{su} ns_2 : l_2[s_1, \ldots, s_m]}{\begin{array}{rcl} ns_1 \preceq_{su} ns_2 \wedge l_1 \preceq_{su} l_2 & \wedge & \bigvee_{\pi \in \Pi_{pr-mon}} (\bigwedge_{(q_+,c)\in\pi} q_+ \preceq_{su} c \\ & \wedge & \bigwedge_{\pi' \in E_{pr-mon}(\pi)} \bigwedge_{(q_-,c)\in\pi'} q_- \preceq_{su} c) \end{array}}$$

Note that only the partial query term list specifications (with list delimiters `{{ }}` and `[[ ]]`) are considered, since negation is not allowed to occur in total query term lists.

Without Elimination. The only step remaining after the mapping of negated query terms to construct terms is the elimination of the without construct. When removing a without construct from the left hand side, the whole simulation constraint is negated.

$$\frac{without\ q \preceq_{su} c}{\neg(q \preceq_{su} c)}$$

Position Elimination. After the mapping of query subterms to construct subterms, position specifications must still be handled.

When mapping query terms with position specifications, the position preserving and position respecting mappings already ensured that the query terms were mapped to the right construct term counterpart. Hence, removing the position specification without further consideration is valid in most cases.

However, the position respecting mappings which were used in the case of ordered construct terms on the right hand side may map position specifying query terms to construct terms without position specification. To resolve such a case, the *index* function that maps a term to its position within the parent is used.

Note however that this index function is context dependent; as Xcerpt handles references transparently, graphs must be handled. In a graph, a node does not have a single parent

node, but several. For each parent node, the child node has a different position. Also note that the *index* function may not be used on subterms of unordered queries, since such terms do not have a position. Respecting these constraints, the position elimination rules can be defined.

$$\frac{position\ var\ X\ q \preceq_{su} position\ var\ Y\ c}{var\ X \preceq_{su} var\ Y \wedge q \preceq_{su} c} \qquad \frac{position\ var\ X\ q \preceq_{su} c}{var\ X \preceq_{su} index(c) \wedge q \preceq_{su} c}$$

Mapping a position specification to a construct term without a position specification can only occur in the case of position respecting mappings. In this case, the index of the construct subterm is well defined.

$$\frac{position\ p\ q \preceq_{su} position\ p\ c}{q \preceq_{su} c} \qquad \frac{q \preceq_{su} position\ p\ c}{q \preceq_{su} c} \qquad \frac{q \preceq_{su} position\ var\ X\ c}{q \preceq_{su} c}$$

Grouping and Aggregation Incompleteness. Note that most of the special construction term constructs of Xcerpt were left aside. The calculus can not handle the aggregation, grouping, sorting and optional terms with default values yet. For this reason, each of these constructs must be resolved (i.e. grounded) before simulation unification is applicable. What the simulation unification calculus can do, however, is testing whether there is a potential match of a query term with an aggregating or grouping construct term.

The constraints generated by the calculus while testing the unifiability of both terms will be discarded. At this stage of calculation, it is just possible to determine whether the simulation constraint is potentially satisfiable. The actual unification will be performed after grounding the construct term on the right hand side.

$$\frac{q \preceq_{su} \texttt{all}\ c\ \texttt{group by}\ varlist_1\ \texttt{order by}\ varlist_2}{q \preceq_{su} c \neq False}$$

$$\frac{q \preceq_{su} \texttt{some}\ n\ c\ \texttt{group by}\ varlist_1\ \texttt{order by}\ varlist_2}{q \preceq_{su} c \neq False}$$

$$\frac{q \preceq_{su} \texttt{optional}\ c_1\ \texttt{with default}\ c_2}{q \preceq_{su} c_1 \neq False \vee q \preceq_{su} c_2 \neq False}$$

The decomposition rules presented in this section reduced the complexity of non-atomic simulation constraints. There are simple decomposition rules for special cases. The most important decomposition rule is the term elimination rule. This rule is complex, especially when treating negated query terms as well. The term elimination rule defines how a list of query terms matches a list of construct terms, and is hence an important rule to understand. Section 2.3 investigates and explains the semantics of the decomposition rules on small examples and provide rules for reading and writing queries with optional and negated query subterms. Furthermore, the Xcerpt query algebra (section 3) and the optimizations presented in section 4 focus mostly on the term elimination rule. The term elimination rule is one of the most important rules of the constraint calculus, as optimizations of this rule lead to considerable performance benefits.

However, the decomposition rules can create several atomic constraints on the same variable. In this case, the pairwise consistency of the variable constraints must be verified. The consistency rules presented in the following section handle this verification.

2.2.4 Consistency Rules

The consistency rules test the compatibility of several atomic simulation constraints on the same variable. The clearest motivation for consistency verification is that if a variable is bound within several subqueries,[7] all generated atomic simulation constraints on the variable must be compatible. The compatibility requirement is expressed by simulation in both directions, that is by bisimulation.

Consistency. The consistency rule adds two simulation constraints to represent the bisimulation requirement. Furthermore, one of the two variable bindings can be omitted, since both bindings are equivalent (in terms of bisimulation) if the bisimulation succeeds.

$$\frac{\begin{array}{c} var\ X \preceq_{su} t_1 \\ var\ X \preceq_{su} t_2 \end{array}}{var\ X \preceq_{su} t_1 \wedge t_1 \preceq_{su} t_2 \wedge t_2 \preceq_{su} t_1}$$

Consistency with Negation. When a variable constraint coincides with a negated constraint of the same variable, the bisimulation must fail.

$$\frac{\begin{array}{c} var\ X \preceq_{su} t_1 \\ \neg(var\ X \preceq_{su} t_2) \end{array}}{var\ X \preceq_{su} t_1 \wedge \neg(t_1 \preceq_{su} t_2 \wedge t_2 \preceq_{su} t_1)}$$

Similarly to the case with a single variable negation, the consistency between single negated and double negated variables must be ensured.

$$\frac{\begin{array}{c} \neg\neg(var\ X \preceq_{su} t_1) \\ \neg(var\ X \preceq_{su} t_2) \end{array}}{\neg(var\ X \preceq_{su} t_2) \wedge \neg(t_1 \preceq_{su} t_2 \wedge t_2 \preceq_{su} t_1)}$$

Since double negated variable constraints are equivalent to unnegated variable constraints except for the fact that they do not generate variable bindings, two more rules are necessary to ensure consistency between both (positive and doubly negated) kinds of atomic simulation constraints.

$$\frac{\begin{array}{c} \neg\neg(var\ X \preceq_{su} t_1) \\ var\ X \preceq_{su} t_2 \end{array}}{var\ X \preceq_{su} t_1 \wedge \neg\neg(t_1 \preceq_{su} t_2 \wedge t_2 \preceq_{su} t_1)}$$

$$\frac{\begin{array}{c} \neg\neg(var\ X \preceq_{su} t_1) \\ \neg\neg(var\ X \preceq_{su} t_2) \end{array}}{\neg\neg(var\ X \preceq_{su} t_1) \wedge \neg\neg(t_1 \preceq_{su} t_2 \wedge t_2 \preceq_{su} t_1)}$$

Transitivity. The transitivity rule applies if a simulation constraint has an occurrence of a variable on the right hand side while it occurs on the left hand side of another atomic

[7] Handling several occurrences of the same variable requires joins between the variable bindings, and that is in fact what this consistency rule does.

simulation constraint.

$$\frac{t_1 \preceq_{su} t_1' \text{ such that } t_1' \text{ contains } var\ X \quad var\ X \preceq_{su} t_2}{t_1 \preceq_{su} t_1'[t_2/var\ X] \wedge var\ X \preceq_{su} t_2}$$

Transitivity with Negation. Similarly to the case of consistency verification, variants of the transitivity rule for negated constraints are necessary.

$$\frac{\neg(t_1 \preceq_{su} t_1') \text{ such that } t_1' \text{ contains } var\ X \quad var\ X \preceq_{su} t_2}{var\ X \preceq_{su} t_2 \wedge \neg(t_1 \preceq_{su} t_1'[t_2/var\ X])}$$

The first rule handles the negation in a straightforward way. However, if the negation is on the atomic simulation constraint, the the non-atomic positive simulation constraint must be kept in order to prevent information loss.

$$\frac{t_1 \preceq_{su} t_1' \text{ such that } t_1' \text{ contains } var\ X \quad \neg(var\ X \preceq_{su} t_2)}{\neg(var\ X \preceq_{su} t_2) \wedge t_1 \preceq_{su} t_1' \wedge \neg(t_1 \preceq_{su} t_1'[t_2/var\ X])}$$

The cases of doubly negated constraint are handled similarly:

$$\frac{\neg(t_1 \preceq_{su} t_1') \text{ such that } t_1' \text{ contains } var\ X \quad \neg\neg(var\ X \preceq_{su} t_2)}{\neg\neg(var\ X \preceq_{su}) t_2 \wedge \neg(t_1 \preceq_{su} t_1'[t_2/var\ X])}$$

$$\frac{\neg\neg(t_1 \preceq_{su} t_1') \text{ such that } t_1' \text{ contains } var\ X \quad \neg(var\ X \preceq_{su} t_2)}{\neg(var\ X \preceq_{su} t_2) \wedge \neg\neg(t_1 \preceq_{su} t_1') \wedge \neg(t_1 \preceq_{su} t_1'[t_2/var\ X])}$$

The consistency rules presented in this section handle atomic simulation constraints. The rules allow to verify the compatibility of two atomic simulation constraints. However, the verification may introduce non-atomic simulation constraints that need to be decomposed with the decomposition rules. For this reason, the cost of consistency verification depends heavily on the Xcerpt program and its input.

The calculus for simulation unification contains three kinds of rules. There are decomposition rules who decompose non-atomic simulation constraints to simpler ones. Furthermore, there are consistency rules that verify the consistency of atomic simulation constraints. Note that these consistency verification rules introduce non-atomic simulation constraints that must be decomposed again. The third kind of rules is boolean equivalence rules, but the classical negation equivalence does not hold in Xcerpt. This is due to the fact that the negation used in Xcerpt is not equivalent to classical negation.

The next section investigates the peculiarities of negated and optional queries, and provides rules for the understanding of queries with negated parts.

2.3 Optional and Negated Term Semantics

The semantics of `without` (and its influence to the `optional` construct in partial queries) may seem overly complicated at first. However, the definition of the semantics is well considered and succeeds in giving common but difficult query patterns a precise meaning, especially in partial and ordered queries. This section demonstrates the peculiarities of both constructs on examples, and give best advice when using `optional` and `without`.

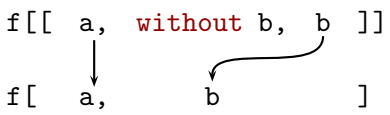
First, consider query term lists with both positive and negative query terms potentially matching the same data terms (as in e.g. `f{{ a, without a }}`). The priority of positive and negative subterms is clearly defined in the constraint calculus. The positive mappings are extended to the negative terms which means that the positive terms have priority over the negative. The negative terms may only match the data terms respecting the constraints imposed by the positive mapping; it is for example impossible for a negative query term to match a data term that a positive data term did match.

Example 6 (Negation and Injectivity constraints) *Consider for example the query* `f{{ a, without a }}` *and the data term* `f{ a }`*. The positive query subterm* `a` *is mapped to the single data subterm* `a` *before any extension for the negated subterm* `without a` *is searched. Intuitively, the positive subterm "possesses" the subterm* `a` *and the negated subterm is not allowed to "steal" it; the positive subterms have hence priority over negative ones.*

In ordered queries, it is forbidden for negative queries to match data terms outside of the range specified by the last preceding and the first following positive term. The negated term must respect the monotonicity constraint, which reduces the possible matches of negated terms and increases the number of solutions. Intuitively, negation is not so restrictive in ordered query terms as in unordered.

Example 7 (Negation and Monotonicity constraints) *Assume the query* $q =$ `f[[ a, without b, b ]]` *and the data terms* $d_1 =$ `f[ a, b ]` *and* $d_2 =$ `f[ a, b, b ]`*.*

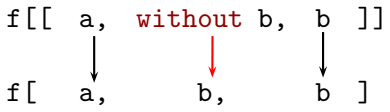
It may seem surprising (at the beginning), that the query term q *matches the data term* d_1*. The query term list* `[[ a, without b, b ]]` *does not mean "match an 'a' so that there is no 'b' afterwards, then match a 'b' ", since the positive term has precedence over the negative. The negative terms may only match data terms that are left over by the positive term matching. Since this query is even ordered, the negative term* `without b` *can only match a data term that is between the data term matched by* `a` *and the data term matched by* `b`*. Since there is no such data term in* d_1*, the query succeeds.*



Thus, the query term $q =$ `f[[a, without b, b]]` *can be read as "match an 'a' and then a 'b', but only in a way that there is no 'b' between both matches."*

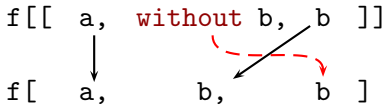
Consider the unification of the query term q *with the data term* d_2*. By now it should be clear that* q *matches with* d_2*. However, the negated query subterm* `without b` *removes one*

mapping possibility. The failing mapping is the following:



This mapping fails because it maps a negative query term to a data term.

However, considering the positive subterms only, the following mapping is possible and not extensible to map the negated subterm `without b` *to any data term, due to monotonicity and injectivity constraints. As stated before,* `without` *must respect the monotonicity and injectivity constraints.*



This mapping is allowed and hence the unification succeeds. If the programmer wanted to exclude d_2*, the query must be* $q' =$ `f[[a, without b, b, without b]]`

This example analyzed the behavior of term negation in ordered query term lists. However, none of the examples above were real examples that made clear what such a semantics is good for. Consider hence the following query:

```
html[[
    body[[
        var Title as h1[[ ]],
        without h1[[ ]],
        var Paragraph as p[[ ]],
    ]]
]]
```

This query retrieves every paragraph with its corresponding title. The negated term `without h1[[ ]]` between the two positive terms guarantees that the variable `var Title` can only be bound to the `h1` element that is immediately preceding the paragraph bound by the variable `var Paragraph`. The semantics of term negation allows hence to query such "flat structures" and create "deep structures" out of them.

Another interesting application of negation is the retrieval of the first or the last title with paragraphs. The query

```
html[[
    body[[
        without h1[[ ]],
        var Title as h1[[ ]],
        without h1[[ ]],
        var Paragraph as p[[ ]],
    ]]
]]
```

retrieves only the first "section" and

```
html[[
    body[[
        var Title as h1[[  ]],
        without h1[[ ]],
        var Paragraph as p[[ ]],
        without h1[[ ]]
    ]]
]]
```

retrieves only the last "section". Such queries are possible because the negated query terms do not have priority, and respect the mapping constraints.

Another interesting programming issue are successive negated terms in ordered query terms. Due to the defined semantics above, the query fails if any of the negated query terms succeeds in finding a match. This is due to the fact that all possible mapping extensions to a given positive mapping are build into one conjunction.

However, the evaluation of successive negated terms in ordered query terms may be counter-intuitive at first, and is currently under further investigation.

Example 8 (Successive Negated Terms in Ordered Queries) *Assume the query term* $q =$ `f[[ a, without b, without c, d]]` *and the data term* $d_1 =$ `f[a, b, c, d]`, $d_2 =$ `f[a, c, b, d]`.

The query term q *fails to match* d_1*, because* d_1 *contains a data term* `b` *and a data term* `c` *between the only candidates for matching the queries* `a` *and* `d`.

Furthermore, the query term q *fails to match* d_2 *for the same reasons. Although the data terms* `b` *and* `c` *are not in the same order as in the query, there is a partial mapping extension on the negative query term* `without c` *that maps it to the data term* `c`*. Due to this mapping, the unification of the query term* q *and the data term* d_2 *fails.*

The described phenomenon above is due to a conflict of the negative terms based on monotonicity. It can be impossible to match all negative query terms at once due to the mapping constraints. Since the query already fails when only one negated query terms finds a successful match, the conflicts on negated terms are not very visible to the programmer. Such kind of conflicts become visible however when optional terms are involved instead of negated terms, as considered in the following.

Optional terms can conflict with with positive or other optional subterms due to the injectivity mapping constraint, the monotonicity mapping constraint or due to conflicting variable bindings. Between positive and optional subterms, the priority is very clear. Whenever an optional subterm is conflicting with a positive subterm due to one of the three reasons, the valid solution is obviously the one matching the positive term and dropping the optional term, as the following examples show:

1. Injectivity: `f{ optional a, a }` $\preceq_{su}$ `f[ a ]`
   The positive subterm has priority, so that the query succeeds.

2. Monotonicity: `f[[ optional a, b ]]` $\preceq_{su}$ `f[ b, a ]`
   Again, the positive subterm has priority so that the optional subterm must be omitted in order to find the right solution.

3. Variables: `f[[ optional var X, var X]]` $\preceq_{su}$ `f[ a, b ]` The variable binding generated by the optional subterm conflicts with the positive variable binding, so that omitting the optional subterm creates the right solution.

The case of conflicting variables is the most complicated to handle, and is the one that prohibits a general optimized treatment of optional subterms in partial specifications.

In the case of total subterm specifications with optional subterms, the fact that variables may be conflicting can be ignored. Taking the example above of conflicting variables and making the query total lead to the following simulation problem: `f[ optional var X, var X]` $\preceq_{su}$ `f[ a, b ]`. Because the data term has two subterms, both query subterms must match regardless of possible variable conflicts (or other reasons such as label mismatch). If taking both query terms fails, the whole query fails because leaving out the optional subterm violates the surjectivity requirement of total queries. For this reason, the evaluation of optional terms in total queries can be optimized drastically (see section 4.3.1).

Consider the priority between optional and negative subterms. The priority is not defined clearly in the constraint calculus, as the calculus rules requires to apply query normalization before it applies. To determine the priority between both kinds of subterms, consider the query `f{{ optional a, without a }}` and the data term `f{ a }`. The normalization of the query above gives

```
or{
    f{{ a, without a }},
    f{{ without a, without a}}
}
```

This query succeeds with the given data term, which means that the optional term has priority over the negative. The same holds for the monotonicity restriction and for conflicting variables as becomes clear by applying the query normalization to some sample cases.

Furthermore, optional terms can conflict with each other due to the three criteria mentioned above. For example, the variable bindings are conflicting in the following query

```
f[
    g[[ optional var X ]],
    g[[ optional var X ]],
    g[[ optional var X ]]
]
```

if the query above is matched with the data term

```
f[
    g[ a ],
    g[ b ],
    g[ b ]
]
```

Since the bindings are optional (and not mandatory), there are two results: $X \preceq a$ and $X \preceq b$. The rationale for this is that enabling all optional subterms leads to the solution $X \preceq a \wedge X \preceq b$, which is inconsistent. The maximal solution does not exist. The next possible solutions are then

1. $X \preceq a \wedge X \preceq b$ (first and second optional terms considered),
2. $X \preceq a \wedge X \preceq b$ (first and third optional terms considered) and
3. $X \preceq b \wedge X \preceq b$ (second and third optional terms considered).

The solutions one and two are again inconsistent, so that the next solutions are

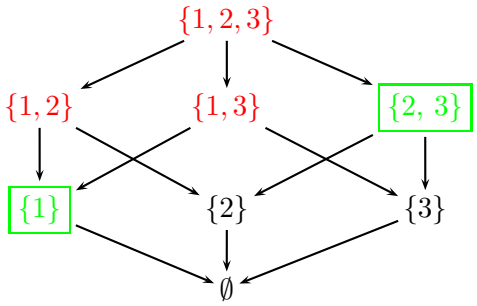
1. $X \preceq a$ (first optional term considered),
2. $X \preceq b$ (second optional term considered) and
3. $X \preceq b$ (third optional term considered).

Solutions two and three are subsumed by the solution $X \preceq b \wedge X \preceq b$ and are hence not maximal. This leaves solution one as maximal solution. The maximization of the solutions yields the two solutions $X \preceq a$ (first optional term enabled) and $X \preceq b$ (second and third optional terms enabled) as result. The solutions are maximal by the subset order defined below.

Definition 7 (Subset Order) *Let S be a set. A subset order is a partial order on the set $\{s \mid s \subseteq S\}$ with the top element S and the bottom element $\emptyset$. The order relation is the subset relation, $\subseteq$.*

Each optional term is assigned a number. The maximal number sets denote the combinations of optional enabling/disabling that generate solutions. Each combination however does not denote one generated solution, but only the combination of optional terms that generate one or more solution that will be available for the construct term grounding. It is crucial to consider only maximal combinations of optional enabling/disabling to prohibit the creation of redundant terms when grounding rule heads. As mentioned in the Xcerpt syntax section, the optional construct specifies that it must be matched if it is possible to do so, and may only be dropped if no valid match exists. This is the reason for the restriction on maximal optional sets.

For the above query, the set of numbers to maximize is $S = \{1, 2, 3\}$, each number representing an enabled optional subterm. The order of S is as follows.



The inconsistent combinations are marked red, and the maximal consistent solutions are green and boxed. The maximal consistent solutions subsume all black marked solutions, which do not generate solutions.

The example above demonstrated that optional terms can prohibit other optional terms to match even in the case that they are not children of the same node. This is an issue concerning optimization, since this is a nonlocal interdependency that must be handled appropriately.

This interdependency could be avoided if the semantics of optional terms were changed. If the sharing of purely optional variables[8] among optional terms were forbidden or if each optional term defined its own scope, this form of interdependency would not occur.

However, there are other possible (local) interdependencies of optional query parts based on monotonicity and injectivity mapping constraints. Assume for example the query `f{ optional a, optional a }` matching the data term `f[ a ]`. It is impossible to match both optional terms to the one `a` due to the injectivity constraint on the paths. Again, both optional query terms interfere, and the maximal solution does not exist. In fact, there are two possible solutions to map this query term to the given data term: one by enabling the first optional term and disabling the second (represented by the set {1}), and the second solution is to disable the first and to take the second (set: {2}). This may be surprising, but optional queries are not linked together by any other condition than by injectivity and monotonicity.[9]

Interference of optional query terms may also occur due to the monotonicity constraint. For example, if the query term `f[[ optional a, optional b ]]` is unified with the data term `f[ b, a ]`, the optional terms interfere. There is no possibility to match both optional terms, but there is the possibility to match each one alone.

In this section, the complex behavior of negation and optionality was investigated. Altogether, there are seven rules to follow when writing and reading Xcerpt programs with optionality and negation:

1. A query is only successful if none of the negated query terms in the query can be matched.

2. Query terms may conflict due to injectivity, monotonicity and variable bindings. If they do, there are precedences between the query term types that hold.

[8] By this, variables are meant that occur only within the scope of optional terms and never in a positive term.

[9] If it were possible to specify a dependency between both as "try to match the second optional query part only if the first one matched" as in `f[optional(a, optional a)]`, there would be only one match of the query in the data.

3. Positive query terms have precedence over optional and negated query terms.
4. Optional query terms have precedence over negated query terms.
5. If two positive terms conflict, the mapping containing the conflict fails (or is never generated).
6. If two optional terms conflict, several "sub-maximal" solutions are generated, each one maximal in the set of possible solutions according to subset order.
7. Negated query terms never conflict with each other on variable bindings, but only on monotonicity and injectivity. These conflicts become visible as apparent violation of the mapping constraints between the negated query terms.

With these rules in mind, the following sections that introduce an algebra for simulation unification (section 3) and optimizations for simulation unification (section 4) may become clearer and easier to understand.

After this presentation of Xcerpt, an algebra for Xcerpt's simulation unification is introduced and investigated in the following.

3 Algebra for Simulation Unification

The algebra that is presented in this section offers a formalization of simulation unification different to the constraint calculus presented in the previous section. Algebras are popular tools in language research and query optimizations. There are several already existing algebras for different programming languages, even in the field of semistructured data querying.

To compare the algebra with the constraint calculus, the query algebra focuses particularly on the decomposition rules. The generation of the mapping from the query to the construct terms for a given query term is decomposed into smaller steps. Furthermore, the algebra proposes a more deterministic formalization of the basic constraint calculus, since the order of the operators define an order of the evaluation process. However, the algebra defines only the data flow. The definitions of the operators do not define the exact control flow of the evaluation. Although the algebra is defined as a set based algebra, a tuple based implementation of the algebra is conceivable.

The structure of the section is as follows. The first section discusses existing algebras (section 3.1). After this short overview of related work, the design decisions made in the Xcerpt query algebra are presented in section 3.2. The Xcerpt query algebra domain and operators are defined in section 3.3, starting with a small set of operators. This operator set is extended step by step to cover attributes, variables, negation, and optional terms. The discussion of the algebra concludes with the definition of rewriting rules in section 3.4 and further extension possibilities in section 3.5. Section 4 demonstrates complexity results for optimizations of the algebra and additional optimizations not incorporated into the algebra.

3.1 Existing Algebras for Semistructured Data and XML

Both scientific and industry research developed algebras for XQuery or XML (one could call this a formal model for XML access) with the purpose of query optimization and formalization.

In the following, some well known algebras in the field of XML querying is discussed. These are XAL [FHP02] (Xml ALgebra), TAX [JLST01] (Tree Algebra for Xml), XAT [ZPR02] (Xml Algebra Tree), OPAL [Lie99] (Ordered list Processing ALgebra), an algebra for XQuery [FSW00] and the elegant but unnamed algebra that Beech et al. propose in [BMR99]. Most of the XML Algebra research concentrates on the support of XQuery and aim to lean on relational algebra or to extend the relational algebra to cover XML. The goal of such an approach is to bring 35 years of algebra optimization expertise to the XML world. How well this worked out might still be subject to controversial discussion.

Since Xcerpt has a different approach to XML querying than most algebras support and has furthermore special constructs such as `without` and `optional`, the algebra for Xcerpt needs to be custom made. Nevertheless, the various algebras for XQuery and XML provided precious input and ideas for the design and development of the Xcerpt query algebra described in section 3.3.

3.1.1 XML Query Algebra

Beech et al. [BMR99] present an XML query algebra . Most notably, the algebra is not tailored to support XQuery or any other query language, but is designed as general purpose query algebra for XML querying.

Their data model consists of graphs with vertices having properties (such as *parent*, *childelements*, *attributes*, and *references*) and typed, named edges. Since the algebra is XML-centric, it supports comments and processing instructions and furthermore treats IDREF-edges as being different from parent-child-edges. The edge types allow to distinguish between children, attributes and referenced elements of a node. Special edge names denote comments, processing instructions and text data.

The algebra uses collections of edges as domain instead of nodes, because edges carry type and label information. The operators take several edges as input and return several edges as output.

The most important operator is the navigation operator $\phi[edgetype, name]$ that navigates from an input edge to an edge having the specified type and name. The edge type may also be a disjunction of types (e.g. $E \mid A$ specifies that the target edge may be an element or an attribute edge). The name of the edge can be specified as regular expression.

To allow descendant navigation, the algebra includes the kleene star operator $*[f(x)](x : expression)$. It applies the function f to the expression bound to x until a fixpoint is reached. f may itself be an algebraic expression. The descendant operator, for example, would be $descendant[edgetype, name](S) = \phi[edgetype, name](*[\phi[E|R, *]](S))$ (edges typed with R are reference edges).

The next basic operator is the selection operator $\sigma[condition(e)](e : expression)$. The condition of the selection operator may use navigation operators, node properties, value comparisons and boolean connectives. Additionally, existential and universal quantification are allowed in conditions. As in XPath, the default quantification for comparison operators is existential (i.e. the comparison θ on a set of values S with a value v is successful if a value v' exists in S such that $v' \,\theta\, v$ holds).

Another fundamental operator is the join operator. This operator introduces new reference

edges into the input graphs. For every node pair satisfying the join condition, the operator introduces a special temporary "join edge". Further selection and navigation operators can use this temporary edge, while data serialization is advised to ignore such edges.

Further useful operators are the map operator $\mu[f(x)](x : expression)$, applying the function f to all elements of the input list, and the expose operator $\epsilon[edge_1(a), \ldots, edge_n(a)](a : expression)$ that exposes several edges of the input collection allowing several navigations at once.

Note that the investigation above focused on the query operators of the algebra only. The algebra offers several constructor operators (node and edge creation, sorting and grouping) that are not discussed here.

This XML algebra lacks direct support for the injectivity restriction on queries. That is to say, whenever a query language want to support query injectivity, the translation into this algebra becomes difficult. However, it provides means to handle ordered and unordered queries and data by an "unorder" operator.

3.1.2 XQuery Algebra from Fernandez et al.

Fernandez et al. propose an algebra for XQuery in [FSW00]. Their work is aimed towards an algebraic formalization of XQuery. As a consequence, the operators of this XQuery Algebra resemble XQuery itself. The presented algebra offers basically four kinds of operators: projection, selection, iteration and element construction. The domain of the XQuery algebra is a duplicate preserving collection of semistructured data (the algebra however provides a distinct operator to remove duplicates). The semistructured data model the algebra uses is a node labeled model with ordered content (note that Xcerpt also uses a node labeled model of semistructured data whereas [ABS00] use a edge labeled model just as the algebra above). Data terms are accessed through variables, that is, the algebra allows to introduce variables in let-expressions of the kind

```
<expression> ::= "let" <variable> ":" <type> "=" <data>
```

The algebra operators can then refer to the data through the variable names.

The projections of this XQuery algebra are path navigations using XPath 1.0 syntax. A projection is for example

```
book/addressbook/person
```

This projection maps the data labeled with the variable `book` to the set of person nodes. An EBNF rule for projection would hence be

```
<projection> ::= <expression> "/" label
```

The iteration operator is the for-loop of XQuery. Its syntax is

```
<iteration> ::= "for" <variable> "in" <expression> "do" <expression>
```

As common practice in XQuery, multi-step XPath navigations are decomposed into nested iteration with one-step navigations.

The selection operator is expressed by a where-clause that specify conditions on the data terms.

```
<selection> ::= "where" <expression> "do" <expression>
```

However, the where-clauses themselves are syntactic sugar for if-then-else expressions.

$$\text{"where"}\ \langle\text{expression}\rangle_1\ \text{"do"}\ \langle\text{expression}\rangle_2$$
$$\equiv$$
$$\text{"if"}\ \langle\text{expression}\rangle_1\ \text{"then"}\ \langle\text{expression}\rangle_2\ \text{"else"}\ \text{"()"}$$

The selection conditions are expressed within the if-clause of the if-then-else expression. Transforming selections in this way is only possible within iterations, since the selection condition can be tested on each element of the input set by iterating over the set.

The element construction operator takes a label which becomes the label of the new constructed element and an algebra expression which creates the content of the element (remember that the value of an algebra expression is a collection of data terms).

```
<construction> ::= label "[" <expression> "]"
```

Above these basic operators, the algebra provides aggregation functions (avg, count, min, max, sum) and allows modularization by function declaration. Joins are specified in the algebra just as in XQuery.

The algebra has a very imperative character due to the kind of operators. One very interesting aspect of the algebra is that it offers a type system to type algebraic expressions (i.e. XQuery expressions). Above this, the algebra offers ten rewriting rules for query optimization (see [FSW00] for details on the optimization rules).

Just as the previous algebras, this XQuery algebra variant does not guarantee the injectivity of queries by default. This XML algebra is close to the XQuery formal semantics [DFF+05], which by itself could be seen as a further "algebra".

3.1.3 The NATIX Algebra

Brantner et al. define an algebra for XPath in [BHKM05], which is a first step towards an XQuery algebra.

The algebra uses sequences of tuples and atomic values (string, boolean, integer) as algebra domain. The values a tuple contains can be retrieved by referring to tuple attribute names. Tuples can be extended to contain a new attribute and restricted to a set of tuple attributes. Each tuple contains an attribute *cn* with the actual context node as value. The tuples are extended with additional attributes as needed in the algebra. This XPath algebra has therefore a flexible domain rather than a fixed one.

The algebra offers several operators, of which the operators map, unnesting, selection and dependency join are used the most in the translation of XPath expressions. The selection

operator $\sigma_p(e)$ removes all tuples from the sequence e that do not satisfy the condition p. This is a canonical selection operator.

The map operator $\chi_{a:e_2}(e_1)$ applies an algebraic function e_2 to each element of the input list e_1, and extends each element of the input list with an attribute a containing the result of the application.

The unnest operator $\mu_g(e)$ distributes the sequence value of the attribute g over each tuple. This means that given a tuple t with a sequence-valued attribute g, the unnest operator creates a tuple for each value in the sequence of g consisting of t and the respective element of the sequence of the attribute g. That is to say, the unnest operator flattens the attribute g into the top level sequence.

The dependency join operator $e_1\langle e_2\rangle$ joins the sequence e_1 and the algebraic function e_2 together. The tuples of e_1 are used as input values to the function e_2. That is to say, the attribute names of the tuples in e_1 are used as variable names, and the attribute values as variable values. The dependency join is used to enumerate a set of context nodes e_1 and execute an algebraic subexpression e_2 for each context node. This is the way the canonical translation operates.

The improved evaluation however uses the map and unnest operators to evaluate the XPath operators in a streamed way without enumerating the contexts to the subexpressions. The context nodes satisfying the already evaluated XPath prefix are passed as input to the algebraic expression of the XPath suffix that is to evaluate.

The interesting aspect of the NATIX XPath algebra is that the algebra domain is not fixed at all. Each operator performs a manipulation of the algebra domain type and requires certain attributes to be present for each tuple of the input tuple list. Such a "malleable" algebra domain allows to decompose the query evaluation into several very small operators. Such a detailed algebra could be another lower level algebra to define for Xcerpt. In this thesis, however, we chose a higher level algebra for Xcerpt than this very detailed NATIX algebra.

3.1.4 The Algebra XAL

XAL [FHP02] uses collections of graphs with typed and named edges as data model as [BMR99] and [ABS00]. The edge types are Attribute, Element and Idref, and the edge names represent XML labels. The operators of the algebra are derived from the relational algebra operators.

The general notation for an operator of the XAL algebra is $op[f(x)](x_1, \ldots, x_n)$. The expressions $x_1, \ldots, x_n$ are algebraic expressions that define how the input to the operator *op* is computed. The function f is itself a function from the algebra domain into the algebra domain. The operator *op* may use this function f as appropriate to compute the result of the operator. However, not every operator is parameterized in this way. Some operators are lacking such a function parameter f.

The projection operator $\pi[type, name](e : expression)$ for example does not have a function parameter. The projection operator however allows regular expressions as specifiers for edge names. Above this, the projection operator requires to specify the type of the navigated edge.

The selection operator $\sigma[condition](e : expression)$ takes a boolean condition that may con-

tain equations with constant values, node values or results from projection operators, as for example $\sigma[\pi[A, Name] = \text{“}Bry\text{”}](e : expression)$.

The cartesian product operator (and the join operator) creates a new graph node with two child nodes for each input pair, the first one being an item of the first input collection, the second child node being an item of the second input collection.

XAL also defines set operators (union, intersection and difference) and construction operators for edges and nodes, allowing to regroup decomposed data. Furthermore, the XAL algebra offers the map, kleene star and unorder operator known from the XML query algebra, and some more that might be relevant for special cases of semistructured data querying (distinct and sort operators e.g.).

The optimization approach of the algebra XAL resembles the optimization rules of the relational algebra, since the equivalence rules for XAL resemble those valid in the relational algebra (i.e. selection commutativity, projection and selection commutativity, selection and cartesian product commutativity).

The same XML data must be input several times to perform a query (just as in the relational algebra), since XAL offers no means to introduce variables for intermediate results or algebraic expressions.

Just as the previous algebras, the XAL algebra does not consider injectivity while querying children of a node. However, with the unique identifiers on each node that the XAL data model provides, it is possible to implement the injectivity requirement in the XAL algebra.

3.1.5 The Algebra TAX

TAX [JLST01] uses an interesting approach, departing widely from the relational and previous algebra. The TAX query operators are selection, projection, product, join, grouping and aggregation. The data model of TAX is again a collection of trees, however this time with unlabeled edges. The nodes contain all information, especially information about the type, label and value.

To retrieve the relevant parts of the data trees, TAX introduces pattern trees, a notion similar to the query terms of Xcerpt: A pattern tree is a graph with numbered nodes and edges labeled with “pc” for parent-child relationship and “ad” for ancestor-descendant relationship. Furthermore, pattern trees contain a formula that specify additional conditions on the pattern nodes, e.g. $\$1.tag = attribute \ \& \ \$1.name = \text{”} * xml * \text{”}$ specifies that node 1 must be an attribute (the type information being stored in the property “tag”) with a name containing the sequence “xml”. The conditions allow comparison with regular expressions, equality and unequality conditions as well as order conditions with the relation “BEFORE”. The formula can contain conjunctions and disjunctions. It is hence possible to express the injectivity of mappings with the formula

$$(\$1 \, BEFORE \, \$2) \ || \ (\$2 \, BEFORE \, \$1)$$

Adding this formula for every two siblings would then lead to an injective pattern tree matching.

The information about how a pattern tree matches a data tree is given as a set of witness trees. A witness tree is an extract of a data tree that matched the pattern tree’s structure. A

witness tree contains only such parts of a data tree having a correspondent node in the pattern tree. As in Xcerpt, matching a pattern tree with a data tree may yield several embedding of the pattern tree in the data tree. For each such embedding, the matching returns a witness tree.

Almost all operators use pattern trees to specify the processed data. The selection operator $\sigma_{\mathcal{P},SL}$ for example returns the witness trees for the pattern tree $\mathcal{P}$, possibly extended by subtrees of nodes occurring in the witness trees and in the additional parameter of the selection operator, the "adornment list" *SL*.

The projection operator $\pi_{\mathcal{P},PL}$ uses a pattern tree together with a projection list; the operator returns only the witness trees nodes that occur in the projection list. The projection operator also preserves descendants of such nodes if the nodes are annotated by "*" in the projection list *PL*.

Similar to the algebra XAL, the cartesian product and join operator constructs a new root nodes for each pair of the two input sets. In order to avoid that these artificial nodes occur in the output, appropriate projection operators must be performed.

The algebra TAX also supports in place modification of trees. It offers operators for node deletion, insertion besides operators for copy-paste. In fact, TAX offers a rich set of operators. Most notably, TAX offers grouping and aggregation operators comparable to the all/some constructs of Xcerpt as well as aggregation functions.

3.1.6 The Algebra XAT

XAT is an algebra extending the relational algebra in a different way than XAL did: XAT uses a table in NFNF (non-first-normal-form) as data model. All operators operate on a such XAT tables.

There are several defined operators (28 to be precise, see [ZPR02] for the complete list of operators) who manipulate the XAT relation by removing, modifying or adding columns based on the content of other columns. Each cell of a table row[10] may contain an atomic value, an XML fragment or a collection of both.

The most important operators for querying are (besides the classical selection and projection operator that are a canonical extension of the relational operators on the NFNF) the navigation operator $\psi^{col'}_{col,path}$ and the FOR operator $FOR_{col}(s, sq)$. The navigation operator takes an XPath expression *path*, an input column name *col* and an output column name col' to write the result of the XPath expression applied to the XML fragment(s) of the input column in the output column.

The FOR-operator is the correspondent operator of the XQuery FOR-clauses. The realization of the operator iterates over a column *col* of a table generated by an other expression *s*. The subquery *sq* is executed for each entry of the column. The operator uses here column names as variable names and columns as variable values to pass the data resulting from one subquery to the other. The result of the FOR-operator is the accumulated result of the subquery evaluation.

[10]Formally, the statement must be rephrased as "each attribute of an element of the relation", but the XAT algebra uses the notion of tables, columns and cells very extensively, so that the terminology was adopted in this presentation of XAT.

XAT offers a rich set of construction operators (especially aggregation, grouping and sorting), most notably the "composer operator". This operator builds an XML document from a table representing parent child relationships through id columns. Further columns specify the attribute names and content, as well as element labels and values.

The XAT algebra is a weakly typed algebra: All operators require tables with certain columns or of a certain structure to work properly. At the same time, XAT has no typing system to infer the types of an algebraic expression before its execution.

3.1.7 The OPAL Algebra

The OPAL Algebra uses a functional programming style approach, since the basic concepts of the algebra are list comprehension and functional pattern matching. As a consequence, the OPAL optimization techniques are derived from list comprehension syntax rewriting rules.

The data model of OPAL uses nested and named lists to represent lists of terms. Data constructors are for example single values, empty lists, singleton lists and list concatenations:

$$value \mid [] \mid [label : list] \mid listexpr_1 + listexpr_2.$$

The list comprehension operators are also list expressions: for list iteration, OPAL introduces

$$[e_2(l, a, v, r) \mid \langle l + a : v + r\rangle \leftarrow e_1] \text{ and } \exists\langle l + a : v + r\rangle \in e : c(l, a, v, r)$$

for existentially quantified conditions on lists. The expression e_2 in the list iteration expression uses the variables l, a, v, r bound by matching the pattern $\langle l + a : v + r\rangle$ with the result of the list expression e_1. Additionally, OPAL allows to use variables at the level of the query algebra to represent list values. These variables are used to represent variables in the initial query and also to store intermediate query results that are used several times.

The supported query language of OPAL is pattern based as Xcerpt and seems to be inspired by XML-QL [DFF+98]. The basic structure of a query is similar to the SQL select-from-where structure, but it uses tag minimization, and data terms external to the program are specified as data sources as in XML-QL (The following example stems from [Lie99]).

```
select <Publ> <CTitle> $c </> <Year> $y </> </>
from   <Publ> <Conf>   $c </> <Year> $y </> </> in db
```

All OPAL query patterns are ordered and partial. Negation is possible at the level of variables. This is different to the negation Xcerpt supports with the `without` construct. OPAL supports exclusion patterns on variables by translating them into negated existential condition expressions (e.g. `not exists(<Name></> in $x)` will be translated into $\neg\exists\langle l + a : v + r\rangle \in x : a = Name$).

OPALs algebra expressions again do not support the injectivity of the sibling matchings, since OPAL processes several subqueries by binding the data term list to a variable and traversing the list with a list iteration expression for each subquery. Here, the later iterations do not consider the chosen terms of previous iterations. How injectivity could be specified is difficult to imagine for the OPAL algebra. However, the algebra demonstrates a different and interesting approach to algebraic XML querying.

The approaches presented in this section depart widely from each other in some cases. There are high level algebras that define operators from a mathematical point of view (TAX, XML Query Algebra, NATIX algebra, XAL and OPAL). On the other hand, there are algebras that seem very language or even implementation oriented (such as XAT and the presented XQuery algebra).

The XML Query Algebra and the XAL algebra build a group of mathematical, high level logical algebras with similar operators. They both define selection and projection operators in a similar way. The other operators such as the map operator and kleene star operator are also present in both algebras.

The OPAL algebra cannot be classified with the other algebra because of two reasons. First, in supports an own query language which is similar to XML-QL. Second, it is a very functional style algebra which use list comprehension and variables in the algebraic expressions themselves.

The NATIX XPath algebra and the TAX algebra are similar in some regards. First, the algebraic domain is "malleable", that is to say, the algebraic domain is not fixed to a certain type. Operators may modify the domain by adding or removing certain data from the domain. However, the NATIX algebra uses a more mathematical approach and aims at optimizations of XPath while the TAX algebra is an algebra for XQuery and SQL.

Classifying the TAX algebra is difficult, because it uses pattern trees. In a way, the pattern trees hide the complexity of querying from the algebra.

With this background of existing algebras, the definition of an algebra for Xcerpt querying is given in the next section. We choose to define a high level algebra that is language dependant in order to support the special query constructs of Xcerpt in the algebra.

3.2 Design Principles of the Xcerpt Algebra

The Xcerpt algebra must support all query term features of Xcerpt, especially negated and optional terms. However, we restrict the algebra in two ways. First, the Xcerpt algebra does not support *cyclic* query terms. The introduction of cycles would lead to cyclic algebraic expressions, which is very counterintuitive at first.[11] Second, the Xcerpt query algebra cannot query construct terms, but only data terms. We do so because the grouping constructs make it necessary to lift the mapping injectivity requirement for certain data terms and the occurrence of variables leads to the occurrence of algebraic expressions in the algebra domain, which again would lead to more complex algebraic operators. Construct terms were left aside to keep the algebra simpler in this regard (more details on these restrictions are discussed in section 3.5). To summarize, the following algebra is restricted to the unification of acyclic query terms with possibly cyclic data terms.

We decided furthermore to process the data terms in the order that the query term expression is built up, that is as recursive descent. The algebra expression first verifies if all constraints imposed by the root of the query term are satisfied, and processes the children afterwards until the leafs of the query term tree are reached. Using a different approach, such as starting with the leafs and performing structural joins until the root is reached, makes data indices to locate the children or several input of the processed data possible and necessary. With

[11] This restriction does not apply to data terms.

the recursive descent approach, the algebra does not depend on data indices and can process semistructured data without further meta information.

A basic idea of the algebra is that querying in Xcerpt is a process of selecting data terms and possibly (and most probably) building sets of variable bindings. For example, the query `html[[ var X ]]` selects all data terms with a root label `html` and ordered content, and binds the children of the root to the variable X.

An Xcerpt query term imposes two kinds of selection criteria in the process of querying: value selections and structural selections. Value selections are selections based on properties of the data node currently processed, such as node label, namespace and attributes. Structural selections are selections that require the verification of selection conditions on data nodes accessible from the current data node, such as children or descendants. To these children or descendants, the selection criteria are again dividable into value selections and structural selections.

Example 9 *Consider the query* `html[ head[[ ]], body[[ desc p[ ] ]] ]`
The query selects HTML documents with a head and at least one paragraph somewhere in the body.

The query imposes value selection criteria on the root node of any data term to match, which are

1. *Its label must be* `html`.
2. *The root node must have two children.*[12]
3. *Its children must be ordered.*[13]

The query also imposes structural selection criteria on the root node:

1. *The label of the first child must be* `head`.
2. *The children of the first child must be ordered.*
3. *The label of the second child must be* `body`.
4. *The children of the second child must be ordered.*
5. *A descendant of the second child must have the label* `p`.

Since the structural selection criteria may impose selection criteria on more than one reachable node, there must be a way to perform several navigations from one context node. Within the algebra, this is achieved by operators that never change the context node. However, special operators, called "map operators", are introduced. These operators apply an algebraic subexpression to the child, attribute or descendant context nodes. These operators change the context internally, whereas the context nodes again never change during the processing of the algebraic subexpression. Such a map operator is basically a variant of the higher order operator "map" from functional programming. It uses an algebraic subexpression and applies

[12] Here, we assume that each node knows the number of children it contains.
[13] Again, we assume that each node knows whether its children content model is ordered or unordered.

it to a certain set of nodes. The map operator however do not only return the results, but combine these results with the input context and add information (about bound variables or mapping paths) to their own context. However, as stated before, they never change the node of context.

Another observation is that value selections are cheaper than structural selections. The navigation operators increase the set of context nodes to consider, since there is usually more than one child or descendant reachable from a context node. In order to reduce the size of intermediate results, it is hence sensible to reduce the input size of a navigation operator as far as possible by value selections before considering the structural selection criteria.

A further interesting point for the algebra are severalfold occurrences of a variable within a query (query term or even query formula). Queries with such properties express joins, because the bindings generated by each occurrence of the variable must be checked for consistency. Just as in relational algebra, a good ordering and early execution of joins (for Xcerpt: performing consistency checks on variable bindings) can reduce the size of intermediate results considerably. For this reason, every operator that may introduce inconsistency by introducing a variable binding or potentially combine several variable bindings performs a consistency verification on the variable bindings. These are the operators that introduce the variable bindings, and also the map operators, as they combine the subquery results with the results from previous variable bindings or map operators.

This section discussed that Xcerpt queries impose two kinds of selection criteria. There are local value selections and structural selection criteria. The distinction between both is justified by their cost: local value selections are very cheap, while structural selections may be arbitrarily expensive. Local value selection is handled with a selection operator, while the structural selection criteria are handled by so-called map operators. A map operator is an operator that applies a algebraic subexpression to a set of reachable context nodes and combines the result of this application with the results of the previous selection results.

3.3 Definition of the Algebra

Similar to the relational algebra, the query algebra is set based, and each set element is a possible solution. Operators of both algebras, the relational algebra and the Xcerpt query algebra, perform selections and transformations on a given set of values. However, there are some differences between both algebras. While the relational algebra produces the outcome of a relational query, the query algebra produces bindings for variables of a given query. The evaluation uses these bindings to generate ground instances of the rule head, but the bindings themselves are never output. Another difference between the Xcerpt query and relational algebra is that the Xcerpt query algebra inputs the data only once, while the relational algebra use the same input relation several times. Furthermore, the Xcerpt query algebra contains only unary operators (but on a complex domain) while the relational algebra consists of binary and unary operators. The restriction on unary operators is a way to propagate query constraints and verify them incrementally, as the next sections demonstrate. The query constraints that are verified in this way are the injectivity and monotonicity constraints on the mapping of siblings.[14]

[14]It is possible however to defer the verification of mapping properties and execute them in join operators. This approach was not considered yet, as it does not fit well with total and ordered query term lists. This

3.3.1 Preliminaries

The definitions in the following sections mostly define functions on sets. The reader will often see definitions like

$$S = \{s \mid P_1(s), \ldots, P_n(s)\}$$

Which can be read as "S is the set of all s with the property that $P_1(s)$, ... and $P_n(s)$ holds". This is one basic structure in the following definitions.

The following sections also use types, since an algebra domain is introduced. A type is understood as a set of elements. An object is hence typed by being element of a type: $object \in Type$. To distinguish a typing statement from a set containment statement, the notation $object : Type$ is used. This notation is inspired from functional programming.

The following definitions also use lists besides sets. The notation for lists of T is $List(T)$, the empty list is *nil* and the direct access to a list is written as $list[index]$. The containment symbol for lists is the same as for sets, $item \in list$. Furthermore, the function $size : List(T) \rightarrow Integer$ is used in some of the following definitions. To keep a consistent notation, the type of sets of T is denoted as $Set(T)$.

The reader will also encounter functions in curried form, that is

$$f(a_1, \ldots, a_n)(b_1, \ldots, b_m).$$

This notation means that $f' = f(a_1, \ldots, a_n)$ is a function of $(b_1, \ldots, b_m)$. $f'(b_1, \ldots, b_m)$ is then equivalent to $f(a_1, \ldots, a_n)(b_1, \ldots, b_m)$.

To decompose severalfold set iterations into smaller understandable steps, the definitions use two higher order functions (as known e.g. from functional programming).

Definition 8 (Map and Iterate) *The map function is*

$$\begin{aligned} map &: (T_1 \rightarrow T_2) \rightarrow Set(T_2) \rightarrow Set(T_2) \\ map\ f\ S &= \{f(s) \mid s \in S\} \end{aligned}$$

The iterate function is

$$\begin{aligned} iterate &: (T_1 \times T_2 \rightarrow T_2) \rightarrow Set(T_1) \rightarrow T_2 \rightarrow T_2 \\ iterate\ f\ st\ S &= \begin{cases} st & if\ S = \emptyset \\ iterate\ f\ (f(st, s))\ S' & if\ S = S' \cup \{s\} \end{cases} \end{aligned}$$

The $map\ f\ S$ function is straightforward. It applies the function f to all elements of S.

The reader might know the $iterate\ f\ st\ S$ function from functional programming as foldl or foldr. The iterate function takes a function f, a start value st and a set S. If S is empty, *iterate* returns the start value st. If it is non-empty, *iterate* chooses s from the set S and calls *iterate* recursively with $f(s, st)$ as new start value and remove s from the set S. This case applies until the empty set is reached. Then, *iterate* returns the start value st as result from *iterate*, that is, the accumulated value. Note that the function f does not rely on any iteration order since sets do not provide any element order.

would be of course an interesting variation of the Xcerpt algebra.

To handle mappings of query terms to data terms, paths are extended while incrementally verifying path conditions. For this reason, two functions on binary relations, *dom* and *range*, are introduced for the domain and the range of a relation.

Definition 9 (Domain and Range) *Let* $Rel : Set(A \times B)$ *be a relation over two sets* A *and* B*.*

$$\begin{aligned} dom(Rel) &= \{a \mid \exists b \in B. \quad (a,b) \in Rel\} \\ range(Rel) &= \{b \mid \exists a \in A. \quad (a,b) \in Rel\} \end{aligned}$$

Furthermore, the function notation is used for binary relations $Rel : Set(A \times B)$, i.e. $Rel(a)$ assuming that the relation is a mapping:

$$Rel(a) = \begin{cases} b & \text{if } \exists (a,b) \in Rel \text{ and } \forall b' \in B. \quad (a,b') \in Rel \Rightarrow b = b' \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The conventions presented in this section are used throughout the following definitions of the Xcerpt query algebra.

3.3.2 Basic Operators and Algebra Domain

To begin with the definition of the algebra, the basic algebra operators and the algebra domain are introduced. Further operators are added in the next sections.

Name	**Notation**	**Short description**
Selection	$\sigma(fun\ \theta\ value)$	Selects solutions based on the actual context.
Child	$Child(i, f_1, \ldots, f_n)[f_{sub}]$	Selects all children fulfilling f_{sub}.
Descendant	$Desc[f_{sub}]$	Selects all descendants fulfilling f_{sub}.

Table 1: The Basic Operators of the Query Algebra

The selection operator $\sigma(fun\ \theta\ value)$ for example removes all solutions in the solution set that do not fulfill the condition of selection. It corresponds to the relational selection operator that removes the tuples from the solution relation which do not fulfill the condition of selection. The child and descendant operators are map operators. They apply the algebraic subexpression f_{sub} to the set of children (and descendants, respectively) and combine the results from the subexpression with the input solution set.

Example 10 (Algebra Expression) *With these operators, the query* `html[[ head[[ ]], body[[ desc p[[ ]] ]] ]]` *is translated into the following algebraic expression.*

$$\begin{aligned} &\sigma(type = \text{“}tree\text{”}) \\ &\sigma(label = \text{“}\texttt{html}\text{”}) \\ &\sigma(arity \geq 2) \\ &\sigma(ordered = True) \end{aligned}$$

$$
\begin{array}{l}
\mathit{Child}(1, \mathit{pos} \leq -1)[\\
\quad \sigma(\mathit{type} = \text{“}\mathit{tree}\text{”}) \\
\quad \sigma(\mathit{label} = \text{“}\texttt{head}\text{”}) \\
\quad \sigma(\mathit{ordered} = \mathit{True}) \\
] \\
\mathit{Child}(2, \mathit{pos} > \mathit{max})[\\
\quad \sigma(\mathit{type} = \text{“}\mathit{tree}\text{”}) \\
\quad \sigma(\mathit{label} = \text{“}\texttt{body}\text{”}) \\
\quad \sigma(\mathit{arity} \geq 1) \\
\quad \sigma(\mathit{ordered} = \mathit{True}) \\
\quad \mathit{Child}(1)[\\
\quad\quad \mathit{Desc}[\\
\quad\quad\quad \sigma(\mathit{type} = \text{“}\mathit{tree}\text{”}) \\
\quad\quad\quad \sigma(\mathit{label} = \text{“}\texttt{p}\text{”}) \\
\quad\quad\quad \sigma(\mathit{ordered} = \mathit{True}) \\
\quad\quad] \\
\quad] \\
]
\end{array}
$$

The expression does the following. First, it cancels out each data term that does not have the proper label `html`*. After verifying that the number of children is greater or equal than two, the child operator applies the nested selection function to all children of the current context nodes. The annotation* $\mathit{pos} \leq -1$ *is a filter condition for the child operator that excludes the last node of each child list from the mapping. It is valid to do so since the data term must have at least one following sibling after the selected one in order to fulfill the query. The operators* $\sigma(\mathit{type} = \text{“}\mathit{tree}\text{”})$*,* $\sigma(\mathit{label} = \text{“}\texttt{head}\text{”})$*, and* $\sigma(\mathit{ordered} = \mathit{True})$ *perform selections local to the children of the root node. The second child operator again applies the nested algebraic expression to the set of all child context nodes. The filter condition* $\mathit{pos} > \mathit{max}$ *requires that the position of the subterms must be greater than the position of the data term selected by the previous* $\mathit{Child}(1, \mathit{pos} \leq -1)$ *operator. In this way, the monotonicity constraint on the mapping is verified incrementally. The nested selection operators perform again local selections on the child context nodes. The third child operator iterates over all child nodes of the nodes selected by the second child operator that fulfill the local selections, and the descendant operator iterates over all descendants of the new context nodes (this navigation in fact includes the possibility to take the same context node). The three local selection criteria* $\sigma(\mathit{type} = \text{“}\mathit{tree}\text{”}), \sigma(\mathit{label} = \text{“}\texttt{p}\text{”})$*, and* $\sigma(\mathit{ordered} = \mathit{True})$ *again discard context nodes from the set of descendant context nodes.*

Textual Notation In the textual notation, it is convenient to write the operators top down instead of one nested into the other. The formal notation, from a mathematical point of view, would make the expression too obscure. The semantics of this textual “top-down” notation is as follows. Let $\circ$ be the higher order function

$$
\begin{array}{rcl}
\circ & : & (T_2 \to T_3) \to (T_1 \to T_2) \to (T_1 \to T_3) \\
f \circ g & = & \lambda x. \quad f(g(x))
\end{array}
$$

Let f_1, f_2 be algebraic expressions. The semantics of the textual notation $\begin{matrix} f_1 \\ f_2 \end{matrix}$ is defined as follows.

$$\begin{matrix} f_1 \\ f_2 \end{matrix} = f_2 \circ f_1$$

The whole expression is again a function from the algebra domain into the algebra domain.

Domain and Operators. In order to introduce the domain, the representation of data terms used in the algebra is defined. This representation is called "data model". The data model consists of one type, *Element*, and functions on this type.

Definition 10 (Data Model) *The data model consists of the type Element and the function elements and descendants.*

- *The function elements is a total function that maps an element node $n \in Element$ to the list of all children of that element node. The order of the list might be relevant (for ordered queries) or of no interest (for unordered queries).*

$$elements : Element \rightarrow List(Element)$$

- *The function descendants is the transitive closure of the function elements.*

$$\begin{array}{lcl} descendants & : & Element \rightarrow Set(Element) \\ descendants\ \ (el) & = & \bigcup_{n=0}^{\infty} elements^n(el) \end{array}$$

Note that by the definition of *descendants*, an element is a descendant of itself.[15] This may seem odd but is necessary to implement the descendant semantics of Xcerpt. In an Xcerpt query, the descendant construct always occurs as a child term, as in `html[[ desc p ]]`. Such a query term is mostly read as "`p` is a descendant of `html`", although this is a slight simplification. The full description would be "`p` is a descendant-or-self of a child of `html`".

Note also that the "data model" here is not the algebra domain, but merely a representation for data terms. The algebra domain uses the data model, but also further data structures.

Besides the given function on nodes, there might be further functions mapping a node to features of that node, but these are not mandatory for the core algebra. Since selections rely on function application together with a comparator operator and a value as discriminator, there must be at least one feature function order to perform local selections. The functions that the algebra requires in order to support the full simulation unification of Xcerpt is presented together with the selection operator below.

Besides the data model, the notion of positions and paths is introduced in the algebra domain. A position is a pair of integer, denoting which query term was mapped to which data term. A path is a set of position, representing a mapping of a query term list to a data term list.

[15] As such, the descendant operator of Xcerpt reflects more the descendant-or-self that the descendant axis of XPath.

Definition 11 (Algebra Domain) *The types Position and Path are defined as follows.*

$$\begin{aligned} Position &= Integer \times Integer \\ Path &= Set(Position) \end{aligned}$$

The algebra domain is the following set.

$$Domain = Set(Element \times Position \times Set(Path))$$

Since the algebra is set based, the domain is a set of potential solutions, represented as tuples of the context node, the node position and the associated set of paths. The first tuple element is the actual context node of the solution. The selection operator uses this node to impose conditions on it (e.g. on label and arity). The third tuple element, a path set, represents the mapping of the query subterms to the data subterms of the actual solution. A tuple (i, j) in a path represents the assignment $q_i \mapsto d_j$. This means that a path stores the indices of the mapped terms instead of storing the whole terms in the mapping. The second item of the tuple is a position, indicating how the actually processed query term is mapped to a data term. If the position is (i, j), then i represents the number of the child operator (which is the same as the position of the query term as specified in the translation scheme) and j represents the data node position within the list of the actual context node. The paths that are associated with a context node represent the mappings of the subquery expressions of map operators of the actual processed query expression to the children nodes of this context node.

Each element of a Set S in the algebra domain represents a "solution context", that is, a potential result of the matching. A solution context is a successful result if its path set is non-empty. If there is no path in the set of paths, this means that the children of the actual context node did not satisfy the subqueries imposed by the child operators of the actual algebraic expression.

Example 11 (Algebra Domain) *The following set* $S : Set(Element \times Position \times Set(Path))$ *is a set of solution contexts:*

$$\{(\texttt{f[a,b]}, (1,1), \{\{(1,1),(2,2)\}\}), (\texttt{g[a,b]}, (1,2), \{\{(1,1),(2,2)\}\})\}$$

The solution context

$$(\texttt{f[a,b]}, (1,1), \{\{(1,1),(2,2)\}\})$$

represents that the context node `f[ a, b ]` *is the first child of a data term list, and was matched by the first map operator of the algebraic expression superior to the actually processed (due to the position* $(1,1)$*). Furthermore, two map operators were applied to the children of the actual context node,* `a, b`*. The first map operator with position* 1 *was mapped to the first child, a, and the second map operator to the second child, b. This is represented by the single path* $\{(1,1),(2,2)\}$ *in the set of paths associated to the context node* `f[ a, b ]`*.*

The following operator has the closest correspondence in the relational algebra, the selection operator σ. Like in the relational algebra, the selection operator filters elements of the input set by a selection criteria.

Definition 12 (Selection) *The selection operator σ takes a function $f : Element \rightarrow T$ that maps a node to a feature of the node, a relation $\_\theta\_ : Set(T \times T')$ in infix notation and a value $v : T'$. Let $S : Set(Element \times Position \times Set(Path))$. The selection operator is*

$$\sigma(f\ \theta\ v)(S) = \{(node, pos, paths) \mid (node, pos, paths) \in S, f(node)\ \theta\ v\}$$

For selections based on the position of a node, the position selection function is defined. Let $S : Set(Element \times Position \times Set(Path))$, $p : Integer$.

$$\sigma^{pos}(p)(S) \quad = \quad \{(el, (i, j), paths) \mid (el, (i, j), paths) \in S, j = p\}$$

The selection operator maps the set S to the set that contains only such solutions in S whose context node satisfies the selection condition. Note that the comparison only reduces the size of the solution set S and operates only on the context node and the position and not on the path set, and that the feature function f is meant to be local, i.e. it does not use the functions *elements* (and *attributes* later on).[16] Note also that selections can express regular expression matching with the comparison operator $\in$ and the value $L(regex)$, e.g. $\sigma(label \in L(\texttt{/.*xml.*/}))$ or with a regular expression matching operator $\sim$ with a value *regexp*, e.g. $\sigma(label \sim \texttt{/.*xml.*/})$. Furthermore, the position is no proper property of a context node, since a node can be a child of several nodes, each time with a different position. For this reason, a second selection operator is introduced to cover selections on node positions.

Definition 13 (Xcerpt Selection Functions) *The following functions are needed to evaluate Xcerpt patterns in the algebra introduced here:*

- *The function type returns the type of an element, that is, whether it is a text element or a tree element.*

$$type : Element \rightarrow \{\text{“}tree\text{”}, \text{“}text\text{”}\}$$

- *The function ordered returns whether the children of the current elements are ordered, if the context is a text element, the value is $\perp$.*

$$ordered : Element \rightarrow Boolean \cup \{\perp\}$$

- *The function arity returns the number of children of the current element, if the context is a tree element and zero if is is a text element.*

$$\begin{array}{lcl} arity & : & Element \rightarrow Integer \\ arity\ \ (el) & = & size(elements(el)) \end{array}$$

- *The function namespace returns the namespace of the element if it is a tree element and $\perp$ otherwise.*

$$namespace : Element \rightarrow String \cup \{\perp\}$$

- *The function label returns the label of the element if it is a tree element and $\perp$ otherwise.*

$$label : Element \rightarrow String \cup \{\perp\}$$

[16] An exception of this rule is the arity function; The arity of each node is assumed to be precalculated on each node in the document.

- *The function value returns the text content if it is a text element and $\perp$ otherwise..*

$$value : Element \rightarrow String \cup \{\perp\}$$

Example 12 (Selection Operator) *Consider the two selections*

$$\sigma_1 = \sigma(arity = 3), \quad \sigma_2 = \sigma(label = f).$$

and the following set $S : Set(Element \times Position \times Set(Path))$

$$S = \{(\texttt{f[a, b, c]}, (1,1), \emptyset), (\texttt{f[a, b]}, (1,2), \emptyset), (\texttt{g[a, b, c]}, (1,3), \emptyset)\}$$

It holds that

$$\begin{aligned} \sigma_1(S) &= \{(\texttt{f[a, b, c]}, (1,1), \emptyset), (\texttt{g[a, b, c]}, (1,3), \emptyset)\} \\ \sigma_2(S) &= \{(\texttt{f[a, b, c]}, (1,1), \emptyset), (\texttt{f[a, b]}, (1,2), \emptyset)\} \\ (\sigma_1 \circ \sigma_2)(S) &= (\sigma_2 \circ \sigma_1)(S) = \{(\texttt{f[a, b, c]}, (1,1), \emptyset)\} \end{aligned}$$

The more complex operators are the child and descendant operators, which are map operators. The child operator takes an algebra expression

$$f_{sub} : Set(Element \times Position \times Set(Path)) \rightarrow Set(Element \times Position \times Set(Path))$$

and applies it to every child of all actual context nodes. Additionally, the child operator takes an index i denoting the operator position and filter conditions. Filter conditions are relations that reduce the number of paths to consider by imposing conditions on the extension of paths. The filter conditions are used to incrementally impose the monotonicity and surjectivity restrictions on paths. Since every path must be injective, there is no filter condition for injective paths and the injectivity condition is verified by default. By respecting the path constraints implied by query specifications, the number of paths may be reduced drastically. For example, a total and ordered specification (i.e. $\texttt{f[}\ t_1, \ldots, t_n\ \texttt{]}$), the monotonicity and surjectivity constraint is so restrictive that there is always only one possible path extension to consider. Hence, the input size can be reduced for the algebraic subexpression f_{sub}. For a partial and ordered specification however (and for unordered specifications even more), the path constraints allow more than one child as match candidate. In such a case, the child operator applies f_{sub} to each candidate. For each candidate that satisfied the algebra subexpression f_{sub}, each existing path is extended with the candidate position if the filter conditions allow this extension.

Recall the statement above about the unary algebra operators being a way to propagate injectivity and monotonicity constraints; the path set is the reason for this property: with a given path set, the child operator can enforce both monotonicity and surjectivity constraints incrementally, as path sets are propagated from child operator to child operator. The operator creates only such paths and adds only such children that do not violate the filter conditions which enforce the path constraints. The filter condition verify if a given extension of the path prefix still satisfies the path constraints. If it does not, the path extension is not created but canceled out.

Definition 14 (Child Operator) *Let* $S : Set(Element \times Position \times Set(Path)), i : Integer$, $f_{sub} : Set(Element \times Position \times Set(Path)) \rightarrow Set(Element \times Position \times Set(Path)), f_1, \ldots, f_n$ *be filter conditions. The child operator is*

$$\begin{aligned} Child \quad & (i, f_1, \ldots, f_n)[f_{sub}](S) = \{(node, pos, paths') \mid (node, pos, paths) \in S, \\ & comp = compatible(size(elements(node)), f_1, \ldots, f_n), \\ & S_{sub} = f_{sub}\{(c, (i, j), \{\emptyset\}) \mid c = elements(node)[j], \exists p \in paths.\ comp(p, (i, j))\}, \\ & paths' = extend(paths, S_{sub}, comp)\} \end{aligned}$$

The function extend used in the definition above is

$$\begin{aligned} extend \quad & (paths, S_{sub}, compatible) = \{path \cup \{pos\} \mid path \in paths, \\ & (c, pos, paths_{sub}) \in S_{sub}, compatible(path, pos)\} \end{aligned}$$

The compatible relation verifies if the path extension is compatible with the already existing path of the solution.

$$\begin{aligned} compatible \quad & (size, f_1, \ldots, f_n) = \{(path, (i, j)) \mid j \notin range(path), \\ & f_1(path, (i, j), size), \ldots, f_n(path, (i, j), size)\} \end{aligned}$$

The child operator does the following. For each solution context in S, it applies the function f_{sub} to the children of the context node. Before doing so however, it verifies for each child if there is a path that can be extended with the child position. An extension is compatible if the filter conditions are valid and if the extended path stays injective. If there is no such path, the child operator will not apply the function f_{sub} to this child. Each child solution context has its position (i, j) associated, so that the function *extend* is able to extend the paths with this position. The *extend* function does so only for paths that, extended with the given position (i, j), respects the filter conditions $f_1, \ldots, f_n$. The filter conditions restrict the possible path extensions and may be one of the following:

Definition 15 (Filter Condition) *A filter condition is a Predicate of* $Path \times Position \times Integer$.

- *Allow only positions less than (or equal to)* $k \in Integer$ *removed from the size of the list:*
$$(pos \leq -k) = \{(path, (i, j), size) \mid j \leq size - k\}$$
- *Allow only the position* $k : Integer$*:*
$$(pos = k) = \{(path, (i, j), size) \mid j = k\}$$
- *Allow only a position that is the maximal position within the path plus one:*
$$(pos = max + 1) = \{(path, (i, j), size) \mid j = max(range(path)) + 1\}$$
Whereby the function max is understood as the maximum of a set of positive integers.
$$\begin{aligned} max \quad & : \quad Set(Integer) \rightarrow Integer \\ max(S) \quad & = \quad \begin{cases} 0 & \text{if } S = \emptyset \\ i \quad \text{such that } \forall j \in S. \quad i \geq j & \text{otherwise.} \end{cases} \end{aligned}$$

- *Allow only following siblings:*

$$(pos > max) = \{(path, (i, j), size) \mid j > max(range(path))\}$$

Example 13 (Filter Conditions and Child Operator) *Some example for filter condition application.*

$$\begin{array}{rcll}
(\emptyset, (1,1), 6) & \in & (pos > max) & since\ 1 > 0 \\
(\{(1,3),(2,4)\}, (3,2), 6) & \notin & (pos > max) & since\ 2 \not> 4 \\
(\{(1,3),(2,4)\}, (3,5), 6) & \in & (pos > max) & since\ 5 > 4 \\
 & & & \\
(\{(1,1),(2,2)\}, (3,5), 6) & \notin & (pos = max + 1) & since\ 5 \neq 3 \\
(\{(1,1),(2,2)\}, (3,3), 6) & \in & (pos = max + 1) & since\ 3 = 3 \\
(\emptyset, (1,1), 6) & \in & (pos \leq -3) & since\ 1 \leq 3 \\
(\{(1,1)\}, (2,4), 6) & \notin & (pos \leq -3) & since\ 4 \not\leq 3
\end{array}$$

Assume the child operator

$$C = Child(1, pos \leq -2)[f_{sub}]$$

And the set

$$S = \{(\texttt{f[a, b, c, d]}, (0,0), \{\emptyset\})\}$$

The evaluation of $C(S)$ starts with choosing an element out of S. Since there is only one element in S, the choice is

$$(\texttt{f[a, b, c, d]}, (0,0), \{\emptyset\})$$

The relation comp is then initialized as

$$comp = compatible(4, pos \leq -2)$$

The set of children to which f_{sub} is applied is

$$\{(\texttt{a}, (1,1), \{\emptyset\}), (\texttt{b}, (1,2), \{\emptyset\})\}$$

because the filter condition $pos \leq -2$ is never satisfied for the subterms with position 3 and 4.

$$\begin{array}{rcll}
(\emptyset, (1,3), 4) & \notin & (pos \leq -2) & since\ 3 \not\leq 2 \\
(\emptyset, (1,4), 4) & \notin & (pos \leq -2) & since\ 4 \not\leq 2
\end{array}$$

Assuming that the result of the algebraic expression f_{sub} is

$$S_{sub} = \{(\texttt{a}, (1,1), \{\emptyset\}), (\texttt{b}, (1,2), \{\emptyset\})\},$$

extend is applied to the paths $\{\emptyset\}$, the above set of subsolutions and the relation comp. Since both positions $(1,1)$ and $(1,2)$ are compatible with the empty path, the extend function returns the path set $\{\{(1,1)\}, \{(1,2)\}\}$. This set represents that the child operator 1 can be mapped to both the first and the second data term.

Altogether, the result is

$$\begin{array}{ll}
Child(1, pos \leq -2)[f_{sub}] & (\{(\texttt{f[a, b, c, d]}, (0,0), \{\emptyset\})\}) = \\
 & \{(\texttt{f[a, b, c, d]}, (0,0), \{\{(1,1)\}, \{(1,2)\}\})\}
\end{array}$$

An operator similar to the child operator is the descendant operator. This operator uses the transitive closure of the *elements* function and applies its algebraic expression f_{sub} to all descendants. Furthermore, it does not extend the path set of the actual context solutions, since the descendant operator will always occur as a single operator within a child operator.

Definition 16 (Descendant Operator) *Let* $S : Set(Element \times Position \times Set(Path))$, $f_{sub} : Set(Element \times Position \times Set(Path)) \rightarrow Set(Element \times Position \times Set(Path))$. *The descendant operator is*

$$\begin{aligned} Desc[f_{sub}] \quad = \quad & \{(node, pos, paths) \mid (node, pos, paths) \in S, \\ & f_{sub}\{(d, (0,0), \{\emptyset\}) \mid d \in descendants(node)\} \neq \emptyset\} \end{aligned}$$

The descendant operator verifies whether there is a descendant fulfilling the selection function f_{sub}. If there is no descendant fulfilling f_{sub}, the subsolution set is empty. Note that a position selection operator is undefined within a descendant operator, because the position is arbitrarily set to zero.

It is possible to retrieve the position of descendant elements, but this would complicate the definition. A problem of doing so is that the single node can be a descendant of an ancestor at two different positions. Thus Xcerpt does not support access to descendant position.

Evaluation of Algebra Expressions. When evaluating an algebraic expression *expr* with a set of nodes E, the algebraic expression can not be applied to the set of nodes directly, since the algebra domain is not $Set(Element)$. Instead, the set of nodes must be prepared. Hence, the expression to evaluate is the following:

$$expr\{(el, (0,0), \{\emptyset\}) \mid el \in E\}.$$

The evaluation result is again a set of $Element \times Position \times Set(Path)$. To return the elements that the query term matched, the evaluation verifies whether a given solution context has a non-empty set of paths associated.

Definition 17 (Evaluation) *The evaluation of an algebraic expression* f *with the input elements* E *is*

$$\begin{aligned} eval \quad & : \quad (Domain \rightarrow Domain) \rightarrow Set(Element) \rightarrow Set(Element) \\ eval\ f\ E \quad & = \quad \{el \mid (el, pos, paths) \in f\{(el, (0,0), \{\emptyset\}) \mid el \in E\}, paths \neq \emptyset\} \end{aligned}$$

The evaluation of an algebra expression returns the nodes satisfying the query. As variables are introduced, the set of constraint stores is returned as the evaluation result (see section 3.3.4).

Translation Scheme. All basic operators are defined, so that a translation scheme can be proposed for simple query terms into the algebra. The base cases for query terms translation,

which are string and regular expression matchings, are as follows.

$$\begin{aligned} Tr_{term}(\texttt{'"'}\ \texttt{string}\ \texttt{'"'}) &= \sigma(type = \text{“}text\text{”}) \\ &\quad\ \sigma(value = string) \\ Tr_{term}(\texttt{"/"}\ \texttt{regexp}\ \texttt{"/"}) &= \sigma(type = \text{“}text\text{”}) \\ &\quad\ \sigma(value \sim regexp) \\ Tr_{term}(\texttt{number}) &= \sigma(type = \text{“}text\text{”}) \\ &\quad\ \sigma(value = number) \end{aligned}$$

The descendant operator requires a recursive call of the translation function and uses the result of the translation as subexpression.

$$Tr_{term}(\texttt{"desc"}\ \texttt{<query-subterm>}) = Desc[\ Tr(\texttt{<query-subterm>})\ ]$$

The algebra handles position specifications by introducing a position selection operator.

$$\begin{aligned} Tr_{term}(\texttt{"position"}\ \texttt{number}\ \texttt{<query-subterm>}) &= \sigma^{pos}(number) \\ &\quad\ Tr_{term}(\texttt{<query-subterm>}) \end{aligned}$$

The translation of composed query terms follows. Partial and total unordered lists are the easiest to translate, since they impose no further path constraints beyond path injectivity.

$$\begin{aligned} &Tr_{term}(\texttt{<ns-prefix>":"<label>"\{"<query-subterm>}_1\texttt{","...","}\ \texttt{<query-subterm>}_n\texttt{"\}"}) \\ =\ & \sigma(type = \text{“}tree\text{”}) \\ & Tr_{val}(namespace, \texttt{<ns-prefix>}) \\ & Tr_{val}(label, \texttt{<label>}) \\ & \sigma(arity = n) \\ & Child(1)[\ Tr_{term}(\texttt{<query-subterm>}_1)\ ] \\ & \ldots \\ & Child(n)[\ Tr_{term}(\texttt{<query-subterm>}_n)\ ] \end{aligned}$$

$$\begin{aligned} &Tr_{term}(\texttt{<ns-prefix>":"<label>"\{\{"<query-subterm>}_1\texttt{","...","}\ \texttt{<query-subterm>}_n\texttt{"\}\}"}) \\ =\ & \sigma(type = \text{“}tree\text{”}) \\ & Tr_{val}(namespace, \texttt{<ns-prefix>}) \\ & Tr_{val}(label, \texttt{<label>}) \\ & \sigma(arity \geq n) \\ & Child(1)[\ Tr_{term}(\texttt{<query-subterm>}_1)\ ] \\ & \ldots \\ & Child(n)[\ Tr_{term}(\texttt{<query-subterm>}_n)\ ] \end{aligned}$$

Since the programmer can specify the labels and namespaces with regular expressions and strings, the translation scheme Tr_{val} is used to handle these instead of introducing selections directly.

$$\begin{aligned} Tr_{val}(f, \texttt{"/"}\ \texttt{regexp}\ \texttt{"/"}) &= \sigma(f \sim regexp) \\ Tr_{val}(f, \texttt{'"'}\ \texttt{string}\ \texttt{'"'}) &= \sigma(f = string) \\ Tr_{val}(f, \texttt{label}) &= \sigma(f = label) \end{aligned}$$

The translation of a total and ordered subterm specifications is similarly straightforward, since the constraints induced are very restrictive: $(pos = max + 1)$ is added as filter condition to every *Child* operator, restricting the number of candidate children to one.

Tr_{term}(<ns-prefix>":"<label>"["<query-subterm>$_1$","...","<query-subterm>$_n$"]")
= $\sigma(type = \text{“}tree\text{”})$
$Tr_{val}(namespace, \texttt{<ns-prefix>})$
$Tr_{val}(label, \texttt{<label>})$
$\sigma(arity = n)$
$Child(1, pos = max + 1)[\ Tr_{term}(\texttt{<query-subterm>}_1)\ ]$
...
$Child(n, pos = max + 1)[\ Tr_{term}(\texttt{<query-subterm>}_n)\ ]$

The translation of partial and ordered subterm specifications require more sophisticated constraints than the previous specifications. First, if there are sibling query terms before the translated term t, the child operator for t must ensure that the new mapping respects the monotonicity constraint, i.e. the position of the next selected sibling must be greater than any data term position in the considered path. This induces the constraint $(pos > max)$ for all child operators except the first. Second, in order to ensure that there is always such a sibling, the child operator must ensure that there are still enough siblings after the currently selected. For a subterm list of size n and a subterm position of p (counting from one), this implies the constraint $(pos \leq -(n - p))$. This obviously holds for all child operators except for the last.

Tr_{term}(<ns-prefix>":"<label>"[["<query-subterm>$_1$","...","<query-subterm>$_n$"]]")
= $\sigma(type = \text{“}tree\text{”})$
$Tr_{val}(namespace, \texttt{<ns-prefix>})$
$Tr_{val}(label, \texttt{<label>})$
$\sigma(arity \geq n)$
$Child(1, pos \leq -(n-1))[\ Tr_{term}(\texttt{<query-subterm>}_1)\ ]$
$Child(2, pos > max, pos \leq -(n-2))[\ Tr_{term}(\texttt{<query-subterm>}_2)\ ]$
...
$Child(n-1, pos > max, pos \leq -1)[\ Tr_{term}(\texttt{<query-subterm>}_{n-1})\ ]$
$Child(n, pos > max)[\ Tr_{term}(\texttt{<query-subterm>}_n)\ ]$

Example 14 (Example Translation and Evaluation) *Consider the following query term:* `f[ a, g{{ position 3 b, h[[ c, d, e ]] }}, desc a ]` *This term is translated into the following expression.*

$\sigma(type = \text{“}tree\text{”})$
$\sigma(label = \text{“}\texttt{f}\text{”})$
$\sigma(arity = 3)$
$\sigma(ordered = True)$
$Child(1, pos = max + 1)[$
$\quad \sigma(type = \text{“}tree\text{”})$
$\quad \sigma(label = \text{“}\texttt{a}\text{”})$
$\quad \sigma(arity = 0)$
$]$
$Child(2, pos = max + 1)[$
$\quad \sigma(type = \text{“}tree\text{”})$
$\quad \sigma(label = \text{“}\texttt{g}\text{”})$
$\quad \sigma(arity \geq 2)$

$$
\begin{array}{l}
\quad Child(1)[\\
\qquad \sigma^{pos}(3) \\
\qquad \sigma(type = \text{“}tree\text{”}) \\
\qquad \sigma(label = \text{“}\texttt{b}\text{”}) \\
\qquad \sigma(arity = 0) \\
\quad] \\
\quad Child(2)[\\
\qquad \sigma(type = \text{“}tree\text{”}) \\
\qquad \sigma(label = \text{“}\texttt{h}\text{”}) \\
\qquad \sigma(arity \geq 3) \\
\qquad \sigma(ordered = True) \\
\qquad Child(1, pos \leq -2)[\\
\qquad\quad \sigma(type = \text{“}tree\text{”}) \\
\qquad\quad \sigma(label = \text{“}\texttt{c}\text{”}) \\
\qquad\quad \sigma(arity = 0) \\
\qquad] \\
\qquad Child(2, pos > max, pos \leq -1)[\\
\qquad\quad \sigma(type = \text{“}tree\text{”}) \\
\qquad\quad \sigma(label = \text{“}\texttt{d}\text{”}) \\
\qquad\quad \sigma(arity = 0) \\
\qquad] \\
\qquad Child(3, pos > max)[\\
\qquad\quad \sigma(type = \text{“}tree\text{”}) \\
\qquad\quad \sigma(label = \text{“}\texttt{e}\text{”}) \\
\qquad\quad \sigma(arity = 0) \\
\qquad] \\
\quad] \\
] \\
Child(3, pos = max + 1)[\\
\quad Desc[\\
\qquad \sigma(type = \text{“}tree\text{”}) \\
\qquad \sigma(label = \text{“}\texttt{a}\text{”}) \\
\qquad \sigma(arity = 0) \\
\quad] \\
]
\end{array}
$$

Another query which will be unified with the data term `f[ a, g[ a, b, f[ c, d ] ] ]` *is* `f[ a, g{{ position 2 b, desc c }} ]`. *The translation of this query term into the algebra is*

$$
\begin{array}{l}
\sigma(type = \text{“}tree\text{”}) \\
\sigma(label = \text{“}\texttt{f}\text{”}) \\
\sigma(ordered = True) \\
\sigma(arity = 2) \\
Child(1, pos = max + 1)[\\
\quad \sigma(type = \text{“}tree\text{”}) \\
\quad \sigma(label = \text{“}\texttt{a}\text{”}) \\
\quad \sigma(arity = 0) \\
]
\end{array}
$$

$$
\begin{array}{l}
\mathit{Child}(2, \mathit{pos} = \mathit{max} + 1)[\\
\quad \sigma(\mathit{type} = \text{“}\mathit{tree}\text{”}) \\
\quad \sigma(\mathit{label} = \text{“}\texttt{g}\text{”}) \\
\quad \sigma(\mathit{arity} \geq 3) \\
\quad \mathit{Child}(1)[\\
\quad\quad \sigma^{\mathit{pos}}(2) \\
\quad\quad \sigma(\mathit{type} = \text{“}\mathit{tree}\text{”}) \\
\quad\quad \sigma(\mathit{label} = \text{“}\texttt{b}\text{”}) \\
\quad\quad \sigma(\mathit{arity} = 0) \\
\quad] \\
\quad \mathit{Child}(2)[\\
\quad\quad \mathit{Desc}[\\
\quad\quad\quad \sigma(\mathit{type} = \text{“}\mathit{tree}\text{”}) \\
\quad\quad\quad \sigma(\mathit{label} = \text{“}\texttt{c}\text{”}) \\
\quad\quad\quad \sigma(\mathit{arity} = 0) \\
\quad\quad] \\
\quad] \\
]
\end{array}
$$

The evaluation begins with the set

$$S = \{(\texttt{f[a, g[a, b, f[c, d]]]}, (0,0), \{\emptyset\})\}.$$

Due to the filter condition $\mathit{pos} = \mathit{max} + 1$ *of the map operator* $\mathit{Child}(1, \mathit{pos} = \mathit{max} + 1)[f_{sub}]$, *the set of children to which* f_{sub} *is applied consists of one single child*

$$\{(\texttt{a}, (1,1), \{\emptyset\})\}$$

The child satisfies the subexpression, so that the position $(1,1)$ *can be combined with the path* $\emptyset$. *The resulting set of solutions is*

$$\mathit{Child}(1, \mathit{pos} = \mathit{max} + 1)[f_{sub}](S) = \{(\texttt{f[a, g[a, b, f[c, d]]]}, (0,0), \{\{(1,1)\}\})\}$$

Consider now $\mathit{Child}(2, \mathit{pos} = \mathit{max} + 1)$. *The only child which satisfies the filter condition is the second.*

$$S_{sub} = \{(\texttt{g[a, b, f[c, d]]}, (2,2), \{\emptyset\})\}$$

The subexpression is applied to this node. The context node satisfies the three selection operators so that the evaluation of the first child operator $\mathit{Child}(1)[f_{sub}]$ *considers all three children of the context node* `g[ a, b, f[ c, d ] ]`.

$$S_{sub_{sub}} = \{(\texttt{a}, (1,1), \{\emptyset\}), (\texttt{b}, (1,2), \{\emptyset\}), (\texttt{f[c, d]}, (1,3), \{\emptyset\})\}$$

The position selection however reduces this set to

$$\sigma^{\mathit{pos}}(2)(S_{sub_{sub}}) = \{(\texttt{b}, (1,2))\},$$

so that after combining the position with the solution context path, the result is

$$\mathit{Child}(1)[f_{sub}](S_{sub}) = \{(\texttt{g[a, b, f[c, d]]}, (2,2), \{\{(1,2)\}\})\}$$

The algebraic subexpression f_{sub} of the second child operator

$$Child(2)[Desc[\sigma(arity = 0) \circ \sigma(label = \text{“}\texttt{c}\text{”}) \circ \sigma(type = \text{“}tree\text{”})]]$$

is applied to the following set of context nodes:

$$S_{sub_{sub}} = \{(\texttt{a}, (2,1), \{\emptyset\}), (\texttt{f[c, d]}, (2,3), \{\emptyset\})\}$$

Note that the second child is missing, since the single path in the path set already matched the second child.

$$2 \in range(\{(1,2)\})$$

The descendant operator is applied to the solution contexts in $S_{sub_{sub}}$. Assume that the first solution context is chosen, and that the descendants of `a` *are initialized:*

$$S_{desc} = \{(\texttt{a}, (0,0), \{\emptyset\})\}$$

Since the selection $\sigma(label = \text{“}\texttt{b}\text{”})$ fails, the subsolution set is empty, and the evaluation of the descendant operator discards the context $(\texttt{a}, (2,1), \{\emptyset\})$. After handling the first context node, the second one is chosen, and the following three descendant nodes are the new candidates for the algebraic subexpression of the descendant operator:

$$S_{desc} = \{(\texttt{f[c, d]}, (0,0), \{\emptyset\}), (\texttt{c}, (0,0), \{\emptyset\}), (\texttt{d}, (0,0), \{\emptyset\})\}$$

The selection $\sigma(label = \text{“}\texttt{c}\text{”})$ reduces this set to

$$\sigma(label = \text{“}\texttt{c}\text{”})(S_{desc}) = \{(\texttt{c}, (0,0), \{\emptyset\})\}$$

Since this set is non-empty, the solution

$$(\texttt{f[c, d]}, (2,3), \{\emptyset\})$$

is kept in the solution set. The child operator $Child(2)[f_{sub}]$ hence gets a subsolution set $S_{sub_{sub}}$ with one solution context. The combination of this solution with the existing path succeeds so that the result becomes

$$Child(2)[f_{sub}](S_{sub}) = \{(\texttt{g[a, b, f[c, d]]}, (2,2), \{\{(1,2), (2,3)\}\})\}$$

The evaluation of the subexpression f_{sub} of the child operator $Child(2, pos = max + 1)[f_{sub}]$ is finished, so that these results must be combined with the paths in S. The position $(2,2)$ is combined with the path $\{(1,1)\}$. The result is

$$Child(2, pos = max{+}1)[f_{sub}](S) = \{(\texttt{f[a, g[a, b, f[c, d]]]}, (0,0), \{\{(1,1), (2,2)\}\})\}$$

as the final solution context. The query is successful since the solution set contains one element with a non-empty set of paths. The result set of the evaluation is

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

The algebra contains two kinds of operators: local operators and map operators. The map operators apply an algebraic subexpression f_{sub} to a set of context nodes that is different to the actual set of context nodes. The result of the application of f_{sub} is then combined with the paths of the actual context nodes. The local operators modify the actual set of contexts without navigating to an other set of context nodes. The only local operator presented in this section was the selection operator, but further are introduced in the next sections.

So far, the evaluation of an algebraic expression can only determine the set of data terms a query matches. The query algebra does not yet provide any means to bind variables. This restriction must be lifted in order to have a valuable algebra for Xcerpt where variables are essential to express n-ary queries as well as non-structural joins. Since attribute specifications may also contain variables, it is advisable to handle attributes first before operators for variables are introduced.

3.3.3 Extending the Algebra for Attributes

This first extension of the algebra covers attribute query term lists. Attribute query lists are similar but simpler in structure than query term lists, since attribute query lists are always unordered. Furthermore, the attributes themselves do not contain further child elements. Because of this simpler structure, it is preferable to execute the attribute query before the subterm query. In order to do this, term attributes are not normalized to a nested attributes element, but supported directly.

Algebra Domain and Operators. The first extension to support attributes is made to the data model. In order to distinguish attributes from elements, another type is introduced into the data model, *Attribute*.

Definition 18 (Data Model Extension) *The data model of the algebra is extended in the following way.*

- *The data model contains a new type Attribute with the property that*

$$Element \cap Attribute = \emptyset.$$

- *The function attributes* : $Element \rightarrow List(Attribute)$ *returns the attributes of an element.*

Since the programmer can specify attribute labels, namespaces and values as strings or regular expressions (or as variables), the path injectivity must be ensured in the same way as for element list queries. In order to keep track of the attribute mapping, the attribute position is used as local identifier for an attribute. Be aware that attribute query terms can not be assumed to be distinct (as in the case of attribute data terms, where each attribute data term of the same node must have a distinct label), since variables and regular expressions can be used instead of attribute labels.

Note that no additional attribute position set is introduced into the algebra domain. The position set of the algebra domain is used for both child and attribute mapping. In order to do so, an additional operator is introduced, the *Clear* operator, which removes all path information.

Definition 19 (Algebra Domain) *The two algebra domains are*

$$\begin{aligned} Domain &= Set(Element \times Position \times Set(Path)) \\ AttrDomain &= Set(Attribute \times Position \times Set(Path)) \end{aligned}$$

A second algebra domain becomes necessary, *AttrDomain*, since selections are also performed on attributes. In order to do so, the selection operator σ must be overloaded to both domains and the functions it uses must be extended. $namespace, label$ and $value$ are defined on attribute nodes with the expected semantics. Additionally, a selection function $arity_a(element) = size(attributes(element))$ is introduced. This function returns the number of attributes of a node. Since the selection operator ignores the path information, the extended σ operator definition is the same as before.

Name	**Notation**	**Short Description**
Selection	$\sigma(f\ \theta\ v)$	Does the same as for elements.
Attribute	$Attr(i)[f_{sub}]$	Selects all attributes satisfying f_{sub}.
Clear	$Clear$	Removes all path information.

Table 2: Operators for Attribute Queries

Definition 20 (Clear Operator) *Let* $S : Set(Element \times Position \times Set(Path))$. *The clear operator is*

$$Clear(S) = \{(node, pos, \{\emptyset\}) \mid (node, pos, paths) \in S, paths \neq \emptyset\}$$

Note that the clear operator also removes the falsified solutions having no paths at all. This is necessary in order to clear only non-falsified solutions. If the condition $paths \neq \emptyset$ was not introduced, falsified solutions would be transformed into valid solutions.

Finally, a second "map"-like operator $Attr(i)[f]$ is introduced, which applies the algebraic expression $f_{sub} : Set(Attribute \times Position \times Set(Path)) \rightarrow Set(Attribute \times Position \times Set(Path))$ to all attributes of all context nodes.

Definition 21 (Attribute Operator) *Let* $S : Set(Element \times Position \times Set(Path))$, $f_{sub} : Set(Attribute \times Position \times Set(Path)) \rightarrow Set(Attribute \times Position \times Set(Path)), i :$ *Integer. The attribute operator is*

$$\begin{aligned} &Attr(i)[f_{sub}](S) = \{(node, pos, paths') \mid (node, pos, paths) \in S, \\ &\quad comp = compatible(0), \\ &\quad S_{sub} = f_{sub}\{(att, (i, j), \{\emptyset\}) \mid att = attributes(node)[j], \exists p \in paths.\ comp(p, (i, j))\}, \\ &\quad paths' = extend(paths, S_{sub}, comp)\} \end{aligned}$$

The attribute operator is very similar to the child operator. However, it does not use any filter conditions as path injectivity is verified by default, and no other filter conditions (based on position) apply to attributes since the are always unordered.

Translation Scheme. The extensions above are already sufficient to handle attributes: a second domain, a new map operator, an overloaded selection operator and new selection functions. With all these defined, we can translate attribute queries into the algebra. The translation schema for the attribute query list is as follows.

$$
\begin{aligned}
& Tr_{att}(\texttt{"("}\ \texttt{<ns-prefix>}_1\ \texttt{":"}\ \texttt{<label>}_1\ \texttt{"="}\ \texttt{<value>}_1 \\
& \qquad \texttt{","}\ \ldots\ \texttt{","} \\
& \qquad \texttt{<ns-prefix>}_n\ \texttt{":"}\ \texttt{<label>}_n\ \texttt{"="}\ \texttt{<value>}_n\ \texttt{")"}) \\
= \quad & \sigma(arity_a = n) \\
& Attr(1)[\\
& \qquad Tr_{val}(namespace, \texttt{<ns-prefix>}_1) \\
& \qquad Tr_{val}(label, \texttt{<label>}_1) \\
& \qquad Tr_{val}(value, \texttt{<value>}_1) \\
&] \\
& \ldots \\
& Attr(n)[\\
& \qquad Tr_{val}(namespace, \texttt{<ns-prefix>}_n) \\
& \qquad Tr_{val}(label, \texttt{<label>}_n) \\
& \qquad Tr_{val}(value, \texttt{<value>}_n) \\
&] \\
& \qquad Tr_{att}(\texttt{"(("}\ \texttt{<ns-prefix>}_1\ \texttt{":"}\ \texttt{<label>}_1\ \texttt{"="}\ \texttt{<value>}_1 \\
& Clear \qquad\qquad \texttt{","}\ \ldots\ \texttt{","} \\
& \qquad\qquad \texttt{<ns-prefix>}_n\ \texttt{":"}\ \texttt{<label>}_n\ \texttt{"="}\ \texttt{<value>}_n\ \texttt{"))"}) \\
= \quad & \sigma(arity_a \geq n) \\
& Attr(1)[\\
& \qquad Tr_{val}(namespace, \texttt{<ns-prefix>}_1) \\
& \qquad Tr_{val}(label, \texttt{<label>}_1) \\
& \qquad Tr_{val}(value, \texttt{<value>}_1) \\
&] \\
& \ldots \\
& Attr(n)[\\
& \qquad Tr_{val}(namespace, \texttt{<ns-prefix>}_n) \\
& \qquad Tr_{val}(label, \texttt{<label>}_n) \\
& \qquad Tr_{val}(value, \texttt{<value>}_n) \\
&] \\
& Clear
\end{aligned}
$$

The algebra expression for the attribute query list is inserted into the translation scheme above between the selection operations of the composed term translation and the child operators (if there are any).

Example 15 (Translation and Evaluation) *Assume the following artificial query term*

```
f( /a*/=/*/, /*b/=/*/ )[[ /*a/, /b*/ ]]
```

and the following data term:

```
f( ab=1, ab=2 )[ a, b ]
```

The translation of the query term into the algebra is

$$
\begin{array}{l}
\sigma(type = \text{“}tree\text{”}) \\
\sigma(label = \text{“}\texttt{f}\text{”}) \\
\sigma(arity_a = 2) \\
Attr(1)[\\
\qquad \sigma(label \sim \texttt{a*}) \\
\qquad \sigma(value \sim \texttt{*}) \\
] \\
Attr(2)[\\
\qquad \sigma(label \sim \texttt{*b}) \\
\qquad \sigma(value \sim \texttt{*}) \\
] \\
Clear \\
\sigma(arity \geq 2) \\
\sigma(ordered = True) \\
Child(1, pos = max + 1)[\\
\qquad \sigma(Type = \text{“}tree\text{”}) \\
\qquad \sigma(label \sim \texttt{a*}) \\
\qquad \sigma(arity = 0) \\
] \\
Child(2, pos = max + 1)[\\
\qquad \sigma(Type = \text{“}tree\text{”}) \\
\qquad \sigma(label \sim *b) \\
\qquad \sigma(arity = 0) \\
]
\end{array}
$$

The evaluation of the algebraic expression with the data term `f( ab=1, ab=2 )[ a, b ]` *starts with the set*

$$S = \{(\texttt{f(ab=1, ab=2)[a, b]}, (0,0), \{\emptyset\})\}.$$

The context node satisfies local selections on type, label and value, so that the attribute operator is evaluated.

$$Attr(1)[\sigma(value \sim \texttt{*}) \circ \sigma(label \sim \texttt{a*})]$$

The set of attributes to consider for the algebraic subexpression is

$$\{(\texttt{ab=1}, (1,1), \{\emptyset\}), (\texttt{ab=2}, (1,2), \{\emptyset\})\}$$

Both attributes satisfy the selection conditions, so that this is at the same time the result of the algebraic subexpression.

$$
\begin{array}{rl}
\sigma(value \sim \texttt{*}) \circ \sigma(label \sim \texttt{*b}) & \{(\texttt{ab=1}, (1,1), \{\emptyset\}), (\texttt{ab=2}, (1,2), \{\emptyset\})\} \\
= & \{(\texttt{ab=1}, (1,1), \{\emptyset\}), (\texttt{ab=2}, (1,2), \{\emptyset\})\}
\end{array}
$$

Combining these results with the path leads to the solution set

$$
\begin{array}{rl}
Attr(1)[\sigma(value \sim \texttt{*}) \circ \sigma(label \sim \texttt{*b})] & \{(\texttt{f(ab=1, ab=2)[a, b]}, (0,0), \{\emptyset\})\} \\
= & \{(\texttt{f(ab=1, ab=2)[a, b]}, (0,0), \{\{(1,1)\}, \{(1,2)\}\})\}.
\end{array}
$$

To evaluate the application of the second attribute operator

$$Attr(2)[\sigma(value \sim \texttt{*}) \circ \sigma(label \sim \texttt{*b})],$$

the subexpression is evaluated with the same set of attributes as above. Note that there is a compatible path in the solution context for both attributes.

$$\begin{aligned} \sigma(value \sim \texttt{*}) \circ \sigma(label \sim \texttt{*b}) \quad & \{(\texttt{ab=1}, (1,1), \{\emptyset\}), (\texttt{ab=2}, (1,2), \{\emptyset\})\} \\ = \quad & \{(\texttt{ab=1}, (2,1), \{\emptyset\}), (\texttt{ab=2}, (2,2), \{\emptyset\})\} \end{aligned}$$

Due to the path injectivity restriction, only two out of four combinations of paths and positions are created:

$$\{\{(1,1),(2,2)\},\{(1,2),(2,1)\}\},$$

because $\{(1,2),(2,2)\}$ *and* $\{(1,1),(2,1)\}$ *are no injective paths. The solution set at this point of the evaluation is*

$$S = \{(\texttt{f(ab=1, ab=2)[a, b]}, (0,0), \{\{(1,1),(2,2)\},\{(1,2),(2,1)\}\})\}.$$

The clear operator removes the path information gathered so far, since the children of the actual context node will be queried next.

$$Clear(S) = \{(\texttt{f(ab=1, ab=2)[a, b]}, (0,0), \{\emptyset\})\}.$$

Applying both child operators will then lead to the solution context

$$S = \{(\texttt{f(ab=1, ab=2)[a, b]}, (0,0), \{\{(1,1),(2,2)\}\})\}.$$

The query is hence successful, and the result is

$$\{\texttt{f(ab=1, ab=2)[a, b]}\}$$

Design Considerations for the Clear Operator. Introducing the clear operator to reuse the path is a design decision leading to a smaller redefinition effort here and in the following. It is possible to introduce a second path element in the algebra domain that is reserved for attribute mappings. This would allow to intertwine attribute and child operators, but also leads to verbose and complicated definitions. For this reason, we choose the shorter and simpler variant, although an operator like the clear operator may seem misplaced in a logical algebra. After the complete definition of the algebra with variables, negation and optional subterm constructs, the reader may want to remove the clear operator and introduce a second path component into the algebra domain.

The extension of the algebra for attribute queries is otherwise a straightforward extension. The new operators are slightly modified versions of the base operators. The concepts for querying attributes are the same as for querying children. Attribute queries are even simpler, since they may not contain nested elements. Attribute queries should be executed first for this reason.

3.3.4 Extending the Algebra for Variables

So far, the algebra considered ground query terms only. In this section, variables are introduced into the algebra. Variables are in fact needed to build ground instances of the rule heads associated with the query. In order to preserve the information about the variable bindings through the query process, the query domain is extended by a set of variable bindings. Variable bindings themselves look like simulation constraints. The variable binding sets are called "constraint store" here to account for the fact that the bindings are constraints, and that the same variable may occur several times and thus become constrained several times.

Definition 22 (Constraint Store) *Let el : Element, str : String and X be a variable. The set of atomic simulation constraints, CS_{atomic}, is the smallest set such that*

- $X \preceq el \in CS_{atomic}$
- $X \preceq str \in CS_{atomic}$

Furthermore, let $Store = Set(CS_{atomic}) \cup \bot$. $\bot$ is called inconsistent constraint store.

The meaning of a constraint is the same as in the constraint calculus in section 2.2. A constraint store is interpreted as a conjunction of its constraints, and the empty constraint store is hence true. The algebra transforms every constraint formula into a disjunctive normal form, handling every disjunct in a separate constraint store.

Name	Notation	Short Description
Bind	$Bind(X)$	Binds the variable X the actual context node.
Bind value	$\beta(X = fun)$	Binds X to the value of fun on the actual context node.
Bind position	$\beta^{pos}(X)$	Binds X to the position of the actual context node.

Table 3: Operators for Variables

The algebra domain is extended to contain a set of constraint stores for each node. More precisely, a path is associated to each constraint store. The path represents, as before, the mapping of query attributes to data attributes and query terms to data terms that lead to the associated constraint store. Hence, more than one solution is grouped together with a single node. This grouping of solutions is very important for the combination of solutions, especially when several solutions must be negated as for the handling of negated and optional subterms as section 3.3.5 reveals.

Definition 23 (Algebra Domain) *A path is redefined an element of the following set.*

$$Path \quad = \quad Set(Integer \times Integer) \times Store$$

The child and attribute operator must initialize the paths of the children with the true path. Previously, this was the empty set. The true path is

$$TruePath = (\emptyset, \emptyset)$$

The algebra domains are thus:

$$\begin{aligned} \mathit{Domain} &= \mathit{Set}(\mathit{Element} \times \mathit{Position} \times \mathit{Set}(\mathit{Path})) \\ \mathit{AttrDomain} &= \mathit{Set}(\mathit{Attribute} \times \mathit{Position} \times \mathit{Set}(\mathit{Path})) \end{aligned}$$

Again, the definition of a path is changed. The selection operator is extended canonically to the new domain, since it leaves the path set unchanged. The extend function used in the child and attribute operators must be redefined to append the constraint store resulting from the subquery to the the store of the solution contexts. At the same time, the descendant operator must be redefined to do the same.

Definition 24 (Extend Function) *The function extend used in the definition of the Attribute and Child operator is redefined as*

$$\begin{aligned} \mathit{extend}\quad & (\mathit{paths}, S_{sub}, \mathit{compatible}) = \{(\mathit{path} \cup \{\mathit{pos}\}, \mathit{store}') \mid (\mathit{path}, \mathit{store}) \in \mathit{paths}, \\ & (c, \mathit{pos}, \mathit{paths}_{sub}) \in S_{sub}, (\mathit{path}_{sub}, \mathit{store}_{sub}) \in \mathit{paths}_{sub}, \\ & \mathit{compatible}(\mathit{path}, \mathit{pos}), \mathit{store}' = \mathit{verify}(\mathit{store}, \mathit{store}_{sub}), \mathit{store}' \neq \bot\} \end{aligned}$$

The verify function is

$$\mathit{verify}(s_1, s_2) = \begin{cases} \bot & \text{if } s_1 = \bot \text{ or } s_2 = \bot \\ s_1 \cup s_2 & \text{if } \forall X \preceq n_1 \in s_1, \forall X \preceq n_2 \in s_2. \quad \mathit{bisim}(n_1, n_2) \\ \bot & \text{otherwise.} \end{cases}$$

The extend function used in the child and attribute operators does not only extend the path with the position of the child or attribute that satisfied the subquery. Additionally, it appends the constraints that the evaluation of the algebraic subexpression created to the constraint store of the path prefix. While doing so however, the consistency of the constraints must be verified. It is possible that the constraints generated by the algebraic subexpression are incompatible with the constraints of the path prefix. In this case, the path prefix cannot be extended with the position associated with the incompatible constraint store, and the extension is dropped.

The descendant operator performs a similar combination of constraint stores. However, there is no filter condition compatibility to check and no path extension to perform when executing a descendant operator.

Definition 25 (Descendant Operator) *Let* $S : \mathit{Set}(\mathit{Element} \times \mathit{Position} \times \mathit{Set}(\mathit{Path}))$. *The descendant operator is*

$$\begin{aligned} \mathit{Desc}[f] &= \{(\mathit{node}, \mathit{pos}, \mathit{paths}') \mid (\mathit{node}, \mathit{pos}, \mathit{paths}) \in S, \\ &\quad S_{sub} = f\{(d, (0,0), \{\mathit{TruePath}\}) \mid d \in \mathit{descendants}(\mathit{node})\}, \\ &\quad \mathit{paths}' = \mathit{extend}_{desc}(\mathit{paths}, S_{sub})\} \\ \\ \mathit{extend}_{desc}(\mathit{paths}, S_{sub}) &= \{(p, \mathit{store}') \mid (p, \mathit{store}) \in \mathit{paths}, \\ &\quad (d, (0,0), \mathit{paths}_{sub}) \in S_{sub}, (p_{sub}, \mathit{store}_{sub}) \in S_{sub}, \\ &\quad \mathit{store}' = \mathit{verify}(\mathit{store}, \mathit{store}_{sub}), \mathit{store}' \neq \bot\} \end{aligned}$$

The function *bisim* computes the bisimulation between two data terms. The specification is given as procedural pseudocode (see code page 67).

```
BISIMULATION(value_1, value_2)
 1  switch
 2    case value_1 is String and value_2 is String :
 3          return value_1 = value_2
 4    case value_1 is String or value_2 is String :
 5          return False
 6    case not type(value_1) = type(value_2) :
 7          return False
 8    case type(value_1) = "text" :
 9          return value(value_1) = value(value_2)
10    case not (namespace(value_1) = namespace(value_2)
11               and label(value_1) = label(value_2)
12               and order(value_1) = order(value_2)) :
13          return False
14    case default :
15          elemlist_1 ← elements(value_1)
16          elemlist_2 ← elements(value_2)
17          attlist_1 ← attributes(value_1)
18          attlist_2 ← attributes(value_2)
19          if not size(attlist_1) = size(attlist_2)
20            then return False
21          if not size(elemlist_1) = size(elemlist_2)
22            then return False
23          for a_1 in attlist_1
24          do matched ← False
25             for a_2 in attlist_2
26             do if (namespace(a_1) = namespace(a_2)
27                        and label(a_1) = label(a_2)
28                        and order(a_1) = order(a_2))
29                   then attlist_2 ← attlist_2 \ {a_2}
30                        matched ← True
31                        break
32             if not matched
33               then return False
34             if ordered(value_1)
35               then for i ← 1 to size(elemlist_1)
36                    do if not BISIMULATION(elemlist_1[i], elemlist_2[i])
37                         then return False
38               else  for e_1 in elemlist_1
39                    do matched ← False
40                       for e_2 in elemlist_2
41                       do if BISIMULATION(e_1, e_2)
42                            then elemlist_2 ← elemlist_2 \ {e_2}
43                                 matched ← True
44                                 break
45                       if not matched
46                         then return False
47          return True
```

The consistency of constraint stores is covered with these definitions.

Since the domain changed, the clear operator must be redefined to keep the constraint stores while removing the path information.

Definition 26 (Clear Operator) *Let* $S : Set(Element \times Position \times Set(Path))$. *The clear operator is*

$$\begin{aligned} Clear(S) &= \{(node, pos, paths') \mid (node, pos, paths) \in S, \\ &\quad paths' = \{(\emptyset, store) \mid (path, store) \in paths\}\} \end{aligned}$$

The operators that introduce constraints are still needed. The idea of the constraint introducing constraints is straightforward: a bind operator binds a node or a node feature to a variable and introduces this binding in every constraint store of all actual solution contexts. Also, consistency verification must be performed, since the bind operators might be outside of any map operator (all map operators perform consistency verifications while extending paths). Another reason for the early consistency verification is to reduce the constraint stores to false and remove them as soon as possible.

Definition 27 (Bind, Value Bind and Position Bind Operators) *Let* $S : Set(Element \times Position \times Set(Path))$, *let* X *be a variable. The bind operator is*

$$\begin{aligned} Bind(X)(S) &= \{(el, pos, paths') \mid (el, pos, paths) \in S, \\ &\quad paths' = add(paths, (X \preceq el))\} \end{aligned}$$

The function add is

$$add(paths, C) = \{(path, store') \mid (path, store) \in paths, store' = verify(store, C), store' \neq \bot\}$$

The value binding operator β *takes a function* $f : Node \to T$ *that maps a node to a feature of the node and a variable* X. *Let* $S : Set(Element \times Position \times Set(Path))$ *(or let* $S : Set(Attribute \times Position \times Set(Path))$ *respectively for attribute queries).*

The value binding operator is

$$\begin{aligned} \beta(X = f)(S) &= \{(node, pos, paths') \mid (node, pos, paths) \in S, v = f(node), \\ &\quad paths' = add(paths, (X \preceq v))\} \end{aligned}$$

The position binding operator is defined for elements only: Let $S : Set(Element \times Position \times Set(Path))$.

$$\begin{aligned} \beta^{pos}(X)(S) &= \{(el, (i, j), paths') \mid (el, (i, j), paths) \in S, \\ &\quad paths' = add(paths, (X \preceq j))\} \end{aligned}$$

Evaluation of Algebra Expressions. The eval function must change since constraint stores and variables were introduced. The evaluation result for a given algebraic expression and a given set of nodes is a set of constraint stores. Each constraint store is the result of a possible matching of the query in a data term of the input.

Definition 28 (Evaluation) *The evaluation of an algebraic expression* f *with the input elements* E *is*

$$\begin{aligned} eval &: (Domain \to Domain) \to Set(Element) \to Set(Store) \\ eval\ f\ E &= \{store \mid (el, pos, paths) \in f\{(el, (0, 0), \{\emptyset\}) \mid el \in E\}, \\ &\quad (path, store) \in paths, store \neq \bot\} \end{aligned}$$

Translation Scheme. As the query term grammar defines, a variable may occur everywhere a label or regular expression may occur, that is, as namespace or label of a term and as namespace, label or value of an attribute. Furthermore, a variables may occur in as-patterns, everywhere query terms occur, and as position variable.

The translation scheme Tr_{val} must be extended with cases for variables since they may appear everywhere a value may occur.

$$Tr_{val}(f, \texttt{"var" label }) \quad = \quad \beta(label = f)$$

Furthermore, the translation scheme Tr_{term} is extended with definitions for the cases `"var" label` and recursive cases for `"var" label "as" <query-subterm>` and `"position" "var" label <query-subterm>`.

$$\begin{aligned} Tr_{term}(\texttt{"var" label}) &= Bind(label) \\ Tr_{term}(\texttt{"var" label "as" <query-subterm>}) &= Bind(label) \\ &\quad Tr_{term}(\texttt{<query-subterm>}) \\ Tr_{term}(\texttt{"position" "var" label <query-subterm>}) &= \beta^{pos}(label) \\ &\quad Tr_{term}(\texttt{<query-subterm>}) \end{aligned}$$

Example 16 (Query Translation and Evaluation) *Assume the query term*

```
f[[ var X, desc var X ]]
```

to query the data term

```
f[ a, b, c, g[ b, c, d ] ]
```

The translation of the query leads to the algebraic expression

$$\begin{aligned} &\sigma(type = \text{“}tree\text{”}) \\ &\sigma(label = \text{“}\texttt{f}\text{”}) \\ &\sigma(ordered = True) \\ &\sigma(arity \geq 2) \\ &Child(1, pos \leq -1)[Bind(X)] \\ &Child(2, pos > max)[Desc[Bind(X)]] \end{aligned}$$

The evaluation starts with the first child operator. The set of solution contexts that will be the input to the subexpression $Bind(X)$ is

$$S_{sub} = \{(\texttt{a}, (1,1), \{(\emptyset, \emptyset)\}), (\texttt{b}, (1,2), \{(\emptyset, \emptyset)\}), (\texttt{c}, (1,3), \{(\emptyset, \emptyset)\})\}$$

The last child is missing in S_{sub} because it does not satisfy the filter condition. Executing the bind operator yields the following set

$$Bind(X)(S_{sub}) = \{(\texttt{a}, (1,1), \{(\emptyset, \{X \preceq \texttt{a}\})\}), (\texttt{b}, (1,2), \{(\emptyset, \{X \preceq \texttt{b}\})\}), (\texttt{c}, (1,3), \{(\emptyset, \{X \preceq \texttt{c}\})\})\}$$

Extending the path set of the input solution contexts of the child operator $Child(1, pos \leq -1)[Bind(X)]$ *gives*

$$Child(1, pos \leq -1)[Bind(X)](S) =$$
$$\{(\texttt{f[a, b, c, g[b, c, d]]}, (0,0), \left\{ \begin{array}{l} (\{(1,1)\}, \{X \preceq \texttt{a}\}), \\ (\{(1,2)\}, \{X \preceq \texttt{b}\}), \\ (\{(1,3)\}, \{X \preceq \texttt{c}\}) \end{array} \right\})\}$$

Consistency verification of the variable bindings is not necessary, since each path contained only one variable binding and the verification rules apply only for two variable bindings of the same variable. The subexpression of the second child operator $Child(2, pos > max)[Desc[Bind(X)]]$ *is applied to the following set*

$$S_{sub} = \{(\texttt{b}, (2,2), \{(\emptyset, \emptyset)\}), (\texttt{c}, (2,3), \{(\emptyset, \emptyset)\}), (\texttt{g[b, c, d]}, (2,4), \{(\emptyset, \emptyset)\})\}$$

The first child is missing, because it does not satisfy the filter condition with any path. The descendant operator $Desc[Bind(X)]$ *will be applied to this set. Successively, each element of* S_{sub} *is chosen, the descendants are computed and the subexpression* $Bind(X)$ *is applied to the set of all descendants. Consider the third element of the set* S_{sub}*. The set of solution contexts to which* $Bind(X)$ *is applied is*

$$S_{desc} = \left\{ \begin{array}{ll} (\texttt{b}, (0,0), \{(\emptyset, \emptyset)\}), & (\texttt{c}, (0,0), \{(\emptyset, \emptyset)\}), \\ (\texttt{d}, (0,0), \{(\emptyset, \emptyset)\}), & (\texttt{g[b, c, d]}, (0,0), \{(\emptyset, \emptyset)\}) \end{array} \right\}$$

Applying the bind operator yields:

$$Bind(X)(S_{desc}) = \left\{ \begin{array}{l} (\texttt{b}, (0,0), \{(\emptyset, \{X \preceq \texttt{b}\})\}), \\ (\texttt{c}, (0,0), \{(\emptyset, \{X \preceq \texttt{c}\})\}), \\ (\texttt{d}, (0,0), \{(\emptyset, \{X \preceq \texttt{d}\})\}), \\ (\texttt{g[b, c, d]}, (0,0), \{(\emptyset, \{X \preceq \texttt{g[b, c, d]}\})\}) \end{array} \right\}$$

The paths of the initial solution context are extended with these results. The verification again does not apply.

$$sc_3 = (\texttt{g[b, c, d]}, (2,4), \left\{ \begin{array}{l} (\emptyset, \{X \preceq \texttt{b}\}), \\ (\emptyset, \{X \preceq \texttt{c}\}), \\ (\emptyset, \{X \preceq \texttt{d}\}), \\ (\emptyset, \{X \preceq \texttt{g[b, c, d]}\}) \end{array} \right\})$$

The desc operator applies to the other two children similarly. The overall result of the descendant operator is

$$Desc[Bind(X)](S_{sub}) = \left\{ \begin{array}{l} (\texttt{b}, (2,2), \{(\emptyset, \{X \preceq \texttt{b}\})\}), \\ (\texttt{c}, (2,3), \{(\emptyset, X \preceq \texttt{c})\}), \\ (\texttt{g[b, c, d]}, (2,4), \left\{ \begin{array}{l} (\emptyset, \{X \preceq \texttt{b}\}), \\ (\emptyset, \{X \preceq \texttt{c}\}), \\ (\emptyset, \{X \preceq \texttt{d}\}), \\ (\emptyset, \{X \preceq \texttt{g[b, c, d]}\}) \end{array} \right\}) \end{array} \right\}$$

The child operator combines these subsolutions with the set of solution contexts from the previous child operator.

$$S = \{(\texttt{f[a, b, c, g[b, c, d]]}, (0,0), \left\{ \begin{array}{l} (\{(1,1)\}, \{X \preceq \texttt{a}\}), \\ (\{(1,2)\}, \{X \preceq \texttt{b}\}), \\ (\{(1,3)\}, \{X \preceq \texttt{c}\}) \end{array} \right\})\}$$

For each path in the single solution context in S, the extend function is applied with S_{sub}. The compatible function, which is the third parameter passed to the extend function, verifies injectivity and the filter condition $pos > max$ for each combination of subsolution and path. First, assume that the following path is chosen:

$$(\{(1,1)\}, \{X \preceq \texttt{a}\})$$

A path and position is selected from the subsolution, so start with the position

$$pos_{sub} = (2,2) \text{ and } store_{sub} = \{X \preceq \texttt{b}\}.$$

The path extension is compatible with the initial path, and the verification of the constraint store returns $\bot$. Since the new constraint store is inconsistent, this path is not added to the set of solutions.

Position	**Substore**	**Extended Path**	**Extended Store**
$(2,2)$	$\{X \preceq \texttt{b}\}$	$\{(1,1),(2,2)\}$	$\bot$
$(2,3)$	$\{X \preceq \texttt{c}\}$	$\{(1,1),(2,3)\}$	$\bot$
$(2,4)$	$\{X \preceq \texttt{b}\}$	$\{(1,1),(2,4)\}$	$\bot$
	$\{X \preceq \texttt{c}\}$		$\bot$
	$\{X \preceq \texttt{d}\}$		$\bot$
	$\{X \preceq \texttt{g[b, c, d]}\}$		$\bot$

As the table above shows, all other combinations of constraints lead to inconsistent constraint stores. The map operator hence drops the path $(\{(1,1)\}, \{X \preceq \texttt{a}\})$. The next considered path is

$$(\{(1,2)\}, \{X \preceq \texttt{b}\})$$

The combination of the subconstraints with this path is as follows.

Position	**Substore**	**Extended Path**	**Extended Store**
$(2,2)$	$\{X \preceq \texttt{b}\}$	*forbidden due to injectivity*	
$(2,3)$	$\{X \preceq \texttt{c}\}$	$\{(1,2),(2,3)\}$	$\bot$
$(2,4)$	$\{X \preceq \texttt{b}\}$	$\{(1,2),(2,4)\}$	$\{X \preceq \texttt{b}\}$
	$\{X \preceq \texttt{c}\}$		$\bot$
	$\{X \preceq \texttt{d}\}$		$\bot$
	$\{X \preceq \texttt{g[b, c, d]}\}$		$\bot$

The combination of the path

$$(\{(1,3)\}, \{X \preceq \texttt{c}\})$$

with the subsolutions is similar to the above.

Position	Substore	Extended Path	Extended Store
$(2,2)$	$\{X \preceq \texttt{b}\}$	*forbidden due to monotonicity*	
$(2,3)$	$\{X \preceq \texttt{c}\}$	*forbidden due to injectivity*	
$(2,4)$	$\{X \preceq \texttt{b}\}$	$\{(1,3),(2,4)\}$	$\bot$
	$\{X \preceq \texttt{c}\}$		$\{X \preceq \texttt{c}\}$
	$\{X \preceq \texttt{d}\}$		$\bot$
	$\{X \preceq \texttt{g[b, c, d]}\}$		$\bot$

The solution of the whole child operator is hence

$$\begin{aligned} &\mathit{Child}(2, \mathit{pos} > \mathit{max})[\mathit{Desc}[\mathit{Bind}(X)]](S) = \\ &\quad \{(\texttt{f[a, b, c, g[b, c, d]]}, (0,0), \left\{ \begin{array}{l} (\{(1,2),(2,4)\}, \{X \preceq \texttt{b}\}), \\ (\{(1,3),(2,4)\}, \{X \preceq \texttt{c}\}) \end{array} \right\})\} \end{aligned}$$

The query is successful and the resulting constraint stores are

$$\{X \preceq \texttt{b}\} \textit{ and } \{X \preceq \texttt{c}\}$$

Extending the algebra with variables requires a substantial extension of the base query algebra. It is necessary to handle variable consistency in the bind and map operators. The map operators are furthermore extended to combine the variable bindings emerging from the application of the respective algebraic subexpression with the already existing solutions in the actual context node.

These two aspects, consistency verification and constraint store combination, are the extensions of the basic algebra necessary to handle variables in the query terms.

3.3.5 Extending the Algebra for Negation

Negation is a common tool in natural and programming languages. Xcerpt offers negation constructs within query terms, specifying that terms having certain subterms are not part of the solution. Bare in mind however that subterm negation is only sensible in partial query term lists. For this reason, the operators introduced for negation do not need to cope with surjectivity requirements of the mappings.

But the negation constructs of Xcerpt require several groupings of solutions within the algebra, which is discussed before the operators are defined.

Negation and Solution Grouping. Adding term negation to Xcerpt is powerful and complex. The definition of the constraint calculus rules are already very sophisticated, and the same holds for the algebraic operators that handle negation. Subterm negation even make certain grouping structures necessary in the algebra.

So far, each context node is associated to a set of constraint stores. It would be possible to associate each node with only one store, and expand each constraint store in a separate solution. However, the negation makes a kind of grouping necessary when all constraint stores associated with a context node must be negated. Negating a disjunctive normal form (DNF) transforms it into a conjunctive normal form (CNF). Computing again the DNF from the negated formula requires to expand all disjuncts of the CNF (which were the constraint

stores previously) with each other. In order to do so, we need to access all conjuncts of the previous DNF. The constraint stores which represents the DNF conjuncts are grouped together for this reason.

Above this, several negated path extensions must be grouped together with one positive path. Remember the term elimination rules with negated subterms (section 2.2.3) . For a given mapping π of the positive subterms, all possible extensions $\pi' \in E(\pi)$ of the mapping π to the negative terms are considered and built into one conjunction:

$$\bigwedge_{\pi' \in E(\pi)} \bigwedge_{(q_-, c) \in \pi'} q_- \preceq_{su} c$$

This means that if one extension fails, the mapping of the positive terms fails. In order to handle negations in the Xcerpt query algebra, all extensions of a given positive mapping must be computed before combining the constraints from these mappings into one conjunction.

For the query algebra domain, a set of path extensions must be grouped together with a positive path. This means that a solution path is an element of type $Set(Position) \times Set(Set(Position))$. The left position set represents the positive path, and the right set of position sets represents the extensions of the positive path. Since the constraints from the path extensions can only be added after matching all map operators, the path extension must also contain a set of constraints. The path type hence becomes $Set(Position) \times Store \times Set(Set(Position) \times Store)$.

At the end of the matching, the negated constraint part must be concatenated with the positive part in order to get the final result. The combine operator is introduced for this task.

Operators and Domain for Negation. First, negated variable bindings are added to the algebra constraints.

Definition 29 (Extension of the constraint store) *Let $el : Element$, $str : String$ and X be a variable. The set CS_{atomic} is the smallest set such that*

- $X \preceq el \in CS_{atomic}$
- $not\ X \preceq el \in CS_{atomic}$
- $not\ not\ X \preceq el \in CS_{atomic}$
- $X \preceq str \in CS_{atomic}$
- $not\ X \preceq str \in CS_{atomic}$
- $not\ not\ X \preceq str \in CS_{atomic}$

As discussed above, the algebra domain is extended so that a set of path extensions is associated to each positive path.

Definition 30 (Algebra Domain Extension) *Let*

$$PathItem = Set(Position) \times Store$$

We redefine the type Path to be

$$Path = PathItem \times Set(PathItem)$$

The true path used in the map operators is

$$TruePath = (\emptyset, \emptyset, \{(\emptyset, \emptyset)\})$$

The new domain uses the redefined Path and stays unchanged otherwise.

$$\begin{aligned} Domain &= Set(Element \times Position \times Set(Path)) \\ AttrDomain &= Set(Attribute \times Position \times Set(Path)) \end{aligned}$$

Name	**Notation**	**Short Description**
Negated Child	$Child_{neg}(i, f_1, \ldots, f_n)[f_{sub}]$	Selects all children satisfying f_{sub} and negates the result.
Negated Attribute	$Attr_{neg}(i, f_1, \ldots, f_n)[f_{sub}]$	Selects all attributes satisfying f_{sub} and negates the results.
Combine negated	$Comb$	Combines the stores of the path with the stores of the extensions.

Table 4: The Operators for Negation

The negated child operator must create all possible extensions to a given positive path. However, it must be verified that the positive paths together with the extension still satisfy the injectivity property (and monotonicity property if the query list is partial and ordered). It may happen that given a positive path π, a set of extensions E and an assignment $m = (q \mapsto d)$, it is possible to extend π by the assignment without violating the imposed path constraints. At the same time, there might be an extension in E incompatible with the assignment m.

Example 17 (Extension Incompatibility) *Take the mapping*

$$\pi = \{q_1 \mapsto d_1\}$$

with the set of extensions

$$\{\{q_2 \mapsto d_2\}, \{q_2 \mapsto d_3\}, \emptyset\}$$

Assume furthermore that the mappings must be injective. Obviously, q_1 is a positive query term and q_2 is a negative query term, i.e. it has the form `without` *q for some q. Take a positive query term q_3 that maps to d_3. It is possible to extend the positive mapping with the assignment*

$$m = (q_3 \mapsto d_3)$$

The extended positive mapping is then

$$\pi = \{q_1 \mapsto d_1, q_3 \mapsto d_3\}$$

However, the mapping extension $\{q_2 \mapsto d_3\}$ is incompatible with the extended positive mapping, since

$$\{q_1 \mapsto d_1, q_2 \mapsto d_3, q_3 \mapsto d_3\}$$

is no injective mapping.

In such a case, the incompatible extensions to the negative query terms must be dropped since the positive query terms have priority (see section 2.2.3). This implies the following modification for the positive map operators: after testing (as before) whether the positive path can be extended with a given position, a new path is created, and all incompatible path extensions are removed.

In the new definition of the *extend* function, the incompatible paths are removed by calling the new function *cut*.

Definition 31 (Positive Extend Function)

$$\begin{aligned} extend \quad & (paths, S_{sub}, compatible) = \{(path \cup \{pos\}, store', Neg') \mid \\ & (path, store, Neg) \in paths, (c, pos, paths_{sub}) \in S_{sub}, \\ & (path_{sub}, store_{sub}, Neg_{sub}) \in paths_{sub}, \\ & compatible(path, pos), store' = verify(store, store_{sub}), \\ & Neg' = cut(Neg, pos, compatible)\} \end{aligned}$$

$$\begin{aligned} cut \quad & (Neg, pos, compatible) = \{(path, store) \mid (path, store) \in Neg, \\ & compatible(path, pos)\} \end{aligned}$$

The negated child operator creates all possible extensions containing a mapping of the query term the operator stands for. Since it is as well possible to drop the negated row, the new extensions are added to the old extensions instead of replacing them. The operator "combine negated" will later create one single solution for a given positive mapping and all extensions.

Before defining the negated operators, the negate function that negates a set of constraint stores is defined. A set of constraint stores S represent the disjunctive normal form $\bigvee_{s \in S} \bigwedge_{c \in s} c$. The negate operator returns the disjunctive normal form of $\neg \bigvee_{s \in S} \bigwedge_{c \in s} c$ by first creating the negation normal form $\bigwedge_{s \in S} \bigvee_{c \in s} \neg c$ and then expanding each disjunction.

The negate function handles the following border cases appropriately: if a substitution set $s \in S$ is empty, it is trivially true. Hence, it is false in the negation normal form and the whole formula is then equivalent to false. If a substitution set $s \in S$ is false, the store is removed. The negation of the false store is true and true is the neutral element in a conjunction. If all stores are true, *negate* returns hence one empty constraint store.

Definition 32 (Negate Function)

$$negate(S) = combine(\{negStore(s) \mid s \in S, s \neq \bot\})$$

$$negStore(store) = \begin{cases} \bot & \text{if } store = \emptyset \\ \{simplify(not\ c) \mid c \in store\} & \text{otherwise.} \end{cases}$$

$$simplify(c) = \begin{cases} not\ c' & \text{if } c = not\ not\ not\ c' \\ c & \text{otherwise.} \end{cases}$$

$$combine(S) = \begin{cases} \{\emptyset\} & \text{if } S = \emptyset \\ \{\bot\} & \text{if } \bot \in S \\ \{\{c\} \cup s_2 \mid c \in s_1, s_2 \in combine(S')\} & \text{if } S = \{s_1\} \cup S' \end{cases}$$

The *combine* function chooses each simulation constraint c out of one actually processed constraint store s_1, and appends each such simulation constraint c to each constraint store

s_2 that results from the recursive call of *combine* on the set $S' = S \setminus \{s_1\}$. If there are n simulation constraints in s_1 and m constraint stores in $combine(S')$, the total number of created constraint stores is $n \cdot m$.

With the negate function, the negated child and negated attribute operators can be defined.

Definition 33 (Negated Child) *Let* $S : Set(Element \times Position \times Set(Path)), i \in Integer$, *Let* $f_{sub} : Set(Element \times Position \times Set(Path)) \rightarrow Set(Element \times Position \times Set(Path)), f_1, \ldots f_n$ *be filter conditions.*

$$
\begin{aligned}
Child_{neg} \quad & (i, f_1, \ldots, f_n)[f_{sub}](S) = \{(node, pos, paths') \mid (node, pos, paths) \in S, \\
& comp = compatible(size(elements(node)), f_1, \ldots, f_n), \\
& S_{sub} = f_{sub}\{(c, (i,j), \{TruePath\}) \mid c = elements(node)[j], \\
& \qquad \exists p \in paths.\ comp(p, (i,j))\}, \\
& S_{neg} = negate_{sub}(S_{sub})), \\
& paths' = iterate\ (extend_{neg}(comp))\ paths\ S_{neg}\}
\end{aligned}
$$

$$
\begin{aligned}
extend_{neg} \quad & (comp)(paths, (pos, stores_{neg})) = \{(path, store, E \cup E') \mid \\
& (path, store, E) \in paths, \\
& comp(path, pos) \Rightarrow E' = extensions(comp)(E, stores_{neg}, pos), \\
& \neg comp(path, pos) \Rightarrow E' = \emptyset\}
\end{aligned}
$$

$$
\begin{aligned}
extensions \quad & (compatible)(E, stores, (i,j)) = \{(path_n \cup \{(i,j)\}, store'_n) \mid \\
& (path_n, store_n) \in E, i \notin dom(path_n), compatible(path_n, (i,j)), \\
& store_{sub} \in stores, store'_n = verify(store_n, store_{sub})\}
\end{aligned}
$$

$$
\begin{aligned}
negate_{sub}(S) \quad & = \{(pos, stores_{neg})) \mid (c, pos, paths) \in S, \\
& stores_{neg} = negate(\{store \mid (path, store, E) \in paths\})\}\}
\end{aligned}
$$

Note that for one context node, the negated child operator iterates over all subsolutions instead of choosing one subsolution and processing the combination. This means that given a context node, all possible extensions are computed by iterating over the subsolutions. If a subsolution is compatible with the path, all possible path extensions with this subsolution are computed and added to the set of path extensions.

Note that the compatibility condition is verified with both the positive path and the path extension. In this way, it is ensured that each positive path together with each extension respects the imposed filter conditions.

The negated attribute operator is very similar to the negated child operator, as it uses the same $extend_{neg}$ function.

Definition 34 (Negated Attribute) *Let* $S : Set(Element \times Position \times Set(Path)), i \in Integer$. *Let* $f : Set(Attribute \times Position \times Set(Path)) \rightarrow Set(Attribute \times Position \times Set(Path))$.

$$
\begin{aligned}
Attr_{neg} \quad & (i)[f](S) = \{(node, pos, paths') \mid (node, pos, paths) \in S, \\
& comp = compatible(0), \\
& S_{sub} = f\{(c, (i,j), \{TruePath\}) \mid c = attributes(node)[j], \\
& \qquad \exists p \in paths.\ comp(p, (i,j))\}, \\
& paths' = iterate\ (extend_{neg}(comp))\ paths\ S_{sub}\}
\end{aligned}
$$

The combine negated operator takes all extensions of a given positive path and appends the constraint stores associated with each extension to the positive constraint store. The combine negated operator creates the conjunction of all stores of all possible extensions, as the calculus rule "decomposition with negation" requires (see section 2.2.3).

Definition 35 (Combine Negated Operator) *Let* $S : Set(Element \times Position \times Set(Path))$.

$$\begin{aligned} Comb(S) &= \{(node, pos, paths') \mid (node, pos, paths) \in S, \\ &\quad paths' = combine(paths)\} \\ combine(paths) &= \{(path, store', E) \mid (path, store, E) \in paths, \\ &\quad iterate\ verify\ store\ (map\ \pi_2 \quad E)\} \\ \pi_2(path, store) &= store \end{aligned}$$

Definition 36 (Clear Operator) *Let* $S : Set(Element \times Position \times Set(Path))$.

$$\begin{aligned} Clear(S) &= \{(node, pos, paths') \mid (node, pos, paths) \in S, \\ &\quad paths' = \{(\emptyset, store, \{(\emptyset, \emptyset)\}) \mid (path, store, E) \in paths\}\} \end{aligned}$$

The verify function must also be extended to handle negated constraints:

Definition 37 (Extension of Consistency Verification) *The function* $verify$ *is*

$$verify(s_1, s_2) = \begin{cases} \bot & \text{if } s_1 = \bot \text{ or } s_2 = \bot \\ s_1 \cup s_2 & \text{if} \quad \forall X \preceq n_1 X \preceq n_2 \in s_1 \cup s_2.\ bisim(n_1, n_2) \\ & \text{and } \forall not\ X \preceq n_1, X \preceq n_2 \in s_1 \cup s_2.\ \neg bisim(n_1, n_2) \\ & \text{and } \forall not\ not\ X \preceq n_1, X \preceq n_2 \in s_1 \cup s_2.\ bisim(n_1, n_2) \\ & \text{and } \forall not\ not\ X \preceq n_1, not\ X \preceq n_2 \in s_1 \cup s_2.\ \neg bisim(n_1, n_2) \\ \bot & otherwise. \end{cases}$$

Informal Translation Scheme. The whole definition for query term lists must be repeated to give a formal translation scheme for queries with negation. To avoid this, only an informal description of the translation is given here. The complete and formal translation scheme is found in appendix B.

To translate negated query terms, the child and attribute operators are replaced by their respective negated counterparts. Furthermore, only the number of positive query terms must be considered when imposing the arity restrictions on the parent node. For example a partial and unordered query term with four subterms imposed previously $\sigma(arity \geq 4)$. If one of these subterms is negative, the arity must only be greater than three, as the negative subterm may not have a match in order to make the query successful. Finally, a combine operator is added after all child operators and after all attribute operators also in order to combine the path extensions.

$$\begin{aligned} Tr_{term}(\texttt{"without" <query-subterm>}) &= Child_{neg}(i, f_1, \ldots, f_n)[Tr_{term}(\texttt{<query-subterm>})] \\ Tr_{att}(\texttt{"without" <attribute>}) &= Attr_{neg}(i, f_1, \ldots, f_n)[Tr_{att}(\texttt{<attribute>})] \end{aligned}$$

If the query term list (or query attribute list for the same) contains negated query terms, a combine negated operator is appended at the end of the term or attribute list translation.

In partial and ordered specifications, filter conditions of the form $(pos \leq -k)$ are introduced. In the presence of negated terms, the value k must be adjusted to be the number of following *positive* terms instead of the number of following terms.

Example 18 (Translation and Execution of Negated Queries) *Consider the query* $q =$ `f[[ var X, without var X]]`. *The translation into the algebra is as follows.*

$$\begin{array}{l}\sigma(type = \text{“}tree\text{”}) \\ \sigma(label = \text{“}\texttt{f}\text{”}) \\ \sigma(arity \geq 1) \\ \sigma(ordered = True) \\ Child(1)[Bind(X)] \\ Child_{neg}(2, pos > max)[Bind(X)] \\ Comb\end{array}$$

The algebraic query expression will be evaluated with the data term `f[a, b, b]`. *The evaluation starts with the set*

$$S = \{(\texttt{f[a, b, b]}, (0,0), \{(\emptyset, \{(\emptyset, \emptyset)\})\})\}$$

Since the context node satisfies the three selection conditions, the set S *is still the same after these selections.*

$$\sigma(arity \geq 1) \circ \sigma(label = \text{“}\texttt{f}\text{”}) \circ \sigma(type = \text{“}tree\text{”})(S) = S$$

The first child operator is positive, so we give the solution right away here.

$$Child(1)[Bind(X)]\ \{(\texttt{f[a, b, b]}, (0,0), \{(\emptyset, \{(\emptyset, \emptyset)\})\})\} = \{(\texttt{f[a, b, b]}, (0,0), paths)\}$$

Where

$$paths = \left\{ \begin{array}{l} (\{(1,1)\}, \{X \preceq a\}, \{(\emptyset, \emptyset)\}), \\ (\{(1,2)\}, \{X \preceq b\}, \{(\emptyset, \emptyset)\}), \\ (\{(1,3)\}, \{X \preceq b\}, \{(\emptyset, \emptyset)\}) \end{array} \right\}$$

The second child operator is negative, which is the new part to demonstrate. The evaluation of the subexpression of the negated child operator returns the set S_{sub}*:*

$$S_{sub} = \left\{ \begin{array}{l} (\texttt{b}, (2,2), \{(\emptyset, \{X \preceq b\}, \{(\emptyset, \emptyset)\})\}), \\ (\texttt{b}, (2,3), \{(\emptyset, \{X \preceq b\}, \{(\emptyset, \emptyset)\})\}) \end{array} \right\}$$

The negated child operator combines the results S_{sub} *with the paths of* S*. The next step to execute in the evaluation of the negated child operator is to negate the solutions* S_{sub}*. Since both solution contexts contain only one path with one constraint in the store, the set* S_{neg} *is the simple negation of* S_{sub}

$$S_{neg} = \{((2,2), \{\{not\ \ X \preceq b\}\}), ((2,3), \{\{not\ \ X \preceq b\}\})\}$$

$$iterate\ extend_{neg}(comp)\ paths\ S_{neg}$$

The next step is to iterate over S_{neg}. The iterate function chooses one element out of S_{sub}, so assume it took

$$s = ((2,2), \{\{not\ X \preceq b\}\})$$

and applies the $extend_{neg}$ function to the pair $(path, solctx)$. The formal parameters of the function are bound as follows.

$$\begin{array}{rcl} comp & = & compatible(3, pos > opmax) \\ paths & = & \left\{ \begin{array}{l} (\{(1,1)\}, \{X \preceq a\}, \{(\emptyset,\emptyset)\}), \\ (\{(1,2)\}, \{X \preceq b\}, \{(\emptyset,\emptyset)\}), \\ (\{(1,3)\}, \{X \preceq b\}, \{(\emptyset,\emptyset)\}) \end{array} \right\} \\ pos & = & (2,2) \\ stores_{neg} & = & \{\{not\ X \preceq b\}\}) \end{array}$$

To evaluate the $extend_{neg}$ function, the definition must be applied to each element of the set paths. Assume the evaluation starts with the path

$$p_1 = (\{(1,1)\}, \{X \preceq a\}, \{(\emptyset,\emptyset)\})$$

Since

$$(\{(1,1)\}, (2,2)) \in compatible(3, pos > max),$$

the set E' must be

$$E' = extensions(compatible(3, pos > max))(\{(\emptyset,\emptyset)\}, \{\{not\ X \preceq b\}\}, (2,2))$$

In order to compute the extensions, an existing extension is chosen from the set $\{(\emptyset,\emptyset)\}$. Since there is only one extension, the extension $(\emptyset,\emptyset)$ is selected. This extension is augmented to $(\{(2,2)\}, \{not\ X \preceq b\})$. The extensions function returns therefore

$$E' = \{(\{2,2\}, \{not\ X \preceq b\})\}.$$

These extensions are added to the set of extensions of the path p_1, so that p_1 becomes

$$p_1 = (\{(1,1)\}, \{X \preceq a\}, \{(\emptyset,\emptyset), (\{(2,2)\}, \{not\ X \preceq b\})\})$$

Continuing with the second path

$$p_2 = (\{(1,2)\}, \{X \preceq b\}, \{(\emptyset,\emptyset)\}),$$

it holds that

$$(\{(1,2)\}, (2,2)) \notin compatible(3, pos > max),$$

and the set E' must be hence $\emptyset$. The second paths p_2 is unchanged. The same holds for the third path p_3, since again

$$(\{(1,3)\}, (2,2)) \notin compatible(3, pos > max).$$

After the first pass of iterate, the path set is $\{p_1, p_2, p_3\}$ with

$$\begin{array}{rcl} p_1 & = & (\{(1,1)\}, \{X \preceq a\}, \{(\emptyset,\emptyset), (\{(2,2)\}, \{not\ X \preceq b\})\}) \\ p_2 & = & (\{(1,2)\}, \{X \preceq b\}, \{(\emptyset,\emptyset)\}) \\ p_3 & = & (\{(1,3)\}, \{X \preceq b\}, \{(\emptyset,\emptyset)\}) \end{array}$$

The second pass of the iterate function takes the second negated solution, which is

$$((2,3),\{\{not\ X \preceq b\}\}).$$

Verifying the compatibility of these negated solutions with the three paths above, it holds that p_3 is still not compatible, since

$$(\{(1,3)\},(2,3)) \notin compatible(3, pos > max).$$

For this reason, p_3 stays as it is (E' must be again the empty set). The case is different for p_1 and p_2, since the position extension is compatible with the positive path. Consider p_2 first; since it is a carbon copy of the handling of p_1 for the first negated subsolution, the extension of p_2 is

$$p_2 = (\{(1,2)\},\{X \preceq b\},\{(\emptyset,\emptyset),(\{(2,3)\},\{not\ X \preceq b\})\})$$

The case is different for p_1, since the set of extensions contains two element. Consider the first element of the extension set, $(\emptyset,\emptyset)$. This element can be extended with the subsolution, since

$$(\emptyset,(2,3)) \in compatible(3, pos > max).$$

Considering however the second element of the extension set,

$$(\{(2,2)\},\{not\ X \preceq b\}),$$

it holds that the pair $(\{2,2\},(2,3))$ is in the relation $compatible(3, pos > max)$. But the extensions function verifies furthermore that $i \notin dom(path_n)$. In this case, it verifies $2 \notin dom(\{2,2\})$, and this does not hold. Hence, the extension $(\{(2,2),(2,3)\},\{not\ X \preceq b\})$ is not valid for p_1. Altogether, the resulting paths from the application of the iterate function are the following:

$$\begin{array}{lcl} p_1 & = & (\{(1,1)\},\{X \preceq a\},\{(\emptyset,\emptyset),(\{(2,2)\},\{not\ X \preceq b\}),(\{(2,3)\},\{not\ X \preceq b\})\}) \\ p_2 & = & (\{(1,2)\},\{X \preceq b\},\{(\emptyset,\emptyset),(\{(2,3)\},\{not\ X \preceq b\})\}) \\ p_3 & = & (\{(1,3)\},\{X \preceq b\},\{(\emptyset,\emptyset)\}) \end{array}$$

The result of the negated child operator is hence

$$\begin{array}{rcl} Child_{neg}(2, pos > max)[Bind(X)] & \{(\texttt{f[a, b, b]},(0,0),\left\{\begin{array}{l}(\{(1,1)\},\{X \preceq a\},\{(\emptyset,\emptyset)\}),\\(\{(1,2)\},\{X \preceq b\},\{(\emptyset,\emptyset)\}),\\(\{(1,3)\},\{X \preceq b\},\{(\emptyset,\emptyset)\})\end{array}\right\})\} \\ & = & (\texttt{f[a, b, b]},(0,0),\{p_1,p_2,p_3\}) \end{array}$$

with the three paths p_1, p_2 and p_3.

The final operator to apply, the combine operator, creates one single constraint store from the store associated with the positive paths and the stores associated with the path extensions. Every constraint store of the extension is placed into conjunction with the constraint store from the positive path. For the first path, the application of the combine operator leads to the constraint store

$$\{X \preceq a, not\ X \preceq b\},$$

which is obviously consistent. The second path however contains the constraints $X \preceq b$ *and not* $X \preceq b$*, which is inconsistent. Hence, this path fails. The third path contains no extension to negative terms so that it stays consistent. In the end, the following two solutions exist:*

$$\{X \preceq a, \mathit{not}\ X \preceq b\}\ \mathit{and}\ \{X \preceq b\}.$$

The last solution might be surprising at first, but is legal as this example shows and as the rules for comprehending `optional` *and* `without` *in section 2.3 specified.*

As discussed in the introduction of this section, extending the algebra for negation requires two kinds of groupings. First, the constraint stores that stem from one context node are grouped together so that it is possible to negate them if necessary. Second, the path extensions of a positive path prefix are grouped together. A path extension to the negative child operators is dropped whenever it becomes incompatible with a positive extension of the positive path prefix. After all map operators were computed, the combine operator combines the positive constraint store with all negative constraints from the path extensions. It is not possible to combine the path extensions with the positive path before the last positive child operator was evaluated, because the child operator may extend the positive path prefix and cancel negative path extensions. This is a very important fact that complicates the handling of negation, but is necessary to produce the abovementioned semantics described in 2.2.3 and 2.3. Since the negated query terms are handled completely, the following section focuses on the evaluation of optional query terms.

3.3.6 Extending the Algebra for Optional Terms

Xcerpts offers optional term queries as a sensible shorthand notation for disjunctions. Every query with an optional query part can be transformed into a disjunction of two queries as defined in the section 2.2.1. Even though the optional construct prohibits simple optimization through subterm reordering (see section 3.4.2), the native treatment of optional query parts presented in the following is more efficient than the simple evaluation scheme by explicit unfolding into disjunctions.

Bare in mind that the optional construct can occur within total query lists, unlike the without operator. Two different operators are needed for optional queries within total specifications and for partial specifications, as negation is only involved in partial query terms. When normalizing a total specification, the subterm is omitted instead of added with a without construct in the second generated query term.

The replacement of the optional construct in total and ordered specifications leads to query terms that have different arity but with positive subterms only. For example, the normalized form of the query `f[ a, optional a ]` is `or{ f[ a ], f[ a, a ] }`. The optional total child will generate both alternatives of dropping the subterm and taking it. Then, the prune optional children operator will remove all solutions that did not match all data subterms and hence violate the totality requirement. It is possible to handle optional subterms efficiently by earlier pruning of solutions with wrong arity, as discussed in section 4.3.1. However, this extended optimization was not incorporated into the algebra yet.

The definitions of the optional child and optional attribute can fortunately reuse previous operator definitions. They are shortcut notations for the union of two operators just as the optional construct is a shortcut for a disjunction of two queries.

Name	Notation	Short Description
Optional Total Child	$Child_{opt,tot}(i, f_1, \ldots, f_n)[f_{sub}]$	Collects all children satisfying f_{sub} and optionally adds the results to the path.
Optional Partial Child	$Child_{opt,part}(i, f_1, \ldots, f_n)[f_{sub}]$	Collects all children satisfying f_{sub} and optionally adds the results to the path.
Optional Total Attribute	$Attr_{opt,tot}(i)[f_{sub}]$	Collects all attributes satisfying f_{sub} and optionally adds the results to the path.
Optional Partial Attribute	$Attr_{opt,part}(i)[f_{sub}]$	Collects all attributes satisfying f_{sub} and optionally adds the results to the path.
Prune Optional Children	$Prune_{Opt}$	Removes the solutions with too short (positive) paths.
Prune Optional Attributes	$Prune_{Opt}$	Removes the solutions with too short (positive) paths.

Table 5: Operators for Optional Queries

Definition 38 (Optional Child and Attribute Operator) *Let* $S : Set(Element \times Position \times Set(Path)), f_{sub} : Set(Element \times Position \times Set(Path)) \rightarrow Set(Element \times Position \times Set(Path)), i : Integer, f_1, \ldots, f_n$ *be filter condition. The optional partial child and optional total child operators are*

$$Child_{opt,part}(i, f_1, \ldots, f_n)[f_{sub}](S) = Child(i, f_1, \ldots, f_n)[f_{sub}](S) \cup Child_{neg}(i, f_1, \ldots, f_n)[f_{sub}](S)$$

$$Child_{opt,tot}(i, f_1, \ldots, f_n)[f_{sub}](S) = Child(i, f_1, \ldots, f_n)[f_{sub}](S) \cup S$$

Let $f_{sub} : Set(Attribute \times Position \times Set(Path)) \rightarrow Set(Attribute \times Position \times Set(Path))$. *The optional partial attribute and optional total attribute operators are*

$$Attr_{opt,part}(i)[f_{sub}](S) = Attr(i)[f_{sub}](S) \cup Attr_{neg}(i)[f_{sub}](S)$$

$$Attr_{opt,tot}(i)[f_{sub}](S) = Attr(i)[f_{sub}](S) \cup S$$

The difference between both operators is that the partial operators take the negated results from f_{sub} while the total operator allows to "bypass" the operator and adds S to the set of results.

The prune optional operator is introduced at the end of a total specification list to remove solutions that are illegal due to the surjectivity requirement on the mapping.

Definition 39 (Prune Optional Operator) *Let* $S : Set(Element \times Position \times Set(Path))$.

$$\begin{aligned} Prune(S) = \{&(node, pos, paths') \mid (node, pos, paths) \in S, s = size(elements(node)), \\ &paths' = \{(path, store, E) \mid (path, store, E) \in paths, |path| = s\}\} \end{aligned}$$

$$\begin{aligned} Prune_{Attr}(S) = \{&(node, pos, paths') \mid (node, pos, paths) \in S, s = size(attributes(node)), \\ &paths' = \{(path, store, E) \mid (path, store, E) \in paths, |path| = s\}\} \end{aligned}$$

It is sufficient to verify the path length, because all paths are injective. Hence, the size of the paths corresponds to the size of the path range.

Informal Translation Scheme. The translation scheme presented here is as informal as the one for negated subterms. The precise and complete translation scheme can be found in appendix B. The child or attribute operator are replaced with the respective optional operator. The $Child_{opt,tot}$ operator is used as translation for optional terms in total specifications, $Child_{opt,part}$ for partial specifications and so on.

As for the translation with negated query terms, the arity conditions imposed on the parent element must be adjusted. In partial term queries, the arity is greater than the number of positive subterms, since optional and negated subterms do not need to match. For total subterm specifications, the arity must be greater than or equal to the number of positive terms and less than or equal to the number of positive and optional terms together. Terms with less children (or attributes) than the number of positive query terms cannot satisfy the query term due to the injectivity constraint, while terms with more children than optional and positive terms cannot satisfy the query due to the surjectivity constraint.

Furthermore, the filter conditions ($pos \leq -k$) introduced in partial and ordered specifications must be changed. In the presence of optional and negated terms, the number of following *positive* (that is, non-optional and non-negated) terms must be used as value for k instead of the number of following terms.

Example 19 (Total Subterm Specification and Optional) *Assume the query* `f[optional a, b]` *to be matched with the data terms* d_1 =`f[ b ]`, d_2 =`f[ a, b ]` *and* d_3 =`f[ b, b ]`. *The query is translated into the algebra as*

$$
\begin{aligned}
&\sigma(type = \text{“}tree\text{”})\\
&\sigma(label = \text{“}\texttt{f}\text{”})\\
&\sigma(ordered = True)\\
&\sigma(arity \geq 1)\\
&\sigma(arity \leq 2)\\
&Child_{opt,tot}(1, pos = max + 1)[\\
&\qquad \sigma(type = \text{“}tree\text{”})\\
&\qquad \sigma(arity = 0)\\
&\qquad \sigma(label = \text{“}\texttt{a}\text{”})\\
&]\\
&Child(2, pos = max + 1)[\\
&\qquad \sigma(type = \text{“}tree\text{”})\\
&\qquad \sigma(arity = 0)\\
&\qquad \sigma(label = \text{“}\texttt{b}\text{”})\\
&]\\
&Prune
\end{aligned}
$$

The query is successful for the data term d_1. *There is no other match for the term* `b`, *so that the* $Child_{opt,tot}$ *operator returns the input as output, that is the single solution with the true path. After successfully matching the second child operator, the prune operator verifies that the length of the path solution is one, as the arity of the data term is one. The match is successful since this test succeeds.*

The data term d_2 *satisfies the query. Since it is possible to match the first* `a`, *there is only one path:* $\{(1,1),(2,2)\}$. *The second path,* $\{(2,1)\}$ *fails due to label mismatch. The prune operator does not discard this solution, since term arity and path length are the same.*

The data term d_3 does not satisfy the query, because there is no solution with a match for the optional subterm. The path the map operators create is $\{(2,1)\}$, since the label matches the selection. But since the path has length 1 and the data arity is 2, this path is discarded by the prune operator.

Optional Terms and Filter Conditions. The filter conditions that are used to verify path monotonicity are tailored to fit with optional term queries. It would be possible to use filter conditions of the form $pos = k$ for total subterm specifications. When an optional subterm occurs however, the positions of the following data term candidate depend on whether the optional query term was matched. The filter condition $pos = max + 1$ handles this case appropriately: the maximal position increases by one when matching the optional subterm, while it stays the same when omitting it.

In partial and ordered subterms, the filter condition $pos > max$ works with optional subterms, but it was designed in this way because it ensures the monotonicity without imposing surjectivity.

The extension of the algebra for optional terms is simple once negation is defined. The optional child and attribute operators reuse the negated child and attribute operator in partial queries, while the child operators may be bypassed in total queries. The prune operator at the end of the total query then verifies that the number of bypassed optional child or attribute is right for the processed data term.

The Xcerpt query algebra is completed with the definition of the optional query term handling. The algebra can handle attributes and variables, negated and optional query terms. The algebra is still limited to acyclic query terms and data term querying, but all query term constructs are supported by the algebra.

3.4 Rewriting Rules

In this section, equivalence rules for algebraic expressions of the Xcerpt algebra are presented. Such rules are crucial for optimizations, since they allow to transform query expressions into potentially more efficient ones. The equivalence rules that are presented in the following allow to reorder single query steps by selectivity.

Determining the selectivity and performing a sensible reordering of query steps is very challenging, especially in the context of semistructured data. In relational databases, the reordering of joins by their selectivity is already NP-hard, so that the use of heuristics based on histograms, approximation techniques such as simulated annealing, evaluation strategies based on decision points and much more sophisticated search strategies are advised. The following sections do not focus on these topics, but present query rewriting rules and give at most some hints towards what could be a sensible reordering.

The Xcerpt algebra is an algebra which obviously contains scopes. Each map operator (child, attribute, descendant and negated forms thereof) defines an own algebraic subexpression, in which the operators can be reordered. The rules presented in the following only allow such reordering. There are no rewriting rules that move operators from within a map operator out of the operator.

3.4.1 Selection and Bind Reordering

The most straightforward reordering rule is the selection reordering rule. Since selections do not modify the single solutions while filtering and all operators are context preserving, selection can be reordered deliberately. The selection is always performed on the same context node within one algebraic expression. However, it is rarely sensible to place selections behind map operators, since a selection is local to a node and very cheap.

Proposition 1 (Rewriting Rules for Selection Operators) *Let op be an algebra operator. It holds that*

$$\begin{aligned} \sigma(f\ \theta\ v) \circ op &\equiv op \circ \sigma(f\ \theta\ v) \\ \sigma^{pos}(i) \circ op &\equiv op \circ \sigma^{pos}(i) \end{aligned}$$

The same rewriting rules exist for the bind operator.

Proposition 2 (Rewriting Rules for Bind Operators) *Let op be an algebra operator. It holds that*

$$\begin{aligned} Bind(X) \circ op &\equiv op \circ Bind(X) \\ \beta(X = f) \circ op &\equiv op \circ \beta(X = f) \\ \beta^{pos}(X) \circ op &\equiv op \circ \beta^{pos}(X) \end{aligned}$$

The rationale behind these rules is the following. For the local operators such as selection or other bind operators, it is not relevant whether the new binding for X is added before them or behind them, except for the case when the same variable is bound in the operator "*op*" that is swapped with the bind operator. Observe that all bind operators perform the consistency verification. In this case, the bind operator introduced the first binding for X, and the operator "*op*" verified consistency. If the two operators are swapped, the "*op*" operator will introduce the binding, and the $Bind(X)$ operator will verify consistency. Since the consistency verification is commutative, the operators can be swapped. Note that this even holds for the combine operator. The combine operator adds negated constraints to the positive constraint store while verifying consistency. When swapping both operators, the bind operator verifies the consistency with the introduced negative simulation constraints on X. Since the same verify function is used for both operators, the result is again the same. The local operators

$$\sigma, \sigma^{pos}, \beta, \beta^{pos} \text{ and } Bind$$

can be hence reordered deliberately.

Example 20 (Selection and Bind Operator Reordering) *Assume the query*

```
f{{
    position 3 a,
    var X as b{{ /*c/{ } }}
}}
```

The canonical translation (with the translation of the position selection pushed at the end for the sake of the example) of the query above is the left algebraic expression:

$\sigma(type = \text{“}tree\text{”})$
$\sigma(label = \text{“}\texttt{f}\text{”})$
$\sigma(arity \geq 3)$
$Child(1)[$
$\quad \sigma(type = \text{“}tree\text{”})$
$\quad \sigma(label = \text{“}\texttt{a}\text{”})$
$\quad \sigma(arity = 0)$
$\quad \sigma^{pos}(3)$
$]$
$Child(2)[$
$\quad Bind(X)$
$\quad \sigma(type = \text{“}tree\text{”})$
$\quad \sigma(label = \text{“}\texttt{b}\text{”})$
$\quad \sigma(arity \geq 1)$
$\quad Child(1)[$
$\quad\quad \sigma(type = \text{“}tree\text{”})$
$\quad\quad \sigma(label \sim *c)$
$\quad\quad \sigma(arity = 0)$
$\quad]$
$]$

$\sigma(type = \text{“}tree\text{”})$
$\sigma(label = \text{“}\texttt{f}\text{”})$
$\sigma(arity \geq 3)$
$Child(1)[$
$\quad \sigma^{pos}(3)$
$\quad \sigma(type = \text{“}tree\text{”})$
$\quad \sigma(label = \text{“}\texttt{a}\text{”})$
$\quad \sigma(arity = 0)$
$]$
$Child(2)[$
$\quad \sigma(type = \text{“}tree\text{”})$
$\quad \sigma(label = \text{“}\texttt{b}\text{”})$
$\quad \sigma(arity \geq 1)$
$\quad Child(1)[$
$\quad\quad \sigma(type = \text{“}tree\text{”})$
$\quad\quad \sigma(arity = 0)$
$\quad\quad \sigma(label \sim *c)$
$\quad]$
$\quad Bind(X)$
$]$

Since position specifications are very selective, it is sensible to test them first. Furthermore, deferring the binding of the variable X were verified is advised until all selection criteria, even the nested, were verified. Assuming furthermore that the arity selection is more selective and cheaper than the regular expression matching, the two selection operators are swapped. Altogether, the transformed algebraic expression is the left algebraic expression above.

The reordering of local selection gives a lot of freedom to the order of selection and variable binding. The next section investigates reorderings of map operators. These reordering rules are more difficult than the reorderings of local selection operators.

3.4.2 Map Reordering

The reordering of map operators (the child and attributes operators as well as their negated and optional forms) is possible to some extend, but far more complicated. It depends above all on the filter condition imposed. Map operators without any filter conditions on them can be reordered easily. These are all attribute operators, and all operators introduced for unordered term specifications. These rewriting rules are of course not surprising.

In the case of total and ordered subterm specification, the filter condition on the map operator can be changed for reordering: in the child operator for the n-th subterm t_n, the filter condition $(pos = max + 1)$ is replaced by $(pos = n)$. This transformation is possible only if there are no optional subterms occurring before the positive subterm to transform. After an optional subterm, the index to match for a solution depends on the data, and can not be determined statically. The filter conditions on child operators after an optional child operator must hence stay $(pos = max + 1)$. However, it is still possible to reorder all child operators that precede

$$
\begin{array}{lcl}
Child(i_1)[f_1] \circ Child(i_2)[f_2] & \equiv & Child(i_2)[f_2] \circ Child(i_1)[f_1] \\
Child(i_1)[f_1] \circ Child_{neg}(i_2)[f_2] & \equiv & Child_{neg}(i_2)[f_2] \circ Child(i_1)[f_1] \\
Child(i_1)[f_1] \circ Child_{opt,part}(i_2)[f_2] & \equiv & Child_{opt,part}(i_2)[f_2] \circ Child(i_1)[f_1] \\
Child(i_1)[f_1] \circ Child_{opt,tot}(i_2)[f_2] & \equiv & Child_{opt,tot}(i_2)[f_2] \circ Child(i_1)[f_1] \\
Child_{neg}(i_1)[f_1] \circ Child_{neg}(i_2)[f_2] & \equiv & Child_{neg}(i_2)[f_2] \circ Child_{neg}(i_1)[f_1] \\
Child_{neg}(i_1)[f_1] \circ Child_{opt,part}(i_2)[f_2] & \equiv & Child_{opt,part}(i_2)[f_2] \circ Child_{neg}(i_1)[f_1] \\
Child_{neg}(i_1)[f_1] \circ Child_{opt,tot}(i_2)[f_2] & \equiv & Child_{opt,tot}(i_2)[f_2] \circ Child_{neg}(i_1)[f_1] \\
Child_{opt,part}(i_1)[f_1] \circ Child_{opt,part}(i_2)[f_2] & \equiv & Child_{opt,part}(i_2)[f_2] \circ Child_{opt,part}(i_1)[f_1] \\
Child_{opt,part}(i_1)[f_1] \circ Child_{opt,tot}(i_2)[f_2] & \equiv & Child_{opt,tot}(i_2)[f_2] \circ Child_{opt,part}(i_1)[f_1] \\
Child_{opt,tot}(i_1)[f_1] \circ Child_{opt,tot}(i_2)[f_2] & \equiv & Child_{opt,tot}(i_2)[f_2] \circ Child_{opt,tot}(i_1)[f_1] \\
Attr(i_1)[f_1] \circ Attr(i_2)[f_2] & \equiv & Attr(i_2)[f_2] \circ Attr(i_1)[f_1] \\
Attr(i_1)[f_1] \circ Attr_{neg}(i_2)[f_2] & \equiv & Attr_{neg}(i_2)[f_2] \circ Attr(i_1)[f_1] \\
Attr(i_1)[f_1] \circ Attr_{opt,part}(i_2)[f_2] & \equiv & Attr_{opt,part}(i_2)[f_2] \circ Attr(i_1)[f_1] \\
Attr(i_1)[f_1] \circ Attr_{opt,tot}(i_2)[f_2] & \equiv & Attr_{opt,tot}(i_2)[f_2] \circ Attr(i_1)[f_1] \\
Attr_{neg}(i_1)[f_1] \circ Attr_{neg}(i_2)[f_2] & \equiv & Attr_{neg}(i_2)[f_2] \circ Attr_{neg}(i_1)[f_1] \\
Attr_{neg}(i_1)[f_1] \circ Attr_{opt,part}(i_2)[f_2] & \equiv & Attr_{opt,part}(i_2)[f_2] \circ Attr_{neg}(i_1)[f_1] \\
Attr_{neg}(i_1)[f_1] \circ Attr_{opt,tot}(i_2)[f_2] & \equiv & Attr_{opt,tot}(i_2)[f_2] \circ Attr_{neg}(i_1)[f_1] \\
Attr_{opt,part}(i_1)[f_1] \circ Attr_{opt,part}(i_2)[f_2] & \equiv & Attr_{opt,part}(i_2)[f_2] \circ Attr_{opt,part}(i_1)[f_1] \\
Attr_{opt,part}(i_1)[f_1] \circ Attr_{opt,tot}(i_2)[f_2] & \equiv & Attr_{opt,tot}(i_2)[f_2] \circ Attr_{opt,part}(i_1)[f_1] \\
Attr_{opt,tot}(i_1)[f_1] \circ Attr_{opt,tot}(i_2)[f_2] & \equiv & Attr_{opt,tot}(i_2)[f_2] \circ Attr_{opt,tot}(i_1)[f_1]
\end{array}
$$

Table 6: Simple Rewriting Rules for Child and Attribute Operators

any optional child operator in the query term list.

$$Child(i_1, pos = p_1)[f_1] \circ Child(i_2, pos = p_2)[f_2] \quad \equiv \quad Child(i_2, pos = p_2)[f_2] \circ Child(i_1, pos = p_1)[f_1]$$

The case of reordering gets more complicated for partial and ordered queries. For the sake of brevity, reordering rules are not provided for query term lists with non-positive query terms. In the following, it is hence assumed that all terms of the list are positive. Furthermore, the following discussion is restricted to compute the proper filter conditions to add to each child operator instead of specifying reordering rules.

the following filter conditions are necessary to support the reordering of subterms:

- Allow only positions greater than or equal to $k : Integer$.

$$(pos \geq k) = \{(path, (i, j), size) \mid j \geq k\}$$

- Allow only positions greater than a certain path position plus an offset $k : Integer$.

$$(pos > path(p) + k) = \{(path, (i, j), size) \mid j > path(p) + k\}$$

- Allow only positions less than a certain path position plus an offset $k : Integer$.

$$(pos < path(p) + k) = \{(path, (i, j), size) \mid j < path(p) + k\}$$

Definition 40 (Term Position Sets) *Let $\langle t_1, \ldots, t_n \rangle$ be a query term sequence with $n \geq 1$. Let $1 \leq p \leq n$. Let π be a permutation on $\{1, \ldots, n\}$.*

- $T_{<p} = \{1, \ldots, p-1\}$ *is the set of all query term positions less than p*
- $T_{>p} = \{p+1, \ldots, n\}$ *is the set of all query term positions greater than p*
- $T^{\pi}_{<p} = \{i \mid \pi(i) < \pi(p), 1 \leq i \leq n\}$ *is the set of all query term positions less than p in the permuted order.*
- $T^{\pi}_{>p} = \{i \mid \pi(i) > \pi(p), 1 \leq i \leq n\}$ *is the set of all query term positions greater than p in the permuted order.*

Considering the filter conditions to add, there are four cases to handle separately. The first case applies if there are query terms occurring before t_p in the initial order that are occurring after t_p in the match order, and none of them were executed before t_p (i.e. $T_{<p} \cap T^{\pi}_{>p} \neq \emptyset$ and $T_{<p} \cap T^{\pi}_{<p} = \emptyset$). In such a case, the child operator must reserve enough data terms to match all query terms before t_p. Hence, the filter condition to add is

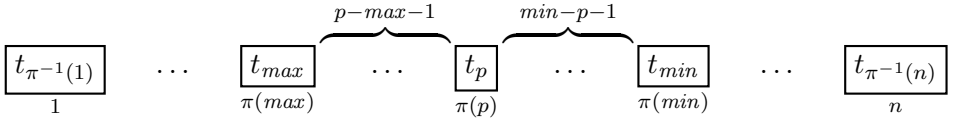
$$pos \geq p$$

The second case applies if there are query terms occurring after t_p in the initial order that are occurring after t_p in the match order and again none of them were executed before (i.e. $T_{>p} \cap T^{\pi}_{>p} \neq \emptyset$ and $T_{>p} \cap T^{\pi}_{<p} = \emptyset$). Similarly to the first case, the child operator must reserve enough data terms to match all query terms after t_p. The filter condition hence becomes

$$pos \leq -(n - p)$$

The third and fourth cases restrict the position of children dynamically with the position of data terms already matched by query terms. It must be verified that the monotonicity constraint is preserved when extending it with the child position. The filter conditions must compare the position of the children with the positions of already matched terms.

The third case, where $T_{<p} \cap T^{\pi}_{<p} \neq \emptyset$, is the case where query terms occurring before t_p in the initial order will be matched before t_p. This case occurs in the normal query term translation without reordering. Here, the child operator must ensure that the matched data term occurs after all already matched data terms with positions in $T_{<p} \cap T^{\pi}_{<p}$. So, the filter condition is like $(pos > path(p-1))$. However, the query term $p-1$ may not have been matched before t_p, so that the path is not populated for $p-1$. Hence, the position max of the term with the largest position within the set $T^{\pi}_{<p} \cap T_{<p}$ must be used. The filter condition must furthermore ensure that enough additional data terms to match the query terms between t_{max} and t_p are present.



The values max and min are $max = max(T_{<p} \cap T^{\pi}_{<p})$ and $min = min(T_{>p} \cap T^{\pi}_{<p})$ (max is the largest index in the set of all term indices of terms occurring before t_p and matched before

t_p, *min* is the smallest index in the set of all term indices of terms occurring after t_p and matched before t_p). In the case $T_{<p} \cap T^{\pi}_{<p} \neq \emptyset$, the following filter condition must be added to the child operator:

$$pos \geq path(max) + p - max$$

Similarly, in the case that $T_{>p} \cap T^{\pi}_{<p} \neq \emptyset$, the filter condition to add is

$$pos \leq path(min) - (min - p)$$

To summarize, the following filter conditions are added to child operators.

- if $T_{<p} \cap T^{\pi}_{>p} \neq \emptyset$ and $T_{<p} \cap T^{\pi}_{<p} = \emptyset$ then $(pos \geq p)$
- if $T_{>p} \cap T^{\pi}_{>p} \neq \emptyset$ and $T_{>p} \cap T^{\pi}_{<p} = \emptyset$ then $(pos \leq -(n - p))$
- if $T_{<p} \cap T^{\pi}_{<p} \neq \emptyset$ then $(pos > path(max) + p - max - 1)$ where $max = max(T_{<p} \cap T^{\pi}_{<p})$
- if $T_{>p} \cap T^{\pi}_{<p} \neq \emptyset$ then $(pos < path(min) - (min - p - 1))$ where $min = min(T_{>p} \cap T^{\pi}_{<p})$

Example 21 *Consider the query term* `f[[ a, b, c, d ]]` *Assuming that the optimized query order is* `b, d, c, a`*, so the permutation on the children position is*

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$$

The filter condition for the child operators are computed as follows.

- *for the term* `b` *the term sets are*
  - $T_{<2} = \{1\}$
  - $T_{>2} = \{3, 4\}$
  - $T^{\pi}_{<2} = \emptyset$
  - $T^{\pi}_{>2} = \{1, 3, 4\}$

  The filter conditions are hence:
  - $(pos \geq 2)$*, since* $T_{<2} \cap T^{\pi}_{>2} = \{1\}$*,* $T_{<2} \cap T^{\pi}_{<2} = \emptyset$*.*
  - $(pos \leq -2)$*, since* $T_{>2} \cap T^{\pi}_{>2} = \{3, 4\}$*,* $T_{>2} \cap T^{\pi}_{<2} = \emptyset$ *and* $n - p = 2$*.*
- *for the term* `d` *the term sets are*
  - $T_{<4} = \{1, 2, 3\}$
  - $T_{>4} = \emptyset$
  - $T^{\pi}_{<4} = \{2\}$
  - $T^{\pi}_{>4} = \{1, 3\}$*.*

  The filter conditions are hence:
  - $(pos > path(2) + 1)$*, since* $T_{<4} \cap T^{\pi}_{<4} = \{2\}$ *and the maximum is* 2*, so that the offset becomes* $4 - 2 - 1$*, that is,* 1*.*

- *for the term* `c` *the term sets are*
  - $T_{<3} = \{1, 2\}$
  - $T_{>3} = \{4\}$
  - $T^{\pi}_{<3} = \{2, 4\}$
  - $T^{\pi}_{>3} = \{1\}$

  The filter conditions are hence:
  - $(pos > path(2))$, *since* $T_{<3} \cap T^{\pi}_{<3} = \{2\}$ *and the maximum is* 2, *so that the offset becomes* $3 - 2 - 1$, *that is,* 0.
  - $(pos < path(3))$, *since* $T_{>3} \cap T^{\pi}_{<3} = \{4\}$ *and the minimum is* 4, *so that the offset becomes* $4 - 3 - 1$, *that is,* 0.
- *for the term* `a` *the term sets are*
  - $T_{<1} = \emptyset$
  - $T_{>1} = \{2, 3, 4\}$
  - $T^{\pi}_{<1} = \{2, 3, 4\}$
  - $T^{\pi}_{>1} = \emptyset$

  The filter conditions are hence:
  - $(pos < path(2))$, *since* $T_{>1} \cap T^{\pi}_{<1} = \{2, 3, 4\}$ *and the minimum is* 2, *so that the offset becomes* $2 - 1 - 1$, *that is,* 0.

To summarize, the initial order was `a, b, c, d`, *the optimized order is* `b, d, c, a`, *and the map operators are*

$$Child(2, pos \geq 2, pos \leq -2)[\ldots]$$
$$Child(4, pos > path(2) + 1)[\ldots]$$
$$Child(3, pos > path(2), pos < path(4))[\ldots]$$
$$Child(1, pos < path(2))[\ldots]$$

The algebraic expression is evaluated with the data term `f[a, b, d, a, c, d, a, b, c, c]`. *The computation is represented with the help of a matching matrix. The symbols* $\surd$ *represents a successful match,* $-$ *a failed match, and* $\times$ *represent cells that are not checked for any solution due to the filter conditions. The matrix represents several solutions at once, and every path will be marked with arrows. The initial matrix is*

	a	*b*	*d*	*a*	*c*	*d*	*a*	*b*	*c*	*c*
a										
b										
c										
d										

Every row represents a query term, every column represents a data term. A cell represents the result of the matching of both.

First, the second query term `b` *is executed, remember the filter conditions were* $(pos \geq 2, pos \leq -2]$.

	a	b	d	a	c	d	a	b	c	c
a										
b	×	√	-	-	-	-	-	√	×	×
c										
d										

There are two matches for `b`*, so the query evaluation contains two solution candidates.*

Second, the fourth query tern `d` *is executed. The filter condition was* $(pos > path(2)+1)$*. The resulting matrix gets*

	a	b	d	a	c	d	a	b	c	c
a										
b	×	√	-	-	-	-	-	(√)	×	×
c										
d	×	×	×	-	-	√	-	-	-	-

The second match for `b` *must be discarded, because no* `d` *exists after position 8.*

Third, the third query term `c` *is executed. Bare in mind that the filter conditions were* $(pos > path(2), pos < path(4))$*, so that previous matchings constraint both the upper and the lower bound are dynamically. The position of data terms must be between* 3 *and* 5 *in this example.*

	a	b	d	a	c	d	a	b	c	c
a										
b	×	√	-	-	-	-	-	(√)	×	×
c	×	×	-	-	√	×	×	×	×	×
d	×	×	×	-	-	√	-	-	-	-

Fourth, the first query term `a` *is executed. Here, the filter condition is* $(pos < path(2))$*, that is, the data term must have a position less than the position of the data term matched by the query term* `b`*.*

	a	b	d	a	c	d	a	b	c	c
a	√	×	×	×	×	×	×	×	×	×
b	×	√	-	-	-	-	-	(√)	×	×
c	×	×	-	-	√	×	×	×	×	×
d	×	×	×	-	-	√	-	-	-	-

Finally, the single existing solution is the path $\{(1,1),(2,2),(3,5),(4,6)\}$*. Note that through the filter conditions and optimization of the query order, the number of tested matrix cells was reduced from* 40 *to* 18*.*

The representation of query evaluation chosen here reflects the algebraic operators only in case of limited example complexity such as the above. If the evaluation of a subquery for example lead to more than one constraint store, there would be several paths with the same cell traversal, but with different constraint stores. The representation of these paths would be difficult in the graphical representation above.

In fact, the matrix representation above reflects the matrix method that is discussed in section 4.1.

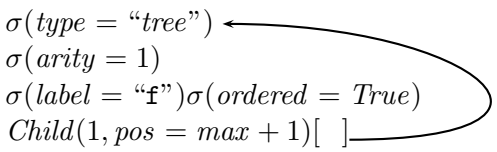
The reordering rules introduced in this section allow to reorder local selections deliberately. The map operators on the other hand can be reordered only under certain conditions. Obviously, map operators that stem from unordered queries can be reordered deliberately, while map operators from ordered queries are more difficult to reorder. However, it is possible to reorder these map operators in the absence of optional and negated query constructs. Further investigations may be made to determine reordering rules in the presence of these special query constructs.

3.5 Extending the Xcerpt Algebra for Construct Terms and Cyclic Query Terms

As already stated in section 3.2, the Xcerpt query algebra does not support the simulation unification with construct terms and does not translate cyclic query terms. The possibilities to lift these limitations and the reasons for these limitations are discussed in the following.

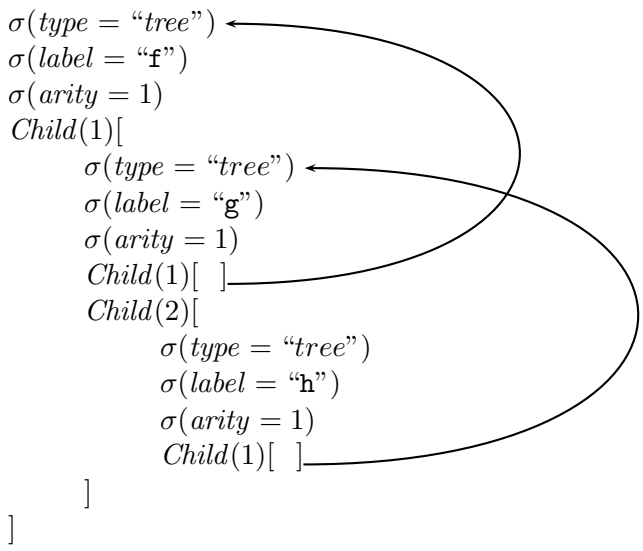
First, consider the restriction on acyclic query terms. The solution to handle cyclic query terms appropriately in the constraint calculus above is to introduce a memoization table. Whenever a simulation constraint with an identifier on the query term is encountered, a table lookup with both the query and construct terms as table key is performed. If an entry for the pair exists, the constraint is reduced to *False* if the query constraint was a descendant construct, and to *True* if it was not. If no entry exists in the table, the entry is inserted into the table. The rationale behind this trick is that all reachable nodes from the actual construct term node were already considered if a pair of query and data term is encountered a second time with the query term being a descendant. Hence, the decomposition of the descendant term will not lead to new solutions. In the case of a non-descendant query term, all constraints stemming from the decomposition of the query term were already added, so that the decomposition of the non-atomic simulation constraint will not add new constraints to the store. It is possible to introduce another set into the algebra domain, to which every child or descendant operator adds a pair consisting of the identifier and the node to realize this memoization. This would however lead to cyclic algebra expressions with sophisticated semantics. Furthermore, it was not investigated how options and negations work within cycles.

Example 22 (Cyclic Query Term and Algebra Expression) *Consider the query term* `q1@f[ ^q1 ]`. *The translation of it would be*



A Query term with two intertwined cycles would be

```
q1@f{ q2@ g{ ^q1, h{ ^q2 } } }
```



As this example shows, there are cases of cyclic query terms where a handling with loop operators in impossible, since the cycles may be intertwined.

Second, the unification with construct terms is considered. Construct terms may contain variables, furthermore grouping and optional constructs. It is of course possible to extend the data model to support these. The map operator then may have to apply a function to a variable or to a construct that cannot be decomposed completely (such as aggregation and grouping constructs). In such a case, another constraint of the form $f \preceq X$ could be introduced with X a variable and f an algebraic expression, stating that f must hold for the bindings of the variable X. The binding operators would have to check whether there are such constraints in the store when adding bindings, and apply the function f to the node they would bind to X. This is roughly the scheme that an extension would have to follow when introducing construct terms, but they are left aside since algebraic expressions would be passed through the constraint store from operator to operator. Furthermore, the injectivity restriction on the mappings must be lifted when encountering grouping construct terms. It is not yet investigated how well this works with the query algebra.

In this section, it is demonstrated how the Xcerpt query algebra decomposes Xcerpt queries into smaller units of execution. There are still functional dependencies between the query operators (e.g. the negative child operator depends on the functionality of the positive child operator), so that changing the semantics of one operator affects the other operators.

The algebra was first defined as a basic version and extended step by step until it covered all query term constructs. However, the Xcerpt query algebra is limited to acyclic query terms and to queries data terms instead of construct terms. Nevertheless, the possibilities to lift these restrictions are discussed and sketched.

The algebra furthermore demonstrated various possible optimizations for the execution of subquery lists. The query optimizations were different for each kind of child query, that is, the general query evaluation was specialized for each of the four query list types.

Furthermore, rewriting rules give complete freedom of reordering selections and variable bind-

ings as appropriate. For example, if an as-pattern query does not contain the as-variable, the binding of the as-pattern variable can be deferred until all child operators were evaluated and thus the constraint store size is reduced. Local selections can be reordered by selectivity, and map operators can be even reordered to some extend; attribute operators and unordered child operators may be reordered arbitrarily, while ordered child operators can be reordered if the query term list does not contain optional or negated subterms.

To summarize, the algebra showed how it is possible to

- reduce the number of candidate nodes for attribute and child operator subqueries based on injectivity and order restrictions,
- reorder selection operators by selectivity, and
- defer variable bindings if it is wise to do so.

Most of the above optimizations are not tractable by complexity analysis, especially the gain through query rewriting is dependant on the meta information analysis and rewrite heuristics that can be used. The only optimization in the Xcerpt query algebra that yields a tractable complexity reduction are the context size reductions through filter conditions. The gain from this optimization is investigated in the following section 4.

Optimizing the query evaluation with an algebra allows static program analysis and static optimizations. Filter conditions based on the kind of subterm are introduced and reorderings of the various operators may be performed based on static program analysis. The algebra operators themselves however perform dynamic optimizations based on the context information of the query evaluation. The map operators for example reduce the number of candidates for the subexpressions by analysis of the path information. The dynamic analysis methods can be pushed much farther than it is yet done in the query algebra. As the next section reveals, there are possibilities of dynamic analysis and optimization through solution pruning not yet incorporated into the algebra that lead to impressive runtime improvements.

4 Algorithmic Optimizations with the Matrix Method

This section demonstrates and analyzes the complexity of algorithmic optimizations of unification. Some of the presented optimizations are incorporated into the Xcerpt query algebra introduced above. The worst case and best case runtime behavior of these optimizations are investigated in this section in order to evaluate the impact of each optimization. The average case can not be evaluated in this work, since reasonable average case assumptions for Xcerpt programs were not investigated yet.

The key optimization concepts are presented in the following without focusing on their incorporation into the algebra. Other optimizations, while not yet incorporated into the query algebra, provide valuable algorithmic complexity reductions for common cases of unification.

The optimizations presented here are based on the matrix method technique for simulation unification of term lists developed by Schaffert [Sch04]. This technique is referenced to as "matrix method" in the following. Section 4.1 introduces the matrix method data structure and algorithms, and analyze the complexity of two algorithm parts that are optimized in the

following sections 4.2 and 4.3. The analysis of the basic algorithm is used to compare the optimizations with each other and to estimate the benefits for simulation unification. Finally, Section 4.4 presents as conclusion some preliminary performance tests of the optimizations with the prototypical Xcerpt interpreter written in Java.

4.1 Query Evaluation with the Matrix Method

The matrix method is a key technique in the existing prototypical implementations. The basic concept of the matrix method is to use a matrix to memoize unification results. The matrix has the following shape: given a query term list $\langle q_1, \ldots, q_n \rangle$ and a construct term list $\langle c_1, \ldots, c_m \rangle$, the matrix cell (i, j) contains the unification result of the query term q_i with the construct term c_j. A matrix row i consists of all unification results of the query term q_i with all construct terms $(c_j)_j$. A matrix column j is composed of all unification results of all query terms $(q_i)_i$ with the construct term c_j.

This restricted form of memoizing leads to tremendous performance improvements, since one simulation unification result is used several times while computing the valid mappings between query and construct term lists.

Using a matrix to represent formulas is not a novel approach. There are several approaches in logical deduction theory that use a matrix to represent sets of facts or formula in CNF (see e.g. [Bib92] for an overview of deduction theories using matrices). The "canonical" semantics of a matrix $M = (a_{ij})_{i,j}$ with $1 \leq i \leq n$ and $1 \leq j \leq m$ is the formula

$$F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} a_{ij}$$

The disjunctive normal form of F can be computed by expanding all combinations of taking a cell from each row in a disjunction.

Example 23 (Disjunctive Normal Form) *Assume the matrix*

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

The disjunctive normal form of the matrix formula

$$\begin{array}{rcl} F & = & (a_{11} \vee a_{12} \vee a_{13}) \\ & \wedge & (a_{21} \vee a_{22} \vee a_{23}) \end{array}$$

is computed by expanding all combinations of cells between rows.

$$\begin{array}{rcccccc} F_{DNF} & = & a_{11} \wedge a_{21} & \vee & a_{11} \wedge a_{22} & \vee & a_{11} \wedge a_{23} \\ & \vee & a_{12} \wedge a_{21} & \vee & a_{12} \wedge a_{22} & \vee & a_{12} \wedge a_{23} \\ & \vee & a_{13} \wedge a_{21} & \vee & a_{13} \wedge a_{22} & \vee & a_{13} \wedge a_{23} \end{array}$$

To represent a single conjunction of the matrix DNF, it is sufficient to list the cell positions instead of the matrix cell contents. Together with the matrix, the set of cell positions can be translated into the formula it represents.

Definition 41 (Matrix Path) *Let $M = (a_{ij})_{i,j}$ be a $n \times m$ matrix. A matrix path of M is a set $p \subseteq \{1, \ldots, n\} \times \{1, \ldots, m\}$*

A matrix path in the unification matrix for the query term list $\langle q_1, \ldots, q_n \rangle$ to the construct term list $\langle c_1, \ldots, c_m \rangle$ represents a mapping of the query list to the construct list if it is total on the positive query terms.

Example 24 (Matrix Path) *The following matrix shows a total, injective and surjective, but non-monotonic path:*

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

This path represents the formula

$$a_{11} \wedge a_{23} \wedge a_{32}$$

The novel idea in the matrix method is to impose mapping constraints (i.e., monotonicity, injectivity, and surjectivity) on the matrix paths. Note that a matrix with constraints on its paths does not represent the formula F anymore. The matrix represents a formula that is a subformula of the DNF of F, since the paths that do not satisfy the path constraints are missing. The formula F represents all possible total paths without any monotonicity, injectivity or surjectivity restrictions.

While collecting the cell contents of a given path to compute the associated constraint set, it is suitable to apply the consistency verification rules incrementally. This allows to drop a path as soon as it becomes inconsistent and avoids redundant consistency verification, which can be highly time consuming.

Computing all results of the unification of a given query term list with a given construct term list requires generating all possible matrix paths of the unification matrix. While generating the paths, it is also suitable to verify the path constraints incrementally to reduce the number of generated paths. Not all path constraints can be easily verified incrementally, however section 4.2 shows how all mapping constraints can be used to reduce the number of generated paths.

Each mapping property can be verified on a path p in the following way.

- injectivity: the path p may touch each column only once.
- monotonicity: for all positions $(i_1, j_1), (i_2, j_2) \in p$ it holds that $i_1 < i_2$ implies $j_1 < j_2$.
- surjectivity: the path p must touch each column.

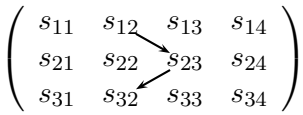
Combining the surjectivity and monotonicity property (in the absence of optional terms) leads to the following constraint:

- monotonicity and injectivity: for all $(i, j) \in p$ it holds that $i = j$.

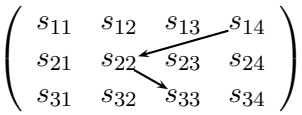
The unification matrix is hence a suitable data structure that memoizes unification results for a fixed time frame; the matrix memoizes all unification results as long as the unification of a given query term list with a given construct term list is processed. After finding the paths and collecting all constraints in the paths, the matrix is discarded with the memoized results.

Example 25 (Constraints on Matrix Paths)

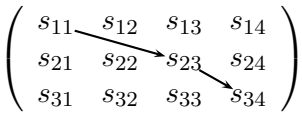
Non-injective path



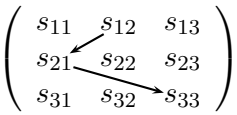
Injective but non-monotonic path



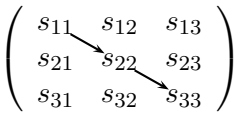
Monotonic path



Bijective and non-monotonic path



Bijective and monotonic path



To generate matrix paths, a matrix is processed row by row. When negated query terms (or optional query terms in partial queries) occur, a matrix path must consist of a positive and a negative part. The positive part consists of one single path, while the negative consists of a set of possible path extensions. These extensions are handled as in the Xcerpt query algebra: If the currently processed row is positive (i.e. stemming from a positive query term), the path constraints are verified on the positive path only, and every extensions that is inconsistent with the extended positive path is dropped. If it is negative, each path extension prefix is extended with each cell that is compatible with both the path extension and the positive path. If the row is optional and the query is partial, it is treated as positive and as negative row. If the query is total, the row is treated as positive, and the unextended paths are added to the extended ones so that the row may be skipped.

Example 26 (Path Extensions) *Assume the following unification problem: a partial and unordered query with three query subterms q_1, q_2, q_3 of which q_3 is negative must be unified with six data terms $d_1, \ldots, d_6$. Given the positive path*

$$\{(1,5),(2,3)\},$$

the compatible extensions are computed. These extensions are

$$\{\emptyset, \{(3,1)\}, \{(3,2)\}, \{(3,4)\}, \{(3,6)\}\}.$$

The unification matrix shows that these are (besides the path extension $\emptyset$) all combinations of the query term q_3 and previously unmatched data terms.

$$\begin{matrix} + \\ + \\ - \end{matrix} \begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} & s_{15} & s_{16} \\ s_{21} & s_{22} & s_{23} & s_{24} & s_{25} & s_{26} \\ s_{31} & s_{32} & s_{33} & s_{34} & s_{35} & s_{36} \end{pmatrix}$$

Assume a second unification problem with again three query terms q_1, q_2, q_3 of which q_2 is negative and six data terms $d_1, \ldots, d_6$. Let the query list be ordered and partial. Given the positive path

$$\{(1,2),(3,6)\},$$

the valid path extensions (that is, the paths satisfying the monotonicity condition) are

$$\{\emptyset, \{(2,3)\}, \{(2,4)\}, \{(2,5)\}\}.$$

These are all cells with column positions between 3 *and* 5*. The path extensions are restricted by the positive terms.*

$$\begin{array}{c} + \\ - \\ + \end{array} \begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} & s_{15} & s_{16} \\ s_{21} & s_{22} & s_{23} & s_{24} & s_{25} & s_{26} \\ s_{31} & s_{32} & s_{33} & s_{34} & s_{35} & s_{36} \end{pmatrix}$$

The algorithm for the unification of a query term list *query* and a construct term list *construct* reads as follows:

```
UNIFY-SUBTERMS(query, construct)
1  matrix ← FILL-MATRIX(query, construct)
2  paths ← MATRIX-PATHS(matrix)
3  solutions ← EXECUTE-PATHS(matrix, paths)
4  return solutions
```

The construction of paths is detached from the constraint combination. In the presence of optional and negated terms, a large number of matrix paths get computed and discarded at a later point of evaluation due to injectivity, surjectivity or monotonicity constraints. The verification of the path constraint consistency at the moment of path construction would lead to unnecessary consistency verification of variable bindings. The detaching of "path generation" (that is, determining the valid paths through a matrix respecting injectivity, surjectivity and monotonicity constraints) and "path execution" (that is, collecting, DNF-expanding and verifying consistency of variable bindings on a given path) hence prevents this consistency verification overhead.

```
FILL-MATRIX(query, construct)
 1  matrix ← nil
 2  row_pos ← 1
 3  for q in query
 4  do row ← nil
 5     col_pos ← 1
 6     for c in construct
 7     do solutions ← UNIFY(q, c)
 8        position ← (row_pos, col_pos)
 9        row ← row + [(position, solutions)]
10        col_pos ← col_pos + 1
11     matrix ← matrix + [row]
12     row_pos ← col_pos + 1
13  return matrix
```

The following sections show optimizations of the first two algorithms, FILL-MATRIX and MATRIX-PATHS. Optimizations of the third algorithm, EXECUTE-PATHS, are not considered in the following, since these optimizations concentrate on the consistency verification rules on which this work did not focus.

The algorithm FILL-MATRIX performs two nested loops. The outer loop iterates over the query list, and the inner loop iterates over the construct term list. The number of recursive

calls to the algorithm UNIFY is $q \cdot c$, where q is the query list size (i.e., the number of query terms in a query term list) and c is the construct list size (i.e., the number of construct terms in a construct term list). The calls to UNIFY may be in turn very expensive. If the query subterm is composed and consists of m terms in total, a recursive unification call can involve m applications of the matrix method, each involving path consistency and variable consistency verification. It is hence sensible to reduce this number of recursive calls. Section 4.2 focuses on this aspect of optimization.

```
MATRIX-PATHS(matrix)
 1  paths ← {(∅, ∅)}
 2  for row in matrix
 3  do switch
 4        case positive(row) :
 5              paths ← EXTEND-POSITIVE(row, paths)
 6        case negative(row) :
 7              paths ← EXTEND-NEGATIVE(row, paths)
 8        case optional(row) and partial(matrix) :
 9              paths ← EXTEND-POSITIVE(row, paths) ∪ EXTEND-NEGATIVE(row, paths)
10        case optional(row) and total(matrix) :
11              paths ← EXTEND-POSITIVE(row, paths) ∪ paths
12  if total(matrix)
13    then PRUNE-NONSURJECTIVE(paths)
14  return paths
```

The algorithm MATRIX-PATHS iterates over the matrix rows applying the appropriate extension algorithm for each row.

The runtime of the MATRIX-PATHS algorithm is determined by the number of existing paths. Assuming that there are no optional and negated queries in the matrix, the number of matrix paths in a $q \times c$ matrix can be calculated as follows.

Query Type	Path Type	Number of Paths
Partial and unordered	Injective	$q! \cdot \binom{c}{q}$
Partial and ordered	Monotonic	$\binom{c}{q}$
Total and unordered	Bijective	$q!$
Total and ordered	Bijective and monotonic	1

Table 7: Number of Paths in a $q \times c$ Matrix with no Special Constructs

Computing an injective path can be represented as choosing q items out of c without repetitions and with the consideration of choice order. Computing a monotonic path is equivalent to choosing q items out of c without considering choice order, since the distribution order of the construct terms is already determined by the order of the query terms. Computing the bijective paths and the bijective monotonic paths is a specialization of both combinatoric computations mentioned above for the case $c = q$.

```
EXTEND-POSITIVE(row, paths)
 1  new_paths ← ∅
 2  for cell in row
 3  do if not falsified(cell)
 4        then pos ← position(cell)
 5             for (path, extensions) in paths
 6             do if compatible(path, pos)
 7                   then path ← path ∪ {pos}
 8                        new_extensions ← ∅
 9                        for extension in extensions
10                        do if compatible(extension, pos)
11                              then new_extensions ← new_extensions ∪ {extension}
12                        new_paths ← new_paths ∪ {(path ∪ pos, new_extensions)}
13  return new_paths
```

```
EXTEND-NEGATIVE(row, paths)
 1  new_paths ← copy(paths)
 2  for cell in row
 3  do if not falsified(cell)
 4        then pos ← position(cell)
 5             for (path, extensions) in paths
 6             do if compatible(path, pos)
 7                   then new_extensions ← ∅
 8                        for extension in extensions
 9                        do if compatible(extension, pos)
10                              then new_extension ← extension ∪ {pos}
11                                   new_extensions ← new_extensions ∪ {new_extension}
12                        extensions ← extensions ∪ new_extensions
13                        new_paths ← new_paths ∪ {(path, extensions)}
14  return new_paths
```

Note that these are the path numbers without optional and negated terms. First consider the number of surjective paths through a $q \times c$ matrix with o optional query subterms (Remember, negated subterms are not allowed in a total query). It must hold for this matrix that $q - o \leq c \leq q$, since there may be at most c construct terms but not less than the number of positive query terms. The number of positive query terms is $p = q - o$. There are 2^o possibilities to choose an arbitrary number i of optional terms with $0 \leq i \leq o$, and there are $\left(\begin{array}{c} o \\ i \end{array} \right)$ possibilities to choose i out of o optional terms. Since the query is total, the paths must be surjective and the number of enabled optional terms must be $i = d - p$. The number of surjective and monotonic paths through a $d \times q$ matrix with o optional terms is therefore $\left(\begin{array}{c} o \\ d-p \end{array} \right)$. The number of surjective paths through the same matrix requires to consider the choice order, so that there are $p! \cdot \left(\begin{array}{c} c \\ p \end{array} \right) \cdot (d-p)! \cdot \left(\begin{array}{c} o \\ d-p \end{array} \right)$ of them.

Unfortunately, the MATRIX-PATHS algorithm computes far more paths and prunes the non-surjective paths at the end. The algorithm computes all combinations for choosing $0 \leq i \leq d - p$ enabled optional terms. Hence, the algorithm computes

$$p! \cdot \binom{d}{p} \cdot \sum_{i=0}^{d-p} \left[i! \cdot \binom{o}{i} \cdot \binom{d-p}{i} \right]$$

bijective paths and

$$\sum_{i=0}^{d-p} \binom{o}{i}$$

bijective monotonic paths in search for the surjective paths.[17]

Query Type	Total and unordered	Total and ordered
Number of Paths	$p! \cdot \binom{d}{p} \cdot (d-p)! \cdot \binom{o}{d-p}$	$\binom{o}{d-p}$
Generated Paths	$p! \cdot \binom{d}{p} \cdot \sum_{i=0}^{d-p} \left[i! \cdot \binom{o}{i} \cdot \binom{d-p}{i} \right]$	$\sum_{i=0}^{d-p} \binom{o}{i}$

Table 8: Number of Paths in a $q \times c$ Matrix with o Optional Query Terms and p Positive Query Terms

The number of generated paths diminishes drastically if paths through false cells are dropped immediately. Note that it is also allowed to drop the paths through false cells of negative rows. A path through a falsified cell of a negative row does not add any constraints to the path, since the negation of *False* is *True*.

Example 27 (Number of Paths) *The explosion of the number of surjective paths in a matrix with optional terms is shown in the following table.*

Matrix Size	4×4	8×8	16×8	16×8	16×8
Optional Terms	0	0	8	12	16
Surjective, Monotonic Paths	1	1	1	495	12,870
Surjective Paths	24	40,320	40,320	19,958,400	518,918,400

To summarize, the algorithms for matrix filling and path generation involve quite an overhead; the matrix filling computes the unification of query and construct term pairs that will never be part of a valid path due to injectivity and monotonicity constraints. The path generation on the other hand creates a tremendous amount of paths and negative path extensions that will be either pruned or subsumed by other paths. The next sections demonstrate optimizations that reduce this overhead considerably.

[17] Computing the number of paths for partial query terms was not possible in the given time frame of this work.

4.2 Optimizing Matrix Filling

The algebra with filter conditions already demonstrated how unification of certain query-data pairs can be avoided. This section investigates the gain through these filter conditions as well as further optimizations.

The FILL-MATRIX algorithm must be specialized for unordered, partial-ordered and total-ordered queries, since the path constraints and hence the possible refinements of the filling scheme differ.

```
FILL-MATRIX(query, construct)
1  switch
2    case ordered(query) and total(query) :
3         return FILL-MATRIX-TOTAL-ORDERED(query, construct)
4    case ordered(query) and not total(query) :
5         return FILL-MATRIX-PARTIAL-ORDERED(query, construct)
6    case not ordered(query) :
7         return FILL-MATRIX-UNORDERED(query, construct)
```

The techniques of each matrix filling algorithm are described in the following three sections. There are possibilities to reduce the number of recursive unification calls especially for total and ordered query term lists (section 4.2.2) as well as for partial and ordered query term lists (section 4.2.1). Especially the first reduction of recursive calls proves valuable. The third reduction of matrix filling for unordered queries (section 4.2.3) is rather marginal but at the same time very cheap. Under certain circumstances, this reduction of recursive calls can become noticeable. Furthermore, general techniques can be identified to abort matrix filling and list unification as a whole. These techniques are discussed in section 4.2.4.

4.2.1 Ordered Queries: Preventing Recursive Calls by Monotonicity

Considering the monotonic paths through a given unification matrix, there are matrix cells that are never part of any monotonic path. Roughly speaking, the matrix cells at the topmost right and the lowermost left corners of the matrix cannot be part of a monotonic path: the monotonic matrix paths progress from the topmost left corner to the lowermost left corner. It is hence possible to omit cell computation in the topmost right and lowermost left corner completely.

Take a partial and ordered query leading to a $q \times c$ unification matrix. For this matrix, it holds that for a given row, the n first cells can be left unfilled. n is the minimal column position of a non-falsified cell in the last preceding positive row. If there is no preceding positive row, then no cell can be dropped. The reason for this is that every path may have chosen the position (i', n) from the last preceding positive row. All paths that choose a cell with a position less than $n + 1$ in the actual row are unordered paths. There are hence no monotonic paths that can choose a cell with a position less than $n + 1$.

It is furthermore possible to leave the m last cells of a row unfilled, if m is the maximal column position of a non-falsified cell in the first following positive row. If there is no following row, then no cell can be dropped. The reason for this is similar to the above. A monotonic path

taking a position of the actual row that is greater than or equal to m will find no non-falsified cell with a column position greater than m.

The cost to realize this optimization is in $O(q \cdot c)$, since removing the cells at the row end requires a second reverse pass over the matrix, and at most, c elements must be removed from q rows. It is however possible to determine in the first pass whether the second reverse pass over the unification matrix is necessary.

```
FILL-MATRIX-PARTIAL-ORDERED(query, construct)
 1  matrix ← nil
 2  first ← 0
 3  last ← 0
 4  following ← positiveTerms(query)
 5  secondpass ← False
 6  for r ← 1 to size(query)
 7  do row ← nil
 8     for c ← first + 1 to size(construct) − following
 9     do position ← (r, c)
10        solutions ← UNIFY(query[r], construct[c])
11        row ← row + [(position, solutions)
12     matrix ← matrix + [row]
13     if positive(query[r])
14       then following ← following − 1
15            first ← min(columnPositions(row))
16     secondpass ← secondpass or (last > max(columnPositions(row)))
17     last ← max(columnPositions(row))
18     matrix ← matrix + [row]
19  if secondpass
20    then last ← size(construct)
21         for r ← size(query) downto 1
22         do row ← matrix[r]
23            DROP-ALL-AFTER(last, row)
24            last ← max(columnPositions(row))
25            return matrix
```

The first pass uses three counters *first*, *last* and *following*. The counter *first* contains the column position of the first non-falsified cell in the last preceding positive row, *last* contains the column position of the last non-falsified cell in the last preceding positive row. The counter *following* contains the number of following positive rows. The counters are initialized as follows: $first = 0, last = c$ and $following = p$, where c is the number of construct terms and p is the number of positive query terms. Each row is filled from column position $first + 1$ to $c - following$. Whenever a positive row is encountered, the counter *following* is decreased by one before filling the row. After filling the row, the column position of the last non-falsified cell is compared with the counter *last*. If the counter *last* is greater than this position, the second pass will be executed. After verifying the condition for the second pass, the counters *first* and *last* are updated respectively with the position of the first non-falsified cell and the

last non-falsified cell of the actual row. When encountering an optional or negated row, the counters stay unchanged and the cells of the row are filled in the range $(\mathit{first}+1, c-\mathit{following})$.

If the second pass condition is not satisfied for any row, the second pass is omitted. Otherwise, the matrix is traversed in reverse order with the counter *last* indicating the column position of the last non-falsified cell of the next following positive row. The counter is initialized with $c+1$, and as the rows are processed in reverse order, the counter is updated every time a positive row is encountered. For each row, the cells with positions greater than or equal to *last* are dropped.

The second pass does not prevent the unification of any query and construct term pair. It frees memory reserved by superfluous matrix cells, and prevents the path generation algorithm from running into dead ends.

The gain from this optimization is the following. In the worst case, it reduces the number of recursive unification calls by a quadratic factor in the query list size. The worst case is that every i-th query term matches the i-th construct term and the second pass does not apply. In this case, a lower-left region of the matrix can be dropped, whose size is $(q-1)+(q-2)+\ldots+1$, that is $\frac{q\cdot(q-1)}{2}$ - assuming that no optional nor negated query term occurs. Due to the usage of the *following* counter in the matrix filling scheme, there is a second upper-right region with unfilled cells. This region has the same size, so that the number of overall dropped cells is $q \cdot (q-1)$ in the worst case. When considering optional and negated subterms as well, the number of unfilled cells depends on the position of these terms in the query term list. However, the number of unfilled cells is obviously still in $O(p^2)$, if p is the number of positive query terms.

Example 28 (Worst Case of the Optimization) *Assume the query term list* `[[ a, b, c, d ]]` *and the construct term list* `[ a, b, c, d, a, b, c ]`. *Unfilled matrix cells are marked with dots. The unification matrix is*

$$\begin{pmatrix} \mathit{True} & \mathit{False} & \mathit{False} & \mathit{False} & . & . & . \\ . & \mathit{True} & \mathit{False} & \mathit{False} & \mathit{False} & . & . \\ . & . & \mathit{True} & \mathit{False} & \mathit{False} & \mathit{False} & . \\ . & . & . & \mathit{True} & \mathit{False} & \mathit{False} & \mathit{False} \end{pmatrix}$$

In the first row, the last three cells stay unfilled, since three positive rows are following. In the second row, the first cell is left empty, since the minimal match position in the previous row is 1. *The last two cells are dropped, too. In the last row, the first three cells are empty because the first position of a match in the previous positive row is* 3. *Altogether,* $4 \cdot (4-1) = 12$ *unifications are avoided.*

The next example shows the optimization with optional and negated subterms. Assume the query term list `[[ a, without a, b, optional b, c ]]` *and the construct term list* `[ a, b, a, b, c, b, c ]`. *The unification matrix for this query is*

$$\begin{array}{c} + \\ - \\ + \\ ? \\ + \end{array} \begin{pmatrix} \mathit{True} & \mathit{False} & \mathit{True} & \mathit{False} & \mathit{False} & . & . \\ . & \mathit{False} & \mathit{True} & \mathit{False} & \mathit{False} & . & . \\ . & \mathit{True} & \mathit{False} & \mathit{True} & \mathit{False} & \mathit{True} & . \\ . & . & \mathit{False} & \mathit{True} & \mathit{False} & \mathit{True} & . \\ . & . & \mathit{False} & \mathit{False} & \mathit{True} & \mathit{False} & \mathit{True} \end{pmatrix}$$

The first row is constrained at the end due to the two following positive terms. The second row (which is negated) is constrained at the beginning due to the position of the first match from the last positive row, which is 1*. Furthermore, the last two cells stay unfilled. The third row is positive, so that there is only one positive term afterwards. The last cell hence stays unfilled. The last positive row is still the first row, so that only the first cell stays unfilled. The third row is the last positive row for the fourth row, so that the first two cells stay unfilled. There is still one positive term following the fourth term, so that the last cell also stays unfilled. The fifth term is positive and has no following positive term, so that the two first cells of the row stay unfilled.*

In the best case however, the refined filling scheme falsifies the whole matrix. The best case is reached when a positive row is created that has no non-falsified cells because of dropping the n first and the m last cells. Since there is no match for this query term, the FILL-MATRIX algorithm can immediately return the failure as the unification result.

If the first row is positive and contains no non-falsified cells, the number of recursive calls to the unification algorithm is $d - p + 1$. This is the lowest number of recursive calls necessary to falsify the whole unification matrix.

Example 29 (Best Case of the Optimization) *Assume the query term list* `[[ a, b ]]` *and the data term* `[ b, b,..., b, a, a ]` *with* n `b` *subterms. The first query term matches the first* `a` *and drops the last* `a`*, since there a positive subterm comes afterwards. The second query term only takes the last data term. The result of the unification is False so that the whole evaluation fails. The number of dropped cells was* 1 *for the first query term and* $n + 1$ *for the second, which corresponds to the size of the data term list. The number of unifications was* $n + 1$ *for the first query term and* 1 *for the second, which is again* $n + 2$*.*

Another less drastic example of the best case is the unification of the query term list `[[ a, b, c, d ]]` *with the data term list* `[ b, c, a, d, b, d, c ]`*.*

$$\begin{pmatrix} \mathit{False} & \mathit{False} & \mathit{True} & \mathit{False} & . & . & . \\ . & . & . & \mathit{False} & \mathit{True} & . & . \\ . & . & . & . & . & \mathit{False} & . \\ . & . & . & . & . & . & . \end{pmatrix}$$

The last row has no data term to match due to the constraints.

To summarize these results in a table, the improvements for a $q \times c$ unification matrix with p positive query terms are the following.

Case	Unoptimized	Worst case optimized	Best case optimized
Recursive calls	$c \cdot q$	$c \cdot q - p^2$	$c - p + 1$

Table 9: Recursive Call Reduction in a $q \times c$ Matrix with p Positive Query Terms

Since the reduction of recursive calls depends heavily on the data and query, investigations on use cases for partial and ordered query terms seem appropriate. However, since the cost for this optimization is rather low, it is valuable to apply it.

4.2.2 Total and Ordered Queries: Preventing Recursive Calls by Surjectivity and Monotonicity

In a total and ordered query term without optional query terms, there is only one valid path through the matrix. It is hence obvious that only the cells on the matrix diagonal are part of the path. The number of recursive calls can be hence reduced from $c \cdot q$ (which is in $O(q^2)$ since $c \leq q$) to q. If optional subterms occur, cells below the matrix diagonal become candidates for valid matrix paths, since it is possible to drop the optional rows and leave the matrix diagonal.

Computing the range of required cells in a row works as follows. Let $(lower, upper)$ be the range of the previous row. If the previous row is an optional row, the range for the actual row becomes $(lower, upper + 1)$. If the row is positive, the range becomes $(lower + 1, upper + 1)$. The range for the first row is obviously $(1, 1)$. If all rows are optional, this filling algorithm still leaves the upper half above the diagonal of the matrix unfilled, so that the fill scheme calls UNIFY at most $c \cdot q - \frac{c \cdot (c-1)}{2}$ times. This is still in $O(q^2)$.

Section 4.3.1 however discusses a path generation algorithm that allows a further refinement of the matrix filling scheme. The path generation algorithm does not generate path prefixes that cannot be extended to surjective paths. Informally, all paths that depart too far below or above the matrix diagonal from cell $(1, 1)$ to cell (q, c) to be a surjective path will be dropped.

Example 30 (Matrix Filling Scheme Refinement) *Take a query with* 8 *optional query terms and a construct term list with length* 6*. The above matrix filling scheme leads to the following matrix*

$$\begin{array}{c} ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{array} \begin{pmatrix} a_{11} & . & . & . & . & . \\ a_{21} & a_{22} & . & . & . & . \\ a_{31} & a_{32} & a_{33} & . & . & . \\ a_{41} & a_{42} & a_{43} & a_{44} & . & . \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & . \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} \end{pmatrix}$$

Note that all paths that do not end with the cells a_{66}, a_{76} or a_{86} are non-surjective. In fact, the cells that contribute to a surjective paths are the following.

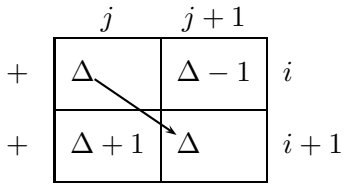
$$\begin{array}{c} ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{array} \begin{pmatrix} a_{11} & . & . & . & . & . \\ a_{21} & a_{22} & . & . & . & . \\ a_{31} & a_{32} & a_{33} & . & . & . \\ . & a_{42} & a_{43} & a_{44} & . & . \\ . & . & a_{53} & a_{54} & a_{55} & . \\ . & . & . & a_{64} & a_{65} & a_{66} \\ . & . & . & . & a_{75} & a_{76} \\ . & . & . & . & . & a_{86} \end{pmatrix}$$

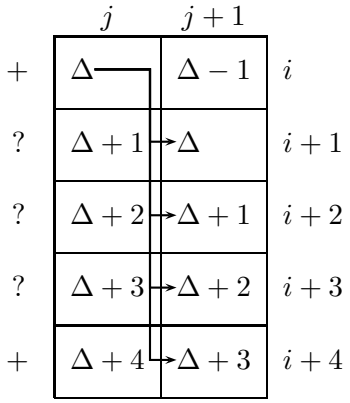
The matrix filling scheme only needs to fill these cells.

It is hence possible to refine the matrix filling scheme above. In order to do so, consider the value $\Delta_{end} = q - c$ which specifies the number of optional terms that must be dropped

altogether. If a path drops more than Δ_{end} optional terms, it is no surjective mapping, since not all construct terms were matched by query terms. If a path drops less than Δ_{end} optional terms, there must be at least one positive query term that has no match candidate. Hence, the path is not total on the positive query terms.

Each cell with position (i, j) is associated with a Δ-value equal to $\Delta = i - j$. Based on the Δ value of a cell and the number o_f of following optional rows, it is possible to decide whether a cell can be part of a valid path. Note that paths taking a cell with a given Δ value can only choose cells with equal or greater Δ-value.





Skipping n optional rows increases the Δ-value of the next path position by n. To keep a Δ-value unchanged, no optional rows may be skipped. A path must hence be extended with cells that have Δ-values greater than or equal to the the last path position.

On the one hand, if $\Delta > \Delta_{end}$, all paths through this cell will drop more than the allowed number of optional terms Δ_{end}. If on the other hand $\Delta + o_f < \Delta_{end}$, there is no possibility to reach Δ_{end} with this path, because even if all following optional terms are dropped, the Δ-value of the paths through this cell can never reach Δ_{end}. The cells that can contribute to the monotone surjective paths have hence Δ-values in the range $\Delta_{end} - o_f \leq \Delta \leq \Delta_{end}$.

Together with the matrix filling scheme above, this leads to the following refined matrix filling scheme. Each row is filled only in a range $(lower, upper)$. The range is initialized with $(1, 1)$. This range holds for the first row. Furthermore, the values $\Delta_{\leftarrow}(i) = i - lower$ and $\Delta_{\rightarrow}(i) = i - upper$ are defined.

Before the row is filled, it is verified whether the previous row was optional and whether $\Delta_{\leftarrow}(i) < \Delta_{end}$ holds for the actual row. If this condition applies, *lower* stays unchanged or is increased by one otherwise. The second condition to be checked before filling the actual row concerns the upper range bound. If the actual row is optional and $\Delta_{\rightarrow}(i) + o_f < \Delta_{end}$, then *upper* stays unchanged. Otherwise, it is increased by one. The row is then filled in the range $(lower, upper)$.

To estimate the worst case complexity of the delta method, it is assumed that $q = o$, i.e., that all query terms are optional. In this case, the number of cells to fill is as large as possible. The number of cells to fill can be estimated as follows: there are c columns to fill, each one must contain $\Delta_{end} + 1$ filled cells, which is $q + 1 - c$. The number of filled cells is $(q + 1 - c) \cdot c$. The worst case number of matrix cells to fill is hence in $O(q \cdot c - c^2)$.

The best case of the optimization occurs when Δ_{end} is equal to the number of optional query terms. In this case, all optional rows are dropped and only $q - o$ cells are filled.

Case	Unoptimized	Worst case optimized	Best case optimized
Recursive calls	$q \cdot c$	$O(q \cdot c - c^2)$	$q - o$

Table 10: Recursive Calls in a $q \times c$ Matrix with o Optional Query Terms

The effort for this optimization is very low. Realizing the filling scheme requires two integer counters and verifying two conditions before filling each matrix row. This optimization gain is quite important, as it reduces the number of recursive unification calls considerably.

```
FILL-TOTAL-ORDERED(query, construct)
 1  Δ_end ← size(query) − size(construct)
 2  o_f ← optionalTerms(query)
 3  lower ← 1
 4  upper ← 1
 5  matrix ← nil
 6  for r ← 1 to size(query)
 7  do row ← nil
 8     if optional(query[r − 1]) and (r − lower < Δ_end)
 9       then lower ← lower − 1
10     for c ← lower to upper
11     do position ← (r, c)
12        solutions ← UNIFY(query[r], construct[c])
13        row ← row + [(position, solutions)]
14     if optional(query[r]
15       then o_f ← o_f − 1
16     if optional(query[r]) and (r − upper + o_f ≥ Δ_end)
17       then upper ← upper + 1
18     matrix ← matrix + [row]
```

4.2.3 Unordered Queries: Preventing Recursive Calls by Injectivity

For unordered queries (regardless whether they are partial or total), the matrix filling algorithm cannot be improved in such a dramatic way. It is however possible to reduce the number of recursive unification calls slightly with a very cheap technique. In some special cases however, this optimization may lead to a noticeable reduction of query runtime.

Whenever a positive matrix row contains only one single unfalsified matrix cell, the whole column is reserved for this row. Every successful path must take this cell, and the injectivity constraint forbids to take the same column a second time. Therefore, the cells of this column do not need to be filled after and even before the reserving row. In a second reverse pass, the cells in the same column as the reserving cell are removed. Note that this can entail other column reservations.

Because the column reservation rule is applicable to positive terms only, it is sensible to reorder the query term list in such a way that all positive query terms occur before all optional or negated query terms. Thus, it is possible to reduce the number of cells for optional and

negated rows, which again leads to a reduction of the number of paths.

In the worst case, this optimization does not reduce the number of recursive calls at all. If no row fulfills the above mentioned condition no columns are reserved. In the best case, this optimization reduces the number of recursive calls by $\frac{p \cdot (p-1)}{2}$. This occurs when every row satisfies the condition for column reservation.

Case	Unoptimized	Worst case optimized	Best case optimized
Recursive calls	$c \cdot q$	$c \cdot q$	$c \cdot q - \frac{p \cdot (p-1)}{2}$

Table 11: Recursive Calls in a $q \times c$ Matrix with q_p Positive Query Terms

The amount of space needed for this optimization is in $O(q)$, since at most q positions may be reserved. The lookup can be implemented with a hashtable with a constant lookup and insertion complexity, so that the cost for this optimization is marginal. However, the table above shows that the reduction of unification call can be reduced considerably, which is especially useful when negated or optional query terms are involved.

```
FILL-UNORDERED(query, construct)
 1  matrix ← nil
 2  reserved ← nil
 3  row_pos ← 1
 4  for c in query
 5  do row ← nil
 6     col_pos ← 1
 7     for d in data
 8     do if not col_pos ∈ reserved
 9           then solutions ← UNIFY(q, c)
10                position ← (row_pos, col_pos)
11                row ← row + [(position, solutions)]
12                col_pos ← col_pos + 1if size(row) = 1
13                                    then col_pos ← singleColumnPosition(row)
14                                         reserved ← reserved + [col_pos]
15     matrix ← matrix + [row]
16     row_pos ← col_pos + 1
17  return matrix
```

4.2.4 Immediate Query Failure

It is possible to detect the failure of a query while filling the unification matrix or immediately after filling it.

The first failure condition is checked during the matrix filling. Whenever a positive matrix row contains no non-empty or non-falsified cells, the unification fails. Since a positive row represents a query term, this case means that a positive query term has no correspondent construct term to match with. Such a query fails obviously.

The second failure condition is valid for total queries only. In a total query, the algorithm must also ensure that every path is surjective. For this reason, the unification fails immediately if the completely filled unification matrix contains a column that has no non-falsified cells.

The optimizations that are presented in these sections demonstrate how to use path constraints to reduce the number of recursive unification calls. Although path constraints are specified at the level of single paths, it is possible to deduce constraints that hold for all paths. With these global constraints, it it possible to reduce the number of recursive unification calls, filled matrix cells and even the number of false paths the path generation algorithm creates and prunes. In partial and ordered queries, the combination of falsified cell constraints and path constraints even reduces the number of filled matrix cells if they are considered in combination.

After this discussion of matrix filling optimization, the next section demonstrates the optimization of the path generation algorithm.

4.3 Optimizing Path Generation

In the following, the path generation algorithm is split in two cases, since some algorithmic improvements are applicable only to total queries and others only to partial ones. The reason for this lies in the fact that negated terms do not occur in total queries and that optional subterms are translated differently in total queries compared to partial ones.

4.3.1 Total Queries: The Delta Method

As already announced in section 4.2.2, it is possible to reduce the number of generated surjective monotonic paths. Moreover, it is also possible to reduce the number of bijective paths generated by the algorithm MATRIX-PATHS. The refined algorithm counts the number of optional terms that are dropped for each path, and compares the result with the total number of optional terms to drop.

Consider the query term sequence $\langle t_1, \ldots, t_q \rangle$ with o optional query subterms and $p = q - o$ positive subterms and the data term sequence $\langle s_1, \ldots, s_c \rangle$. The length of the construct term list c must fulfill the condition $p \leq c \leq q$. Furthermore, the injectivity constraint imposes that exactly $o' = c - p$ optional query term must find a match. Consequently, the following four cases need to be considered:

1. If $c = p$, the positive terms must match all construct terms. All optional query terms are considered nonexistent.
2. If $c = q$, then $o' = o$, that is, every optional subterm must find a match. All optional query terms are considered positive.
3. If $p < c < p + o$ and the query is unordered, generate all $o'! \cdot \binom{o}{o'}$ variations of optional subterms instead of

$$\sum_{i=0}^{o'} \left[i! \cdot \binom{o}{i} \cdot \binom{o'}{i} \right].$$

4. If $p < c < p + o$ and the query is ordered, generate all $\binom{o}{o'}$ combinations of optional subterms instead of
$$\sum_{i=0}^{o'} \binom{o}{i}.$$

The cases 1 and 2 already cover all cases with zero or one optional query subterm without any special treatment of optional query terms within the matrix method being necessary.

If the algorithm furthermore considers the number $o_m \leq o$ of optional subterms with a non-falsified match in the data term sequence $\langle s_1, \ldots, s_c \rangle$, the number of combinations to be considered can be reduced even more.

1. If $o_m < o'$, the query fails.
2. If $d = p + o_m$, each of the o_m optional subterms is treated as mandatory.
3. if $p < c < p + o_m$ and the query is unordered, generate $o'! \cdot \binom{o_m}{o'}$ variations of optional subterms instead of $o'! \cdot \binom{o}{o'}$.
4. if $p < c < p + o_m$ and the query is ordered, generate $\binom{o_m}{o'}$ combinations of optional subterms instead of $\binom{o}{o'}$.

The creation of the $\binom{o_m}{o'}$ combinations is then a combinatoric exercise similar to the matrix filling scheme for total and ordered queries. Every surjective path will have to omit $\Delta_{end} = o_m - o'$ optional terms. This means that for every solution where Δ optional terms were dropped, the following conditions are tested whenever an optional row is encountered:

1. $\Delta + 1 \leq \Delta_{end}$. If the equation holds, the paths dropping the optional row do not exceed the number of optional rows that must be dropped altogether.
2. $\Delta + o_f \geq \Delta_{end}$, where o_f is the number of optional rows following the current one. If the equation holds, the paths matching a cell of the optional row that do not increase Δ are valid. This is due to the fact that there are o_f following optional rows that can be dropped to reach $\Delta = \Delta_{end}$.

Considering the worst case of path reduction, it obvious that the largest number of existing paths is given when $o_m = o$ and the binomial coefficient reaches its maximum, that is $o' = \frac{o}{2}$.

Example 31 (Worst Case of Optimization) *Assume that the query term list* { `optional a, optional a, optional a, optional a` } *is to be unified with the data term list* `{ a, a }`. *Since* $o_m = o = 4$ *and* $o' = \frac{o}{2} = 2$, *this is one of the worst case examples for optimization.*

Case and Algorithm	Generated Monotonic Paths	Generated Injective Paths
Unoptimized	$\sum_{i=0}^{o} \binom{o}{i}$	$p! \cdot \binom{c}{p} \cdot \sum_{i=0}^{o} i! \cdot \binom{o}{i}$
Worst Case, Optimized ($o = q, c = \frac{o}{2}$)	$\binom{o}{\frac{o}{2}}$	$(\frac{o}{2})! \cdot \binom{o}{\frac{o}{2}}$
Best Case, Optimized	1	$p!$

Table 12: Generated Paths with the Delta Method

The number of paths to compute is $2! \binom{4}{2} = 12$*, and the paths are*

$$\{(1,1),(2,2)\},\{(1,1),(3,2)\},\{(1,1),(4,2)\},\{(2,1),(3,2)\},\{(2,1),(4,2)\},\{(3,1),(4,2)\},$$
$$\{(1,2),(2,1)\},\{(1,2),(3,1)\},\{(1,2),(4,1)\},\{(2,2),(3,1)\},\{(2,2),(4,1)\},\{(3,2),(4,1)\}$$

However, there are $0! \cdot \binom{4}{0} \cdot \binom{2}{0} + 1! \cdot \binom{4}{1} \cdot \binom{2}{1} = 9$ *paths not considered by the optimized path generation algorithm.*

The following table continues the series of the example above with queries containing o subterms `optional a` *and containing* $o' = \frac{o}{2}$ *data terms of the form* `a`*.*

Number of Optional Terms(o)	2	4	8	16
Data Size (o′)	1	2	4	8
Unoptimized Unordered Paths	2	21	3,393	1,174,226,049
Optimized Unordered Paths	2	12	1,680	518,918,400
Pruned Unordered Paths (percent)	0.00%	42.85%	50.46%	55.80%

The number of paths still grows exponentially with the size of the problem o, so that the complexity of the problem is not reduced. However, the number of paths is roughly cut by half.

Example 32 (Best Case of Optimization) *In the best case, the computation time can be reduced to be linear in the number of query terms. Assume the query term list* `[ a, optional a, optional a ]` *and the data term list* `[ a, a, a ]`*. Since* $q = c$*, all optional terms are handled as positive terms. The algorithm generates only one path. Similarly, if the query is unified with the data term list* `[ a ]`*, all optional terms are dropped and the path generation algorithm again creates only one path.*

The cost for this optimization is again in $O(q)$, since the number of following optional terms must be computed.

The optimization of the path generation algorithm for total queries is one of the most important optimizations investigated so far. The average runtime reduction that is reached with this algorithmic optimization is vital in queries with more than one optional term.

4.3.2 Partial Queries: Greedy Algorithm

This section presents a greedy algorithm approach that is not applicable in the general case. This section furthermore discusses sufficient conditions for the application of the greedy algorithm. The verification of these conditions requires an analysis of the unification matrix. For this reason, the choice of the path generation algorithm can only be made at runtime. We believe that the conditions proposed here are satisfied for most unordered partial query cases, but only for a small set of ordered partial queries. The greedy algorithm itself leads to a complexity reduction from originally exponential size in the number of negated and optional terms to linear size in this number, as the experimental evaluation shows clearly (see section 4.4).

Handling negated query terms appropriately involves an exponential computation effort. For a given mapping of the positive query terms to construct terms, there are exponentially many mapping extensions of negated query terms to construct terms. Assuming a $q \times c$ matrix with n negative query terms and no optional terms, the number of path extensions for one positive path is in $O(2^n)$.

The greedy algorithm is based on the observation that the constraint calculus requires considering all possible extensions $\pi' \in E(\pi)$ of a given positive mapping π. The constraints that arise from these extensions $\pi' \in E(\pi)$ are taken into one conjunction in the decomposition rule conclusion :

$$\ldots \wedge \bigwedge_{\pi' \in E(\pi)} \bigwedge_{(q_-,c) \in \pi'} q_- \preceq_{su} c$$

This formula however contains several duplicate simulation constraints, since a single pair (q, c) usually occurs in several path extensions. These duplicates are redundant, since the absorption rule $a \wedge a \equiv a$ holds in the boolean algebra. It is hence sufficient to consider the set of all pairs in the set of mapping extensions:

$$\Pi = \bigcup E(\pi) = \{(q, c) \mid \pi' \in E(\pi), (q, c) \in \pi'\}$$

The relation Π is called extension relation. Π is no mapping, since it can map one negative query term to several construct terms. The relation Π is noninjective, as it may map two negative query terms to one construct term. These violations of path constraints are admitted, since the constraint set that stems from Π is still equivalent to the constraint set that stems from $E(\pi)$.

The reason why the algorithm Matrix-Paths needs to keep the path extensions π' in distinct paths is that it computes path extensions in a single pass alongside with the positive path. Since path extension prefixes may be dropped as the positive path prefix is extended, it is not possible to restrict the algorithm to the generation of maximal extension prefixes. The maximal extension prefixes may be incompatible with the match of a following positive query term and may be dropped. In such cases, the maximal path extension prefixes are discarded and previously nonmaximal paths become maximal.

In the algorithm Matrix-Paths, the extension incompatibility can also occur between two negative query terms. The maximal extension prefix matching the first query term may prevent extension prefixes to match the second query term due to monotonicity or injectivity path constraints. The greedy variant of the path generation algorithm will compute the extension relation Π. Extension incompatibility is not considered in Π; as long as a given cell

position is compatible with the positive path, there is a path extension containing this cell and it is hence in the relation Π.

Proposition 3 (Greedy Condition for Unordered Queries) *Let M be a $q \times c$ matrix. The greedy algorithm is applicable for unordered queries, if for all non-positive (i.e. optional or negated) rows $i \in \{1, \ldots, q\}$ and all $j \in \{1, \ldots, c\}$ with $M_{ij} \neq$ False it holds that there is no following non-negative (i.e. positive or optional) row $i' \in \{i+1, \ldots, q\}$ with a cell $M_{i'j} \neq$ False. The algorithm to compute the unordered greedy condition is as follows.*

```
UNORDERED-GREEDY-CONDITION(matrix)
 1  for row in matrix
 2  do if negative(row) or optional(row)
 3        then for cell in row
 4              do if not falsified(cell)
 5                    then (i, j) ← position(cell)
 6                         for row′ in following(row)
 7                         do if positive(row′) or optional(row′)
 8                               then cell′ ← row′[j]
 9                                    if not falsified(cell′)
10                                      then return False
11  return True
```

Proposition 4 (Greedy Condition for Ordered Queries) *Let M be a $q \times c$ matrix. The greedy algorithm is applicable for ordered queries, if for all non-positive rows $i \in \{1, \ldots, q\}$ and all $j \in \{1, \ldots, c\}$ with $M_{ij} \neq$ False it holds that there is no following non-negative row $i' \in \{i+1, \ldots, q\}$ and column $j' \in \{1, \ldots, j\}$ with a cell $M_{i'j'} \neq$ False. The algorithm to compute the ordered greedy condition reads as follows.*

```
ORDERED-GREEDY-CONDITION(matrix)
 1  for row in matrix
 2  do if negative(row) or optional(row)
 3        then for cell in row
 4              do if not falsified(cell)
 5                    then (i, j) ← position(cell)
 6                         for row′ in following(row)
 7                         do if positive(row′) or optional(row′)
 8                               then for cell′ in row′
 9                                    do (i′, j′) ← position(cell′)
10                                       if j′ ≤ j and not falsified(cell′)
11                                         then return False
12  return True
```

There are two sufficient conditions that guarantee the absence of positive extension incompatibilities, one for non-surjective unordered queries and one for non-surjective ordered queries.

These sufficient conditions guarantee that no element of the extension relation Π will become incompatible with the extension of the positive path prefix if it was compatible as the element was added to Π.

The appropriate greedy condition for the query matrix ensures that no generated path extension prefix generated by the MATRIX-PATHS algorithm is dropped. In this case, it is even possible to compute the relation Π in a single pass.

Speaking in terms of query and construct terms, the sufficient condition guarantees the following: If it is possible to map a negated query term q_i to a construct term c_j (respecting the injectivity and order constraints imposed by the matching of the previous query terms $q_1, \ldots, q_{i-1}$), there is no following positive query term in $q_{i+1}, \ldots, q_n$ that can also unify with the construct term c_j. For ordered partial queries, the greedy condition furthermore guarantees that the positive queries in $q_{i+1}, \ldots, q_n$ do not unify with any construct term in $c_1, \ldots, c_j$. For this reason, there is a mapping extension $\pi' \in E(\pi)$ that contains the pair (q_i, c_j), and the cell position (i, j) can be taken into the relation Π.

```
GREEDY-EXTEND-POSITIVE(row, paths)
1  new_paths ← ∅
2  for (path, Π) in paths
3  do for cell in row
4      do pos ← position(cell)
5          if compatible(path, pos)
6             then new_paths ← new_paths ∪ {(path ∪ {pos}, Π)}
7  return new_paths

GREEDY-EXTEND-NEGATIVE(row, paths)
1  new_paths ← copy(paths)
2  for (path, Π) in new_paths
3  do for cell in row
4      do pos ← position(cell)
5          if compatible(path, pos)
6             then Π ← Π ∪ {pos}
7  return new_paths
```

Note that still every positive path has its own extension relation Π. The size of the relation Π is bound by $O((o+n) \cdot c)$, where c is the number of construct terms and o and n are the number of optional and negated query terms respectively. This upper bound is estimated by assuming that all optional and negative query terms match with all construct terms.

Example 33 (Greedy Algorithm) *The first example presented here shows a matrix satisfying the greedy condition for unordered queries. Assume that the query is unordered and partial and that the second and third query terms are negated. Assume furthermore that the matrix reads as follows:*

$$\begin{array}{c} + \\ - \\ - \\ + \end{array} \begin{pmatrix} . & a_{12} & a_{13} & . & a_{15} & . & a_{17} \\ . & . & a_{23} & a_{24} & a_{25} & . & . \\ a_{31} & . & . & . & . & . & . \\ . & . & . & . & . & a_{46} & a_{47} \end{pmatrix}$$

Verifying the greedy condition for this matrix leads to the following considerations. The first negative row to be checked is row number two. It has non-negative cells at position $(2,3),(2,4)$ *and* $(2,5)$*. For all these cells, the cells in the same columns of the following rows are unpopulated. There are no non-false entries at* $(3,3),(3,4),(3,5)$*, neither are there at* $(4,3),(4,4)$ *and* $(4,5)$ *and at* $(5,3),(5,4)$ *and* $(5,5)$*. The second negative row to be considered is row number three. This row contains a single non-negative entry at* $(3,1)$*. The last row however has no entry at position* $(4,1)$*, so that this matrix satisfies the greedy condition for unordered queries.*

Applying the greedy variants of row processing above to this matrix lead to the following computation.

The algorithm starts with the path set $\{(\emptyset,\emptyset)\}$*. The first row is positive, so that* Greedy-Extend-Positive *applies. All cells from the first row are compatible with the empty set. The new path set hence becomes*

$$\{(\{(1,2)\},\emptyset),(\{(1,3)\},\emptyset),(\{(1,5)\},\emptyset),(\{(1,7)\},\emptyset)\}$$

Since the second row is negative, the algorithm Greedy-Extend-Negative *applies. The outer for-loop iterates over a copy of the existing path prefixes. Starting with e.g.* $(\{(1,2)\},\emptyset)$*, the inner for-loop adds all cell positions which are compatible with the positive path prefix to the set* Π*. Since all cells of the negative row are compatible with* $\{(1,2)\}$*, the new path becomes* $(\{(1,2)\},\{(2,3),(2,4),(2,5)\})$*. The other three path prefixes are extended similarly, but since the cell position* $(2,3)$ *is incompatible with the path* $\{(1,3)\}$*, the set* Π *of the second path is not extended by the position* $(2,3)$*. The cell* $(2,5)$ *is missing in the third path. The paths after the second row are hence*

$$\begin{array}{l}\{(\{(1,2)\},\{(2,3),(2,4),(2,5)\}),\\ (\{(1,3)\},\{(2,4),(2,5)\}),\\ (\{(1,5)\},\{(2,3),(2,4)\}),\\ (\{(1,7)\},\{(2,3),(2,4),(2,5)\})\}\end{array}$$

Since there are no incompatibilities with the cell position $(3,1)$*, processing the third row extends each set* Π *by* $(3,1)$*.*

$$\begin{array}{l}\{(\{(1,2)\},\{(2,3),(2,4),(2,5),(3,1)\}),\\ (\{(1,3)\},\{(2,4),(2,5),(3,1)\}),\\ (\{(1,5)\},\{(2,3),(2,4),(3,1)\}),\\ (\{(1,7)\},\{(2,3),(2,4),(2,5),(3,1)\})\}\end{array}$$

The last row is positive again, so that it extends the positive path by its cell positions. This leads to the following final set of matrix paths:

$$\begin{array}{l}\{(\{(1,2),(4,6)\},\{(2,3),(2,4),(2,5),(3,1)\}),\\ (\{(1,3),(4,6)\},\{(2,4),(2,5),(3,1)\}),\\ (\{(1,5),(4,6)\},\{(2,3),(2,4),(3,1)\}),\\ (\{(1,2),(4,7)\},\{(2,3),(2,4),(2,5),(3,1)\}),\\ (\{(1,3),(4,7)\},\{(2,4),(2,5),(3,1)\}),\\ (\{(1,5),(4,7)\},\{(2,3),(2,4),(3,1)\})\}\end{array}$$

Take a second matrix (again, with negative second and third rows)

$$\begin{matrix} + \\ - \\ - \\ + \end{matrix} \begin{pmatrix} . & a_{12} & a_{13} & . & a_{15} & . & a_{17} \\ . & . & a_{23} & a_{24} & a_{25} & . & . \\ a_{31} & . & . & . & . & . & . \\ a_{41} & . & . & . & . & a_{46} & a_{47} \end{pmatrix}$$

As the reader may see right away, the greedy condition does not hold because of negative third row. It contains the position $(3,1)$*, while the cell* $(4,1)$ *in the fourth row is not falsified. For this reason, this matrix does not satisfy the greedy condition for unordered queries.*

Take another matrix to test the greedy condition for ordered queries. Assume that the second row is negative, while the first and third are positive.

$$\begin{matrix} + \\ - \\ + \end{matrix} \begin{pmatrix} a_{11} & . & . & . & a_{15} & . & . \\ a_{21} & . & . & . & a_{25} & . & . \\ . & . & . & . & . & a_{36} & a_{37} \end{pmatrix}$$

The greedy condition for ordered queries requires to check each cell in the negative rows, which is the case for the second one. The cell at position $(2,1)$ *satisfies the greedy condition: Since there is only one following row, the cells of this row that are in the range from* $(3,1)$ *to* $(3,1)$ *must be verified. All cells in this range are false, so that the second cell of the row, which is at position* $(2,5)$*, is investigated. Since again, there are no non-falsified cells in the range from* $(3,1)$ *to* $(3,5)$*, the greedy algorithm is applicable.*

Assume a fourth matrix, with again the second row being negative.

$$\begin{matrix} + \\ - \\ + \end{matrix} \begin{pmatrix} a_{11} & . & . & . & a_{15} & . & . \\ a_{21} & . & . & . & a_{25} & . & . \\ . & a_{32} & a_{33} & a_{34} & . & a_{36} & a_{37} \end{pmatrix}$$

This matrix does not satisfy the greedy condition for ordered queries, since the verification of the condition for the cell $(2,5)$ *fails; there are cells in the range from* $(1,3)$ *to* $(3,5)$ *that are non-falsified, namely at position* $(3,2), (3,3)$ *and* $(3,4)$*. Hence, the greedy algorithm is not applicable.*

Unfortunately, matrices of the fourth kind are created in the evaluation of common ordered queries with negation such as

```
html[[
    body[[
        var Title as h1[[  ]],
        without  h1[[ ]],
        var Paragraph as p[[ ]],
    ]]
]]
```

Hence, it can be expected that the greedy condition for ordered queries is rarely satisfied while the greedy condition for unordered queries will mostly hold.

For unordered query terms, the greedy condition hits almost every time if the rows are reordered. The rows must be reordered in order to avoid the occurrence of an optional or negative query term before any positive query term, and in order to prevent no negative query terms from being before any optional query term. In this case, extension incompatibility can only occur between optional query terms. Since the optional query terms are processed twice (as positive and negative) they extend both the positive path and the extension relation Π, it is impossible to reorder optional terms in such a way that extension incompatibility can be excluded statically.

The greedy algorithm reduces the number of generated path extensions drastically. The general algorithm generates a number of path extensions in $O(2^(n+o))$ with a total number of cell positions in $O(2^{(n+o)\cdot c})$ for each positive path, while the greedy algorithm generates one single extension relation for each positive path with a number of cell positions in $O((n+o)\cdot c)$. n is the number of negative query terms, and o is the number of optional query terms in the query list.

This section has shown that under certain conditions, the complexity of negation handling in the MATRIX-PATHS algorithm can be reduced drastically. The greedy algorithm can be applied, if the query matrix satisfies the greedy condition. Unfortunately, the greedy condition for ordered queries is rarely satisfied. The greedy algorithm applies especially for unordered queries, since reordering the query term list results in almost every query matrix satisfying the greedy condition.

4.3.3 Extending the Greedy Algorithm with Backtracking

The greedy algorithm presented in the previous section is no solution for the general case. This section demonstrates how the greedy algorithm can be extended with backtracking to get an efficient evaluation algorithm able to handle the cases where the greedy conditions do not hold.

The greedy conditions specified in the previous section ensured that every element added to the extension relation Π never becomes incompatible with a following positive or optional row cell. If an extension incompatibility occurs however, it would also be possible to remove the positions that conflict with a positive path extension from the relation Π. The conflicting positions can be "cut away", just as the inconsistent path extensions were removed in the positive child operator of the Xcerpt query algebra (see section 3.3.5).

Example 34 (Path Extension Backtracking) *Take an ordered query and the path prefix*

$$(p, \Pi) = (\{(1,1),(2,2)\}, \{(3,3),(3,4),(3,5),(3,6)\})$$

that must be extended by a cell in the positive row number four. Assume that the fourth row is

$$(. \quad . \quad a_{43} \quad a_{44} \quad a_{45} \quad a_{46} \quad .)$$

The positive path p is compatible with each of the cells above. Choosing any cell from the fourth row keeps the positive path ordered. The extension Π however contains positions that are incompatible with these extensions of the positive path prefix p. These conflicting positions must be removed from Π. If (i,j) is the position of the new cell, all positions (i',j') with the

property that $j' \geq j$ are removed from Π. Hence, the processing of the positive row should yield the following set.

$$\begin{array}{l}\{(\{(1,1),(2,2),(4,3)\},\emptyset),\\ (\{(1,1),(2,2),(4,4)\},\{(3,3)\}),\\ (\{(1,1),(2,2),(4,5)\},\{(3,3),(3,4)\}),\\ (\{(1,1),(2,2),(4,6)\},\{(3,3),(3,4),(3,5)\})\}\end{array}$$

The backtracking variant of the greedy algorithm is as follows:

```
GREEDY-BACKTRACK-EXTEND-POSITIVE(row, paths)
1  new_paths ← ∅
2  for (path, Π) in paths
3  do for cell in row
4      do pos ← position(cell)
5          if compatible(path, pos)
6            then Π' ← copy(Π)
7                 REMOVE-CONFLICTING(Π', pos)
8                 new_paths ← new_paths ∪ {(path ∪ {pos}, Π')}
9  return new_paths
```

```
GREEDY-BACKTRACK-EXTEND-NEGATIVE(row, paths)
1  new_paths ← copy(paths)
2  for (path, Π) in new_paths
3  do for cell in row
4      do pos ← position(cell)
5          if compatible(path, pos)
6            then Π ← Π ∪ {pos}
7  return new_paths
```

```
REMOVE-CONFLICTING(Π, pos)
1  for pos_neg in Π
2  do if conflicting(pos_neg, pos)
3        then Π ← Π \ {pos_neg}
```

The function $conflicting((i_1, j_1), (i_2, j_2))$ yields the result $True$ in the following two cases:

- The query is unordered and $j_1 = j_2$.
- The query is ordered and $j_1 \geq j_2$.

Using a hashtable with list entries or a sorted map with list entries as implementation for Π leads to a more efficient algorithm than the presented REMOVE-CONFLICTING. Instead of $O(|\Pi|)$, removing the conflicting elements can be reduced to $O(1)$ (but still $O(|\Pi|)$ worst

case complexity) for unordered queries (using a hashtable) and $O(log\ |range(\Pi)|)$ for ordered queries (using a sorted map).

Due to the backtracking extension, the greedy algorithm can be used even in cases where the greedy conditions do not hold. Obviously, the ordered partial queries profit the most from this improvement. Another benefit of the extension is that there is no need to verify the greedy condition anymore. If the greedy condition holds, the algorithm above will never find conflicting elements in Π, but with the algorithmic improvement of hashtables and sorted maps, the overhead is kept low.

Altogether, two improvements of the algorithmic complexity of simulation unification were presented. The first algorithmic improvement demonstrated how surjectivity requirements and the lack of negation can be used to improve the performance of total queries. The second algorithmic improvement showed that a greedy algorithm can lower the runtime drastically. The greedy algorithm was extended to a generally usable algorithm in a second step by introducing a kind of backtracking.

The next section demonstrates the complexity reduction with the help of the Xcerpt prototype and selected Xcerpt programs.

4.4 Preliminary Performance Tests

The runtime reduction through the greedy algorithm is demonstrated in this section. The machine used for the runtime measurement is an AMD Athlon 2600XP processor with one gigabyte of RAM, and Windows XP as operating system. The Xcerpt prototype used is implemented in Java. The JVM version used is the Sun implementation at version `1.5.0_06-b06`.

Each program measurement has been evaluated one hundred times and the overall runtime of these program runs has been divided by 100. In order to avoid measuring the JVM startup and class loading as well, a program has been executed before performing the actual measurement. In this way, the necessary classes are already loaded into the JVM heap and eventual Just-In-Time compilation of bytecode fragments performed.

Each program that is evaluated in the following is parameterized by two factors n and m. The factor n influences the query size, while m influences the data size. The factors n and m determines how often certain term blocks delimited with `<` and `>` braces are repeated. The factor that determine how often the block within `<` and `>` is repeated is noted in braces after the parameterized block. Furthermore, each parameterizing part of the program is colored green. The experimental tests are performed with the following two programs:

```
GOAL
    result{}
FROM
    f{{ < without ai{{ }}, ai{{ }} >(n) }}
END

CONSTRUCT
    f[ < < ai{{ }} >(n) >(m)]
END
```

Program 1

```
GOAL
    result{}
FROM
    f[[ a, < without bi{{ }} >(n), c]]
END

CONSTRUCT
    f[ < a, < bi{{ }} >(n), c{{ }} >(m)]
END
```

Program 2

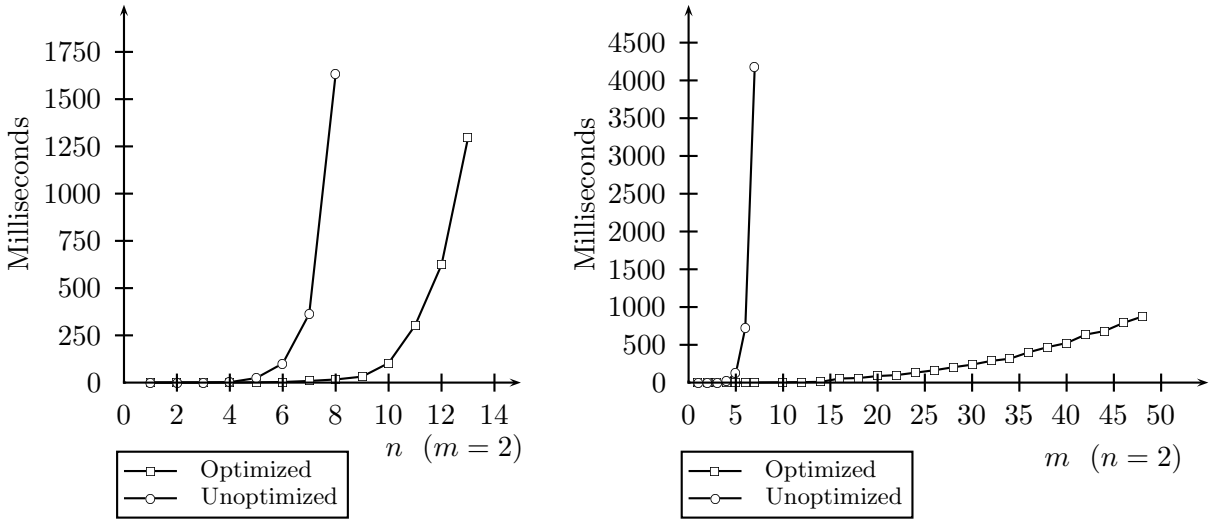


Table 13: Results for Program 1

Each program is measured twice, the first measurement demonstrates the runtime increase with the increase of the n parameter, while the second experiment shows the dependency of the runtime from the m parameter. The second parameter is kept constant while modifying the first.

The implementation of the Xcerpt prototype allows to measure the runtime of program evaluation repeatedly without parsing the program several times.

The experimental results demonstrate clearly that the greedy algorithm is an improvement compared to the basic algorithm. However, as the query size increases in program 1, the evaluation runtime explodes even when applying the greedy algorithm. This is due to the fact that the number of valid matrix paths explodes with n (the query size), as there are always two possibilities to match one positive query term `ai`. The greedy algorithm nevertheless reduces the runtime considerably for query sizes that probably occur in real programs. When m increases however, the complexity reduction of the greedy algorithm becomes apparent.

Program 2 is a program where the number of existing positive paths does not increase with n, but only with m. Here, the runtime of the greedy algorithm stays barely measurable when increasing n, but the runtime of the unoptimized algorithm explodes.

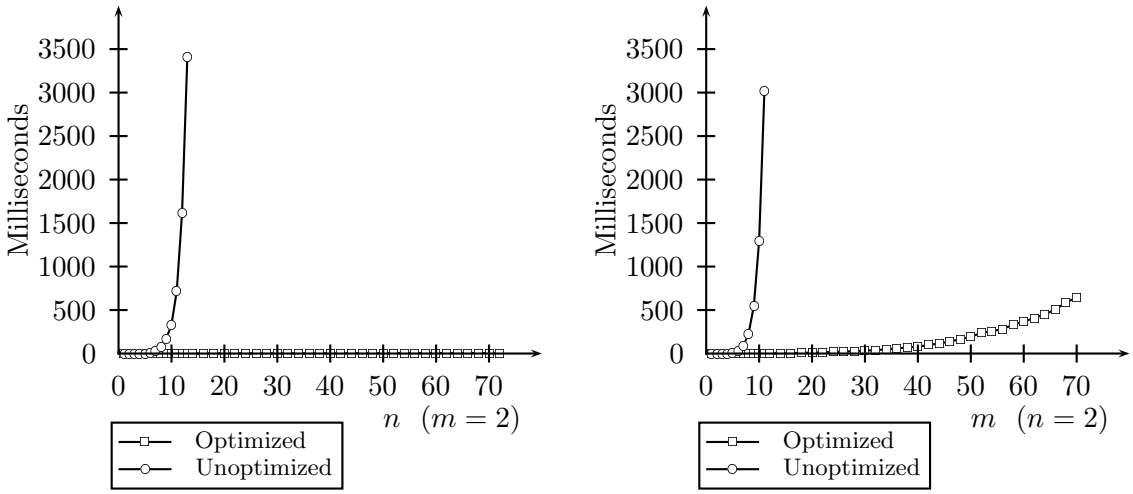


Table 14: Results for Program 2

As m increases, however, the number of existing positive paths increases, and so does the runtime of the optimized algorithm. Again, the runtime of the unoptimized algorithm grows exponentially.

The greedy algorithm is not the best possible algorithm. In the examples above, the use of a divide and conquer algorithm as sketched in section 5.1 would allow to discard the paths earlier. The experiments show very clearly, however, that the greedy algorithm already constitutes a major improvement.

To conclude, two aspects of the matrix method are investigated in this section. The first aspect is matrix filling. The optimization of the matrix filling algorithm involves a reduction of recursive unification calls. These recursive calls can be arbitrarily expensive, so that the reduction of these calls is valuable. It is shown how the matrix path constraints can be used to define refined matrix filling schemes for each kind of query term list. The second aspect that is investigated for optimizations in this section is path generation. Again, the matrix path constraints are used to deduce the possibility of early path pruning. Furthermore, a greedy algorithm with a backtracking extension is introduced. It is demonstrated by preliminary experimental results that the improved algorithm reduces program runtime tremendously when several negated and optional terms are involved. The algorithms that are presented in this section demonstrate that the high level constructs `without` and `optional` can be evaluated with a satisfactory runtime.

However, the investigations on optimization of the matrix method are far from being complete. As stated in the next section, there are still several possible directions to be investigated. The investigation of the matrix method itself opened up several further interesting research directions worth exploring.

5 Future Work

While the definition of the algebra is complete, this thesis also shows further opportunities for interesting research and extensions. Especially, there are many other optimization possibilities

for path generation, some of which are presented here. Another important aspect to improve performance would be to consider the path execution for optimization. This aspect was left aside in this work. A third aspect for Xcerpt optimization would be the investigation of chaining.

5.1 Further Optimizations of Path Generation

Some possible optimizations of the path generation algorithm beyond the presented optimizations are the following:

First, it can be observed that in ordered and partial queries, the pruning of the path extension relation Π is only based on the chosen position in the positive row following the negative rows. Given a negative row, the greedy algorithm chooses to add all cell positions to the extension relation Π that are compatible with the positive path. Assume that in the last positive row, the cell (i_1, j_1) was selected. All compatible cells in the negative rows immediately following this positive row are cells with column positions $j' > j_1$. Assume that the algorithm selects the cell (i_2, j_2). This choice implies that all positions with column position $j' \geq j_2$ must be discarded from Π.[18]

However, if a cell (i', j') of a negated row contains a match without further constraints, any positive path whose path extension Π contains (i', j') will clearly fail, and the path does not need to be generated. This technique would be especially valuable when unifying ordered and partial query lists with construct term lists with repetitive patterns, as for example the query:

```
html[[
    body[[
        var Title as h1[[  ]],
        without  h1[[ ]],
        var Paragraph as p[[ ]],
    ]]
]]
```

with any term that contains more than one title `h1` with following paragraphs `p`.

Applying a divide and conquer scheme to the matrix method may also lead to fruitful results. The matrix can be divided in partitions that do not conflict on the path constraints. For an unordered query for example, the matrix can be divided into partitions containing rows having a non-falsified cell in the same column, since this represents a potential conflict of query terms. One advantage is that the result from each partition can be combined with the other partition results without verifying path constraints. Furthermore, the failure of a single partition leads to the failure of the whole query. For example, whenever negative queries without variables fall into a partition that contains no positive terms, the partition fails obviously, and so does the whole matrix. There might be several such conditions that can be easily verified due to the partition property. Note that the divide and conquer scheme does not conflict with the greedy algorithm, as the greedy algorithm may be applied to solve each partition.

[18] This fact does not hold for unordered queries: the choice of any following positive row may discard elements of Π.

Another interesting optimizing technique could involve a static analysis of variables in optional terms. If the interdependency of optional terms based on variables can be excluded, it may be possible to find a more efficient handling of optional terms in partial queries.

There are, of course, further possible optimization techniques that may be investigated. This single example is just one of several possible refinements, others may be found as the optimized evaluation algorithm is reviewed a second time.

5.2 Data Indexing Techniques

Another interesting field of research would be different data indexing techniques for the matrix method and simulation unification as a whole. Since the arity verifications and constraints make it necessary to count the number of children, it would be possible for example to create an index with the namespaces and labels of the children, or to count the number of children with certain namespaces and labels. This could lead to further improvements in the performance of query evaluation, but the amount of saved time and space is still to be investigated.

5.3 Optimizing Path Execution

Path execution means the collecting of constraints and the application of consistency verification rules. The current specification of the algebraic operators as well as the matrix method use disjunctive normal forms for the representation of solutions. Searching for ways to avoid the entire expansion of the formula to its disjunctive normal form may be promising for size and even runtime reduction.

There are several possibilities to avoid the complete expansion of all solutions. For example, when a certain matrix region guarantees that it cannot violate the path constraints, the canonical formula of this matrix region represents the same as the formula under path constraints. It is hence not necessary to expand this matrix by computing all the paths through the matrix.

Another possibility to optimize path execution would be to refine the consistency verifier, called solver in the prototype. The solver operates on pairs of constraints, since the consistency verification rules build conclusions from constraint pairs. The verifier could be refined in such a way that it is able to find constraint pairs within formulas that are not in disjunctive normal form. It is even possible to determine the weight of a constraint pair, that is, the number of disjuncts in which it would be contained. Furthermore, when resolving the pair A, C to a formula A', it would be possible to replace all occurrences of A, C within a formula without having do expand it completely.

$$(A \vee B) \wedge (C \vee D) \equiv A \wedge C \vee (A \wedge C) \vee (A \vee B) \wedge D \equiv A \wedge C \vee A \wedge D \vee B \wedge (C \vee D)$$

This is possible by applying this partial expansion rule recursively until A and C are in one direct conjunction.

Both the partial expansion rule for constraint substitution and the pair weight to determine the substitution order could be used to reduce the memory usage of program evaluation.

5.4 Matrix Method and Data Schema

The matrix method has proven to be a very valuable technique for query evaluation. It is hence worth to investigate whether the matrix method can be also used to verify whether data compliant to a given schema can potentially satisfy a given query term. This could be especially valuable for the optimization of chaining, since certain chaining applications could be excluded statically.

Another interesting approach would be to use schema information to optimize query evaluation. The rewriting of algebraic query expressions could be, for example, led by the given data schema.

5.5 Optimizing Rule Chaining

Rule chaining in the current prototype is very inefficient. Tabling techniques ([War92]) could help to reduce program runtime and even program nontermination, as shown in XSB prolog[19]. It must be investigated however whether the optimization techniques are still applicable when changing the unification method from the unification used in logic programming unification to simulation unification.

A further possible research area would be to introduce a second language to specify optimization hints for chaining. With the help of such a optimization language, it would be possible to experiment with different optimization techniques in various scenarios.

5.6 Abstract Machine based on the Algebra

The query algebra is only a step towards an abstract machine for Xcerpt. There have already been investigations on the Xcerpt abstract machine, but the query algebra for Xcerpt offers a more versatile approach to an abstract machine using logical and physical algebras as implementation specifications. An abstract machine based on the defined algebra could be easily used for code optimization analysis.

6 Conclusion

Xcerpt is a rule based query language for semistructured data. It carries concepts from logic programming over to semistructured data querying, but also introduces new constructs.

The aim of this thesis was to investigate optimization of query evaluation, that is to say, the optimization of simulation unification. The optimizations were investigated from two viewpoints. First, this work considered a high level logical query algebra that leads to more general considerations of query optimization by expression rewriting. Throughout this work, it becomes clear that especially the evaluation of the new constructs `optional` and `without` needed an algorithmic improvement. For this reason, the second part of this thesis investigates algorithmic improvements of query evaluation for `optional` and `without` constructs.

[19] See `http://xsb.sourceforge.net`

The contributions of this thesis to Xcerpt are the following. Section 2.3 investigates the semantics of the `without` and `optional` constructs from the viewpoint of the programmer. The conclusion of this investigation is given in seven rules that ease the understanding and the conception of Xcerpt queries with these special Xcerpt constructs. The section shows before all that the three query term types (positive, optional and negative) follow precedence rules, and that every of the three query term types have different conflict resolution rules.

Section 3 presented a new formalization of the simulation unification with the help of a higher level algebra. The algebra focused on the incorporation of negated and optional queries. Furthermore, the algebra specified reordering rules usable for static program optimization based on program analysis and data meta-information such as histograms and data schemas.

Section 4 finally contributes the complexity analysis of specific parts of the matrix method algorithm, and investigates new algorithms that are conceivable for simulation unification. The presented techniques reduce the number of (arbitrarily expensive) recursive calls to simulation unification on the one hand. On the other hand, they allow to deduce immediate failure of the unification based on analysis of the unification matrix without generating a possibly exponential number of paths. Furthermore, it is investigated how the complexity of path generation can be reduced in the presence of negated or optional query terms. As the complexity analysis of the new techniques and the preliminary experimental results demonstrate, a major runtime reduction can be achieved in several cases.

The algorithmic optimizations presented in this thesis can be used to improve the implementation of the algebraic operators. In fact, the greedy algorithm and the delta-algorithm can be used as an implementation of the query algebra. The details of this redefinitions have not been investigated yet, but they seem to be an interesting refinement of the query algebra.

A further important step towards an efficient evaluation of Xcerpt programs would be the investigation of chaining optimizations. Tabling techniques have proven to be valuable for logic programming, and some of the optimization techniques may be transferred to Xcerpt.

Furthermore, the Xcerpt query algebra could be extended to include data construction and rule chaining operators, and finally, the logical algebra could be refined to define an abstract machine for Xcerpt, especially using the optimized evaluation techniques of the matrix method.

As a whole, this work has shown that simulation unification can be optimized in several ways, and that especially the negation and optionality constructs `without` and `optional` can be evaluated efficiently. This result is very satisfying from the viewpoint of language design. It demonstrates that introducing higher order constructs that are shortcuts for often repeated programming patterns is sensible. On the one hand, they enable the programmer to write shorter and clearer programs. On the other hand, the evaluation of the shortcut constructs can be more efficient than for the initial program pattern it stems from. Program evaluation can use certain evaluation tricks to improve performance that are based on the semantics of the new construct and are not accessible to the programmer. Hence, both the programmer and the program evaluation can profit from the introduction of such constructs. As this thesis has shown, especially the `optional` construct has proven to be a valuable higher order construct.

Appendix

A The Operators of the Xcerpt Query Algebra

Name	**Notation**
Selection	$\sigma(fun\ \theta\ value)$
Select position	$\sigma^{pos}(i)$
Map	$Child(i, f_1, \ldots, f_n)[f_{sub}]$
Descendant	$Desc[f_{sub}]$
Attribute	$Attr(i)[f_{sub}]$
Bind	$Bind(X)$
Bind value	$\beta(X = fun)$
Bind position	$\beta^{pos}(X)$
Negated Child	$Child_{neg}(i, f_1, \ldots, f_n)[f_{sub}]$
Negated Attribute	$Attr_{neg}(i)[f_{sub}]$
Combine negated	$Comb$
Optional Total Child	$Child_{opt,tot}(i, f_1, \ldots, f_n)[f_{sub}]$
Optional Partial Child	$Child_{opt,part}(i, f_1, \ldots, f_n)[f_{sub}]$
Optional Total Attribute	$Attr_{opt,tot}(i)[f_{sub}]$
Optional Partial Attribute	$Attr_{opt,part}(i)[f_{sub}]$
Prune Optional Children	$Prune$
Prune Optional Attributes	$Prune_{Attr}$
Clear	$Clear$

B Translation Scheme: Query Term to Algebra Expression

$$
\begin{array}{lcl}
Tr_{term}(\texttt{'"'}\ \texttt{string}\ \texttt{'"'}) & = & \sigma(type = \text{“}text\text{”}) \\
 & & \sigma(value = string) \\
Tr_{term}(\texttt{"/"}\ \texttt{regexp}\ \texttt{"/"}) & = & \sigma(type = \text{“}text\text{”}) \\
 & & \sigma(value \sim regexp) \\
Tr_{term}(\texttt{number}) & = & \sigma(type = \text{“}text\text{”}) \\
 & & \sigma(value = number) \\
Tr_{term}(\texttt{"var"}\ \texttt{label}) & = & Bind(label) \\
Tr_{term}(\texttt{"var"}\ \texttt{label}\ \texttt{"as"}\ \texttt{<query-subterm>}) & = & Bind(label) \\
 & & Tr_{term}(\texttt{<query-subterm>}) \\
Tr_{term}(\texttt{"desc"}\ \texttt{<query-subterm>}) & = & Desc[\ Tr(\texttt{<query-subterm>})\ ] \\
Tr_{term}(\texttt{"position"}\ \texttt{number}\ \texttt{<query-subterm>}) & = & \sigma^{pos}(number) \\
 & & Tr_{term}(\texttt{<query-subterm>}) \\
Tr_{term}(\texttt{"position"}\ \texttt{"var"}\ \texttt{label}\ \texttt{<query-subterm>}) & = & \beta^{pos}(label) \\
 & & Tr_{term}(\texttt{<query-subterm>})
\end{array}
$$

$Tr_{term}(\texttt{<ns-prefix>}\ \texttt{":"}\ \texttt{<label>}\ \texttt{<attributes>}\ \texttt{<subterms>})$

$$
\begin{array}{lcl}
 & = & \sigma(type = \text{“}tree\text{”}) \\
 & & Tr_{val}(namespace, \texttt{<ns-prefix>}) \\
 & & Tr_{val}(label, \texttt{<label>}) \\
 & & Tr_{atts}(\texttt{<attributes>}) \\
 & & Tr_{terms}(\texttt{<subterms>})
\end{array}
$$

$$
\begin{array}{lcl}
Tr_{val}(f, \texttt{"var"}\ \texttt{label}\ ) & = & \beta(label = f) \\
Tr_{val}(f, \texttt{"/"}\ \texttt{regexp}\ \texttt{"/"}) & = & \sigma(f \sim regexp) \\
Tr_{val}(f, \texttt{'"'}\ \texttt{string}\ \texttt{'"'}) & = & \sigma(f = string) \\
Tr_{val}(f, \texttt{label}) & = & \sigma(f = label)
\end{array}
$$

Let o be the number of subterms in the term list of the form

<query-subterm> = "optional" <query-subterm>

If $o = 0$ then:

Tr_{terms}("{" <query-subterm>$_1$ "," ... "," <query-subterm>$_n$ "}")
= $\sigma(arity = n)$
$Child(1)[\ Tr_{term}$(<query-subterm>$_1$)]
...
$Child(n)[\ Tr_{term}$(<query-subterm>$_n$)]

Tr_{terms}("[" <query-subterm>$_1$ "," ... "," <query-subterm>$_n$ "]")
= $\sigma(arity = n)$
$Child(1, pos = max + 1)[\ Tr_{term}$(<query-subterm>$_1$)]
...
$Child(n, pos = max + 1)[\ Tr_{term}$(<query-subterm>$_n$)]

If $o > 0$ then:

Tr_{terms}("{" <query-subterm>$_1$ "," ... "," <query-subterm>$_n$ "}")
= $\sigma(arity \geq (n - o))$
$\sigma(arity \leq n)$
$Tr_{child,tot}(1)$(<query-subterm>$_1$)
...
$Tr_{child,tot}(n)$(<query-subterm>$_n$)

Tr_{terms}("[" <query-subterm>$_1$ "," ... "," <query-subterm>$_n$ "]")
= $\sigma(arity \geq (n - o))$
$\sigma(arity \leq n)$
$Tr_{child,tot}(1, pos = max + 1)$)(<query-subterm>$_1$)
...
$Tr_{child,tot}(n, pos = max + 1)$(<query-subterm>$_n$)

$Tr_{child,tot}(i, f_1, \ldots, f_n)$("optional"<query-subterm>)
= $Child_{opt,tot}(i, f_1, \ldots, f_n)[\ Tr_{term}$(<query-subterm>)

$Tr_{child,tot}(i, f_1, \ldots, f_n)$(<query-subterm>)
= $Child(i, f_1, \ldots, f_n)[\ Tr_{term}$(<query-subterm>)

Let $positive(i) : Integer \rightarrow Integer$ be a function that returns the number of positive subterms after the subterm at position i. Let $p : Integer$ be the number of positive query subterms in the translated query term list.

Tr_{terms}("{{" <query-subterm>$_1$ "," ... "," <query-subterm>$_n$ "}}")
$=$ $\sigma(arity \geq p)$
$Tr_{child,part}(1)$(<query-subterm>$_1$)
...
$Tr_{child,part}(n)$(<query-subterm>$_n$)
$Comb$

Tr_{terms}("[[" <query-subterm>$_1$ "," ... "," <query-subterm>$_n$ "]]")
$=$ $\sigma(arity \geq p)$
$Tr_{child,part}(1, pos \leq -positive(1))$(<query-subterm>$_1$)
$Tr_{child,part}(2, pos > max, pos \leq -positive(2))$(<query-subterm>$_2$)
...
$Tr_{child,part}(n-1, pos > max, pos \leq -positive(n-1))$(<query-subterm>$_{n-1}$)
$Tr_{child,part}(n, pos > max)$(<query-subterm>$_n$)
$Comb$

$Tr_{child,part}(i, f_1, \ldots, f_n)$("without" <query-subterm>)
$=$ $Child_{neg}(i, f_1, \ldots, f_n)[\ Tr_{term}$(<query-subterm>)

$Tr_{child,part}(i, f_1, \ldots, f_n)$("optional" <query-subterm>)
$=$ $Child_{opt,part}(i, f_1, \ldots, f_n)[\ Tr_{term}$(<query-subterm>)

$Tr_{child,part}(i, f_1, \ldots, f_n)$(<query-subterm>)
$=$ $Child(i, f_1, \ldots, f_n)[\ Tr_{term}$(<query-subterm>)

Let o be the number of attributes in the attribute list of the form

<attribute> = "optional" <attribute>

If $o = 0$ then:

Tr_{atts}("(" <ns-prefix>$_1$ ":" <label>$_1$ "=" <value>$_1$
"," ... ","
<ns-prefix>$_n$ ":" <label>$_n$ "=" <value>$_n$ ")")

$=$ $\sigma(arity_a = n)$
$Attr(1)[$
$Tr_{val}(namespace, \texttt{<ns-prefix>}_1)$
$Tr_{val}(label, \texttt{<label>}_1)$
$Tr_{val}(value, \texttt{<value>}_1)$
$]$
...
$Attr(n)[$
$Tr_{val}(namespace, \texttt{<ns-prefix>}_n)$
$Tr_{val}(label, \texttt{<label>}_n)$
$Tr_{val}(value, \texttt{<value>}_n)$
$]$
$Clear$

If $o > 0$ then:

Tr_{atts}("(" <attribute>$_1$"," ... ","<attribute>$_n$ ")")
$=$ $\sigma(arity_a \geq (n - o))$
$\sigma(arity_a \leq n)$
$Tr_{att,tot}(1)(\texttt{<attribute>}_1)$
...
$Tr_{att,tot}(n)(\texttt{<attribute>}_n)$
$Clear$

$Tr_{att,tot}(i)$("optional" <ns-prefix> ":" <label> "=" <value>)
$=$ $Attr_{opt,tot}(i)[$
$Tr_{val}(namespace, \texttt{<ns-prefix>})$
$Tr_{val}(label, \texttt{<label>})$
$Tr_{val}(value, \texttt{<value>})$
$]$

$Tr_{att,tot}(i)$(<ns-prefix> ":" <label> "=" <value>)
$=$ $Attr(i)[$
$Tr_{val}(namespace, \texttt{<ns-prefix>})$
$Tr_{val}(label, \texttt{<label>})$
$Tr_{val}(value, \texttt{<value>})$
$]$

Let $p : Integer$ be the number of positive attributes in the translated attribute list.

Tr_{atts}("((" <attribute>$_1$"," ... "," <attribute>$_n$ "))")
= $\sigma(arity_a \geq p)$
$Tr_{att,part}(1)$(<attribute>$_1$)
...
$Tr_{att,part}(n)$(<attribute>$_1$)
$Comb$
$Clear$

$Tr_{att,part}(i)$("optional" <ns-prefix> ":" <label> "=" <value>)
= $Attr_{opt,part}(i)$[
$Tr_{val}(namespace,$ <ns-prefix>$)$
$Tr_{val}(label,$ <label>$)$
$Tr_{val}(value,$ <value>$)$
]

$Tr_{att,part}(i)$("without" <ns-prefix> ":" <label> "=" <value>)
= $Attr_{neg}(i)$[
$Tr_{val}(namespace,$ <ns-prefix>$)$
$Tr_{val}(label,$ <label>$)$
$Tr_{val}(value,$ <value>$)$
]

$Tr_{att,part}(i)$(<ns-prefix> ":" <label> "=" <value>)
= $Attr(i)$[
$Tr_{val}(namespace,$ <ns-prefix>$)$
$Tr_{val}(label,$ <label>$)$
$Tr_{val}(value,$ <value>$)$
]

References

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[BBF+05] Oliver Bolzer, François Bry, Tim Furche, Sebastian Kraus, and Sebastian Schaffert. Development of Use Cases, Part I. Deliverable, REWERSE IV Network of Excelence, 2005.

[BBN99] Michel Biezunski, Martin Bryan, and Steven R. Newcomb. ISO/IEC 13250:2000 Topic Maps: Information Technology – Document Description and Markup Languages. International Standard, International Organization for Standardization and International Electrotechnical Commission, 1999.

[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 *VLDB'03: Proceedings of 29th International Conference on Very Large Data Bases*, pages 1053–1056. Morgan Kaufmann, 2003.

[BCF+05] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Candidate Recommendation, World Wide Web Consortium, 2005.

[BFB+04] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Identification of Design Principles. Deliverable, REWERSE IV Network of Excelence, 2004.

[BFB+05] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems (IJSWIS)*, 1(2), 2005.

[BHKM05] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-Fledged Algebraic XPath Processing in Natix. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 705–716. IEEE Computer Society, 2005.

[Bib92] W. Bibel, editor. *Deduktion: Automatisierung der Logik*. Oldenbourg, München, 1992.

[BM05] François Bry and Massimo Marchiori. Ten Theses on Logic Languages for the Semantic Web. In *Proceedings of W3C Workshop on Rule Languages for Interoperability*. World Wide Web Consortium, 2005.

[BMR99] D. Beech, A. Malhotra, and M. Rys. A Formal Data Model and Algebra for XML. Communication to the w3c, unpublished, 1999.

[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*, volume 2876 of *LNCS*, pages 17–34. Springer-Verlag, 2003.

[BSS04] François Bry, Sebastian Schaffert, and Andreas Schröder. A contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Proceedings of 15th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2004, and 18th Workshop on Logic Programming, WLP 2004*, volume 3392 of *LNCS*, pages 258–268. Springer-Verlag, 2004.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Recommendation, World Wide Web Consortium, 1999.

[Cla99] James Clark. XSL Transformations (XSLT) Version 1.0. Recommendation, World Wide Web Consortium, 1999.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[DFF+98] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. XML-QL: A Query Language for XML. Submission to the World Wide Web Consortium, 1998.

[DFF+05] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Candidate Recommendation, World Wide Web Consortium, 2005.

[FHP02] Flavius Frasincar, Geert-Jan Houben, and Cristian Pau. XAL: An Algebra for XML Query Optimization. In *ADC'02: Thirteenth Australasian Database Conference*, volume 5, pages 49–56. ACS, 2002.

[FSW00] Mary Fernandez, Jerome Simeon, and Philip Wadler. An Algebra for XML Query. In *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science: 20th Conference*, volume 1974 of *LNCS*, pages 11–46. Springer-Verlag, 2000.

[JLST01] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Database Programming Languages, 8th International Workshop, DBPL 2001*, volume 2397 of *LNCS*, pages 149–164. Springer-Verlag, 2001.

[Lie99] Hartmut Liefke. Horizontal Query Optimization on Ordered Semistructured Data. In *WebDB (Informal Proceedings)*, pages 61–66, 1999.

[MB04] Brian McBride and Dave Beckett. RDF/XML Syntax Specification (Revised). Recommendation, World Wide Web Consortium, 2004.

[MBG04] Brian McBride, Dan Brickley, and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation, World Wide Web Consortium, 2004.

[MH04] Brian McBride and Patrick Hayes. RDF Semantics. Recommendation, World Wide Web Consortium, 2004.

[MKC04] Brian McBride, Graham Klyne, and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, World Wide Web Consortium, 2004.

[Sch04] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, Institute for Computer Science, University of Munich, 2004.

[War92] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[ZPR02] Xin Zhang, Bradford Pielech, and Elke A. Rundesnteiner. Honey, I shrunk the XQuery!: an XML algebra optimization approach. In *WIDM '02: Proceedings of the 4th international workshop on Web information and data management*, pages 15–22, New York, NY, USA, 2002. ACM Press.