Xcerpt: A Rule-Based Query and Transformation Language for the Web

Dissertation

zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

von
Sebastian Schaffert

Oktober 2004

Erstgutachter:	Prof. Dr. François Bry (Universität München)
Zweitgutachter:	Prof. Dr. Georg Gottlob (TU Wien)
Tag der mündlichen Prüfung:	13. Dezember 2004

Abstract

This thesis investigates querying the Web and the Semantic Web. It proposes a new rule-based query language called *Xcerpt*. Xcerpt differs from other query languages in that it uses patterns instead of paths for the selection of data, and in that it supports both rule chaining and recursion. Rule chaining serves for structuring large queries, as well as for designing complex query programs (e.g. involving queries to the Semantic Web), and for modelling inference rules. Query patterns may contain special constructs like partial subqueries, optional subqueries, or negated subqueries that account for the particularly flexible structure of data on the Web.

Furthermore, this thesis introduces the syntax of the language Xcerpt, which is illustrated on a large collection of use cases both from the conventional Web and the Semantic Web. In addition, a declarative semantics in form of a Tarski-style model theory is described, and an algorithm is proposed that performs a backward chaining evaluation of Xcerpt programs. This algorithm has also been implemented (partly) in a prototypical runtime system. A salient aspect of this algorithm is the specification of a non-standard unification algorithm called *simulation unification* that supports the new query constructs described above. This unification is symmetric in the sense that variables in both terms can be bound. On the other hand it is in contrast to standard unification assymmetric in the sense that the unification determines that the one term is a subterm of the other term.

Zusammenfassung

Diese Arbeit untersucht das Anfragen des Webs und des Semantischen Webs. Sie stellt eine neue regel-basierte Anfragesprache namens Xcerpt vor. Xcerpt unterscheidet sich von anderen Anfragesprachen insofern, als dass es zur Selektion von Daten sog. Pattern („Muster") verwendet und sowohl Regelschliessen als auch Rekursion unterstützt, was sowohl zur Strukturierung größerer Anfragen als auch zur Erstellung komplexer Anfrageprogramme, und zur Modellierung von Inferenzregeln dient. Anfrage-Pattern können spezielle Konstrukte, wie partielle Teilanfragen, optionale Teilanfragen, oder negierte Teilanfragen, enthalten, die der besonders flexiblen Struktur von Daten im Web genügen.

In dieser Arbeit wird weiterhin die Syntax von Xcerpt eingeführt, und mit Hilfe mehrerer Anwendungsszenarien sowohl aus dem konventionellen als auch aus dem semantischen Web erläutert. Ausserdem wird eine deklarative Semantik im Stil von Tarski's Modelltheorie beschrieben und ein Algorithmus vorgeschlagen, der eine rückwärtsschliessende Auswertung von Xcerpt durchführt und in einem prototypischen Laufzeitsystem implementiert wurde. Wesentlicher Bestandteil des Rückwärtsschliessens ist die Spezifikation eines nicht-standard Unifikations-Algorithmus, der die oben genannten speziellen Xcerpt-Konstrukte berücksichtigt. Diese Unifikation ist symmetrisch in dem Sinne, dass sie Variablen in beiden angeglichenen („unifizierten") Termen binden kann. Andererseits ist sie im Gegensatz zur Standardunifikation assymmetrisch in dem Sinne, dass der dadurch geleistete Angleich den einen Term als „Teilterm" des anderen erkennt.

"At times our own light goes out and is rekindled by a spark from another person. Each of us has cause to think with deep gratitude of those who have lighted the flame within us."

— Albert Schweitzer

Acknowledgements

The language Xcerpt presented in this thesis would not exist today without the continuous support and contribution of many fellow researchers and students at the University of Munich, at the University of Linköping (Sweden), and elsewhere. Of those, particular gratitude goes to the following colleagues:

- *François Bry*, University of Munich, with whom I had many – sometimes heated but always fruitful – discussions on almost all issues concerning Xcerpt.
- *Włodzimierz Drabent*, Polish Academy of Sciences, Warszawa, for discussing and proof-reading, and hinting to some important mistakes.
- *Norbert Eisinger*, University of Munich, for spending much time with me discussing syntax and semantics of the language, and for being (almost) always there when I needed him.
- *Georg Gottlob*, Technical University of Vienna, for being willing to take the burden of being second supervisor of this thesis
- *Jan Małuszyński*, University of Linköping, for giving me the chance to present my work and for many interesting discussions and ideas.

Also, I thank Tim Furche (University of Munich), Paula-Lavinia Pătrânjan (University of Munich), and Artur Wilk (University of Linköping) for working with me on several aspects of Xcerpt.

Furthermore, several graduate students, who worked on diploma and project theses during the development of Xcerpt, deserve to be mentioned separately:

- *Sacha Berger*, who is now a fellow researcher, developed in his diploma thesis the visual language *visXcerpt* which builds upon Xcerpt and has received a lot of attention in the research community. A short introduction into this work can be found near the end of this thesis.
- *Oliver Bolzer* currently investigates Semantic Web querying with Xcerpt as part of his diploma thesis and contributed many improvements to the source code of the prototypical runtime system presented in this thesis.
- *Sebastian Kraus* worked extensively with the language Xcerpt during his diploma thesis, in which he developed a comprehensive set of use cases for Xcerpt, some of which also appear in this thesis. His work had much influence on the development of appropriate language constructs for Xcerpt.
- *Andreas Schroeder* developed in his project thesis several different approaches to backward chaining in Xcerpt. Discussions and work with him ultimately lead to the operational semantics as it is presented in this thesis.

. . . and last but not least: most gratitude goes to my family for supporting me patiently during the course of this thesis.

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

Traunstein and Munich, October 2004

CONTENTS

Part I

Introduction and Motivation

CHAPTER

ONE

Introduction

1.1 Motivation

Data on the Web

The advent of the Internet, and in particular of the World Wide Web (in the rest of this thesis usually referred to as "the Web"), has resulted in the availability of huge amounts of data that are accessible by anyone. However, such data – being represented in documents written in the language HTML (*Hypertext Markup Language*) – is mostly aimed at presentation in a user agent (the *browser*) and not meant for further automatic treatment, like information extraction, combination or other further calculation. For example, a Web shop might represent a list of products in an HTML table that contains one column for prices, but the user agent is merely capable of displaying this table and e.g. cannot automatically add the value added tax to prices, since it has no means to differentiate between the price and e.g. the article number. Such documents are thus mostly *layout oriented*.

The *World Wide Web Consortium* (W3C)[1] recognised this deficiency and in 1996 initiated the development of the *Extensible Markup Language* (XML). XML allows authors to define custom, application-specific markup languages (cf. Section 2.2) that may be used for structuring documents according to their *content* rather than according to their *layout*. For example, the Web shop mentioned above might represent its article list using custom markup that clearly distinguishes between prices, article numbers and article names. Layout is added to such documents by using external *stylesheets* that the browser can use to arrange content properly.

XML is currently not only used for representing documents in the traditional sense (i.e. documents containing mostly text), but also as a means for exchanging and storing arbitrary data, such as data stored in a relational or object oriented database. In particular, it is nowadays the data exchange format of choice in application areas such as electronic commerce, molecular biology and astronomy, and is used as the basis for many Web applications. Interestingly, however, XML is much less restrictive than traditional database formats regarding the structure and schema of the data. Therefore, XML data is often also called *semistructured* (cf. Chapter 2).

In order to retrieve information from structured documents, the Web needs query languages (cf. [73]). A Web query language needs to consider properties that are peculiar to the representation and querying of data on the Web. In particular, it needs to be able to deal with partial information, multiple sources that are not managed by a central administration, and changing data structures, and it has to be simple enough to be used by a wide range of users that are not experts in programming, but want to formulate simple queries. Querying and transformation of XML data has received much attention, and the W3C proposals *XQuery* [113] and *XSLT* (both described in Section 3.3) have become de facto standards for this purpose, although they are often criticised – among other things – for their complexity.

[1] `http://www.w3.org`

The Semantic Web

In addition to the development of XML, a major endeavour in Web research is the so-called *Semantic Web*, a term coined by W3C founder Tim Berners-Lee in a *Scientific American* article[19] describing the future of the Web. The Semantic Web aims at enriching data (that is e.g. represented in XML) by meta-data and (meta-)data processing that allows Web based systems to take advantage of "intelligent" reasoning capabilities.

Such meta-data is currently mainly represented in *ontologies* (e.g. using the languages *OWL* and *RDF* [118, 119], or *Topic Maps* [85]) that describe hierarchies of concepts, and relations between them. For example, a Semantic Web application for a book store could assign categories to books as shown in Figure 1.1. A customer interested in *history* and *fiction* might also get offers for books that are in the subcategories *classic*, *mediæval* and *modern* (like the book *Folket i Birka*[2] in the Figure), although these books are not directly contained in the category *history*, because the data processing system has access to the ontology and can thus infer the fact that a book about mediæval times is a historic book.

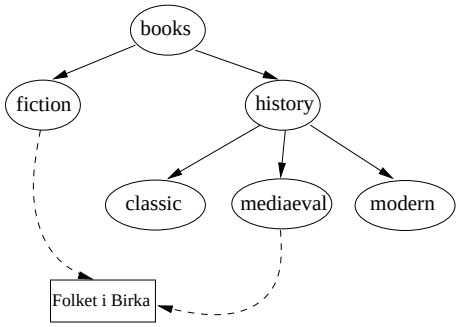

Figure 1.1: A categorisation of books as it might occur in a Semantic Web ontology

Obviously, Semantic Web processing also needs query languages, but although most Semantic Web formalisms are based on XML (e.g. *OWL* or *RDF* [118, 119]), current XML query languages like *XQuery* and *XSLT* are not well suited for the task, as they lack reasoning capabilities. Instead, there are several proposals for specific Semantic Web query and reasoning languages (e.g. *OWL-QL* or *RQL*). However, all current proposals are special purpose languages that only implement a specific form of reasoning, e.g. that of a certain description logic like $\mathcal{SHIQ}$ [58, 59], and are only capable of querying data in specific formats like *OWL* or *RDF*.

Xcerpt

The language Xcerpt introduced in this thesis is a declarative, rule-based query language for Web data (in particular XML) that is based on concepts from logic programming [71, 103]. It differs from conventional XML query languages in several aspects (cf. *Design Principles* below). Xcerpt aims at being simple to use for a wide range of users while being powerful enough to build complex query programs. For this reason, it is developed based on many practical applications, while at the same time providing a solid formal semantics that allows the implementation of different runtime systems.

In contrast to conventional XML query languages, Xcerpt provides means to reason with Semantic Web data similar to those of other rule-based or logic programming languages (e.g. Prolog). In contrast to special purpose Semantic Web query languages, Xcerpt is capable of querying any kind of Web data (combining meta-data with data), and has been conceived to allow to implement a wide range of different reasoning mechanisms as needed.

The goal of this thesis is to introduce the language Xcerpt as a query language for the conventional and the Semantic Web, and provide a formal semantics that it is suitable for the implementation of a runtime system. A major contribution of this work is the specification of a non-standard unification algorithm called *simulation unification* that is well-suited for querying Web data, because it allows to use less rigid structures and supports incomplete query specifications. This unification is symmetric in the sense that is capable of binding variables occurring in both of the unified structures; on the other hand it is – in contrast to the standard unification specified by Robinson [91] – asymmetric in that it tries to find the one structure as a substructure of the other instead of trying to make them equal.

As part of this thesis, a prototypical runtime system has been developed that partly implements Xcerpt based on the formal semantics presented here.

[2] English translation (from Swedish): "The people of Birka"

1.2 Outline of this Thesis

This thesis is structured into three parts consisting of ten chapters, and an appendix consisting of four chapters:

- The first part (called *Introduction and Motivation*) introduces into the area of Web query languages. Chapter 1 is this introduction. Chapter 2 describes data representation formats for the Web, in particular XML and semistructured data. Chapter 3 concludes the first part and describes requirements and proposals for Web query languages.

- The second part (called *The Language Xcerpt*) gives a detailed description of the language Xcerpt. Chapter 4 begins with the syntax and an informal description of the semantics. Chapter 5 provides an extensive list of different use cases for Xcerpt. Chapter 6 describes syntactic restrictions imposed on Xcerpt programs. In the formal semantics, Xcerpt programs are assumed to conform to these restrictions. Chapter 7 proposes a declarative semantics for Xcerpt in the form of a Tarski style model theory, and Chapter 8 provides a complementary backward chaining operational semantics. In particular, the operational semantics describes the simulation unification algorithm.

- The third part (called *Conclusion*) describes perspectives for further work on Xcerpt (in Chapter 9) and concludes this thesis (in Chapter 10).

- The appendix contains supplemental material. Most importantly, it contains a description of the prototypical runtime system implemented as part of this thesis (Appendix A) and some of the more extensive proofs of theorems in Chapters 7 and 8 (Appendix B). Also, this part contains an index, a list of examples, the bibliography, and a resume of the author.

1.3 Design Principles of Xcerpt

The following major design principles have guided the design of the language Xcerpt, and to some extent also differentiate Xcerpt from other query languages that have been proposed for the conventional Web, the Semantic Web, and for querying databases.

1.3.1 Referential Transparency and Answer Closedness

Referential Transparency

Referential transparency means that all occurrences of an expression have the same meaning (within a certain scope of definition). This is an important property of declarative languages, as it eases understanding of programs and thus allows for an easier development, maintenance, and optimisation. Referential transparency is usually found in purely functional languages and in logic programming languages (like Haskell, SML, or Prolog), but not in imperative languages (like Java or C), whose notion of state inherently conflicts with referential transparency. In particular, the XML query languages XQuery and XSLT are not fully referentially transparent due to their notion of *context nodes*.

Answer Closedness

The property *answer closedness* expresses that every answer can itself be used as a query. In a query language, this means in particular that any subquery can (syntactically) be replaced by an answer to this subquery, yielding a new, valid query. Usually, logic programming languages are answer closed (e.g., occurrences of variables can be replaced by their bindings), but languages using a different syntax for querying than for the data are not (e.g. SQL or XQuery). Answer closedness is desirable, as it eases understanding of programs for developers by ensuring similar syntaxes for both data and query.

1.3.2 Answers as Arbitrary XML Data

XML is the *lingua franca* of data interchange on the Web. As a consequence, answers should be expressible as every possible XML application. This includes both text without markup and text with freely chosen markup and structure. This requirement is obvious and widely accepted for conventional Web query languages. Semantic Web query languages, too, should be capable of delivering answers in every possible XML application so as to make it possible, e.g. to mediate between RDF and XTM data or to translate RDF data from one RDF syntax into another RDF format.

1.3.3 Pattern-Based Queries

The Navigational Approach

XML documents describe tree or rooted graph structures. Current XML query languages like XQuery or XSLT [113, 106] use a *navigational* or *path-based* approach to select data items in such tree structures, i.e. a selection is specified in terms of a path expression (usually expressed in the language XPath [108]) consisting of a sequence of location steps that specify how to reach the node that contains the desired data in a stepwise manner[3]. For instance, consider a (well-formed) XML document containing the data of an address book. Such a document could look as follows:

```
<address-book>
  <person>
    <name>
      <first>Mickey</first>
      <last>Mouse</last>
    </name>
    <phone>19281118</phone>
    <email>mickey@mouse.org</email>
  </person>
  <person>
    <name>
      <first>Donald</first>
      <last>Duck</last>
    </name>
    <email>donald@duck.com</email>
  </person>
</address-book>
```

Constructs like `<address-book>` are so-called *opening tags* of an element, and constructs like `</address-book>` are so-called *closing tags* of an element. An *element* is the part of the document between, and including, an opening tag and a fitting closing tag (e.g. `<email>mickey@mouse.org</email>` is an element). The *element label* is the label used in the opening and closing tags of the element definition. Everything (i.e. both text and other elements) enclosed by the opening and closing tag of an element is called the *element content*, if this again includes elements, those are called *child elements* of the element. Child elements of the same *parent* are called *siblings*. In the example above, the element with label `address-book` contains two child elements with label `person`, each of which contains child elements with labels `name` and `email`. As with every well-formed term structure, it is easy to see that the nesting of such elements describes a tree structure.

For retrieving the phone number of the person with first name “Mickey” and last name “Mouse” using a navigational query language, one has to construct a path expression that navigates this tree structure by first looking at the element with label `address-book`, then moving to each child element labelled `person` in turn and from there into the `name`, `first` and `last` elements to ensure that the person is in fact “Mickey Mouse”, then back again to select the content of the sibling element labelled `phone`. In XPath, this selection can thus be expressed as follows:

[3] XQuery, XSLT and XPath are introduced in much more detail in Chapter 3

```
/child::address-book/child::person/child::name[child::first="Mickey" and
                                    child::last="Mouse"]/following-sibling::phone
```

Navigation paths enclosed in [ ] are so-called *qualifiers* that are used as conditions for selecting elements, i.e. in the example above, they are used to select only such `name` elements that have a child with label `first` and content "Mickey" and a child with label `last` and content "Mouse".

Arguably, such a path navigation is often straightforward for simple queries, but can be awkward for everything that goes beyond that, as longer queries become difficult to comprehend and the structure of the queried data is lost. Furthermore, the navigational approach only allows to select one data item at a time. For example, it is not possible to select both the phone number and the email address in a single query. Instead, several queries have to be performed and the results composed afterwards.

Another problem is that it is possible to use many different path expressions to select the same data items. The path expression used above retrieves the same data as the following XPath expression (which differs in that it selects the phone number as child of the `person` element instead of as sibling of the `name` element):

```
/child::address-book/child::person[child::name/child::first="Mickey" and
                                    child::name/child::last="Mouse"]/child::phone
```

Note that if a person entry contained two `name` children instead of only one, the two XPath expressions would in fact differ, because the second expression could select the `first` element of the first `name` element, and the `last` element of the second `name` element, whereas the first XPath expression requires both to be children of the same `name` element.

In addition, XPath also supports backward navigation steps that allow to specify navigations that move upwards in the tree. A third XPath expression using backward navigation steps (`parent::*`) that retrieves the same data as the previous two is for example:

```
/child::address-book/child::person/child::name[child::first="Mickey" and
                                      child::last="Mouse"]/parent::*/child::phone
```

Note that backward navigation steps require to also keep in memory such nodes that have already been visited and thus can be problematic with respect to efficiency. Moreover, patterns with backward steps are often difficult for a programmer to understand and/or correctly specify.

Obviously, this multitude of different path expressions for the same query also contributes to queries being difficult to comprehend.

The Positional Approach

In contrast, in a *positional* or *pattern-based* approach a query pattern is like a *form* that gives an *example* of the data that is to be selected, like the forms of the language QBE [127] or query atoms in logic programming. So as to retrieve data items, a query pattern can furthermore be augmented by zero or more variables. In a pseudo XML notation, the query for the phone number of "Mickey Mouse" could be expressed in a pattern as follows (variables are, as in XQuery, indicated by a leading $):

```
<address-book>
  <person>
    <name>
      <first>Mickey</first>
      <last>Mouse</last>
    </name>
    <phone>$PHONE</phone>
  </person>
</address-book>
```

Note how the query pattern very closely resembles the queried data. As a query pattern may contain more than one variable, also selecting the email address is a trivial task:

```
<address-book>
  <person>
    <name>
      <first>Mickey</first>
      <last>Mouse</last>
    </name>
    <phone>$PHONE</phone>
    <email>$EMAIL</email>
  </person>
</address-book>
```

Arguably, such a pattern-based approach to querying has several advantages compared to the path-based approach discussed above:

1. Query patterns resemble the data very closely and are thus easy to grasp.
2. Query patterns provide only a limited set of possibilities to select the same data items.
3. The higher level of abstraction of query patterns leaves more room for automatic optimisations.
4. In query patterns, several data items can be selected in a single query (using one variable for each item).

Interestingly, a navigational language that only allows forward axes is very similar to a query pattern (with only a single variable). The article [81] shows that any XPath expression containing backward navigation steps can be transformed into an expression containing only forward navigation steps. Thus, query patterns are at least as expressive as path selections.

One of the goals of this thesis is to show that a pattern-based approach to querying Web data is feasible and may result in simpler, more declarative queries. Pattern-based queries for querying Web data have first been proposed in the languages UnQL [31] and XML-QL [45], but those languages have never gained much acceptance. Xcerpt, presented in this thesis, builds upon these approaches and improves them in many aspects.

Comparison with Relational Database Query Languages

Note that query languages for relational database systems usually also use a path-based approach (based on the *tuple calculus* [41, 42]). Suppose the address book used above is given as a table (or *relation*) in a relational database as follows:

addressbook	*first*	*last*	*phone*	*email*
	Mickey	Mouse	19281118	mickey@mouse.org
	Donald	Duck	NULL	donald@duck.com

An SQL expression selecting the phone number of Mickey Mouse would look as follows (in a verbose notation that adds the relation name to all selections, so as to resemble the XML example above more closely):

```
SELECT addressbook.phone
FROM addressbook
WHERE addressbook.first='Mickey' and addressbook.last='Mouse'
```

In relational database systems, the path-based approach (expressed in the *tuple calculus*) is however very similar to the pattern-based approach (expressed in the *domain calculus*), because the flat, relational tuples (in the first normal form) do not leave much room for navigation steps. A transformation from path-based into pattern-based queries and vice versa is possible and has been shown by *Edgar Codd* [41, 42]. In Datalog (cf. for example [101]), which is very close to the domain calculus, the query above is expressed as follows using atomic formulas (from first order logic) as query patterns:

```
phone(P) :- addressbook('Mickey','Mouse',P,_) .
```

If one goes beyond such simple structures, like XML documents or non-first normal form tuples, the difference becomes increasingly apparent (like in the XML examples above).

In relational databases, a language that employs a positional approach is the language *QBE* [127] (*query by example*), which is the foundation of the easy-to-use database user interface MS Access. In QBE, a query for the phone number of Mickey Mouse can be specified by giving a query pattern of the following form:

addressbook	*first*	*last*	*phone*	*email*
	Mickey	Mouse		P._email

The email address is bound to the variable `_email` and printed by the command `P.`. Note the similarity of this QBE query pattern and of the first order logic or Datalog formula.

Comparison with the Object Query Language OQL

Object oriented database systems are capable of representing tree and graph shaped data very similar to the trees and graphs represented by XML (but requiring a much more rigid structure). The most prominent language for object oriented database systems is *OQL* [5] (the *Object Query Language*). Being influenced by SQL, OQL also uses path-based selections of values in the data tree. In OQL, the query for the phone number of the person with first name "Mickey" and last name "Mouse" is specified as follows (note the close resemblance with the XPath selection):

```
SELECT p.phone
FROM AddressBook a, a.person p
WHERE p.name.first = 'Mickey' AND p.name.last = 'Mouse'
```

1.3.4 Incomplete Specification of Query Patterns

Although query patterns resemble terms or atoms in logic programming, they have to take into account properties that are peculiar to Web data and queries to the Web[4]. The most significant difference is that Web data (as represented in HTML or its generalisation XML) has a much more flexible schema compared to data in logic programming or relational databases, even to the extent that much of the schema might be unknown or irrelevant to a query author.

Consider for example an address list published on a Web page. Although this data might conform to a certain schema (like HTML), the actual structure of the document is still largely unknown, because schemas for data on the Web allow much flexibility (like arbitrary repetition of substructures, optional substructures or alternative substructures). For instance, the address list might contain presentational markup and an introductory text, but a query for the phone number of "Mickey Mouse" should be equally valid if it does not, since these parts are irrelevant to the query.

The "rigid" query patterns of logic programming are not feasible for such queries: authors of a query would need to consider the complete and exact structure of the document and provide at least wildcards for data that is irrelevant to the query, i.e. they still have to *care* about something *irrelevant* or *unknown*.

For a pattern-based Web query language, it is thus desirable to be able to specify *incomplete query patterns*. Incompleteness has several facets:

1. Incompleteness in *breadth*[5] allows to omit wildcards for neighbouring nodes in the data tree. E.g. in a query for the phone number in an address book, it is not necessary to provide wildcards for all email addresses that might also be part of the address book entry.

2. Incompleteness in *depth* allows to select data items that are located at arbitrary, unknown depth and skip all structure in between. E.g. when querying address entries in a Web page that are located in a table somewhere in an HTML element, it is possible to just skip all intermediate structure between the root node of the data tree and the table containing the entries.

[4] Chapter 2 discusses in more detail how data is represented on the Web

[5] *Breadth* and *depth* refer to the tree or rooted graph induced by an XML document. This graph is further discussed in Section 2.5.

3. Incompleteness with respect to *order* allows to specify neighbouring nodes in a different order than the one in that they occur in the data tree. E.g. when selecting the phone number and email of an address book entry, it is irrelevant whether the email address occurs before or after the phone number.

4. Incompleteness with respect to *optional elements* allows to query for certain substructures if they exist, but still let the query succeed if they do not exist. In the address book, this may be used to select the phone number if it exists, but still retrieve the name, regardless whether or not it does.

A further goal of this thesis is to discuss extensions to the (elementary) query patterns of logic programming, i.e. the atomic formulas of first order logics, that are suitable to querying conventional and Semantic Web data featuring various facets of incompleteness.

1.3.5 Rules

Deduction Rules

The language Xcerpt described in this thesis uses *deduction rules* (or short *"rules"*) to structure query programs. A deduction rule is simply an *if* ... *then* ... statement. If the condition (a query pattern) is satisfied, then the consequence (a result or construction pattern) is assumed to hold. Usually, the construction pattern uses data selected in the query pattern. Thus, a deduction rule is similar to a `VIEW` in relational database systems.

For instance, a rule may be used to express that *if* the query for the email address of "Mickey Mouse" succeeds *then* there exists a row in an HTML table listing this address:

if

```
<address-book>
  <person>
    <name>
      <first>Mickey</first>
      <last>Mouse</last>
    </name>
    <email>$EMAIL</email>
  </person>
</address-book>
```

then

```
<tr>
  <td>Mickey Mouse</td>
  <td>$EMAIL</td>
</tr>
```

The condition part of a rule (introduced by *if*) is often referred to in the following as the *body* or *query part* of a rule, whereas the consequence (introduced by *then*) is often referred to as the *head* or *construct part* of a rule.

Arguably, rules are declarative, rather easy to understand and provide a high level of abstraction. They are thus well suited for both, novice users who only casually write queries, and experienced programmers who want to write complex query programs.

Rule Chaining

Since a rule defines itself new data of the same kind as the queried data (similar to `VIEWS` in relational database systems), it is also possible to query this new data by other rules. This process is often referred to as *rule chaining*. Rule chaining can serve to conceptually structure larger programs into parts that are easier to manage. Such structuring is advantageous for both, the human user (as programs can be grasped and maintained more easily), and the machine (as code can possibly be reused and does not need to be processed more than once during parsing). In a sense, rule chaining can thus be compared to function or subroutine calls in programming languages.

Furthermore, rule chaining is necessary to perform *deduction*, i.e. to specify how non-existing data is derived from existing data. Deduction rules thus allow to give intensional semantics to data, very much like rules in logic programming, and might be well-suited for Semantic Web applications.

Recursion

Querying Web data often requires more expressive power than is usually found in query languages, e.g. for complex restructuring of tree or graph structured data [4], or to define infinite data sets [101]. To provide this expressive power, it is sometimes desirable to allow *rule recursion*, i.e. a rule may query not only the results of other rule applications but also the results of a previous application of itself. An interesting application for recursive rule chaining is e.g. a Web crawler that recursively follows the hyperlinks found in an HTML document.

This thesis aims at demonstrating this on a wide range of example applications (Chapter 5 is dedicated to different use-cases and applications of Xcerpt).

1.3.6 Forward and Backward Chaining

Forward and Backward Chaining

Chaining in rule-based languages can be evaluated using two different approaches:

- **Forward Chaining.** Forward Chaining is a *data driven* approach. Starting with the initial database, rules are evaluated iteratively against the data until saturation is achieved ("fixpoint"), i.e. no rule application yields data items that have not already been deduced. Forward Chaining is useful for instance for materialising views and for view maintenance, but can be problematic if the fixpoint is infinite, i.e. the iteration never terminates. Also, as forward chaining is not goal oriented, most of the derived data is usually irrelevant to the query.
- **Backward Chaining.** Backward Chaining is a *query driven* approach. Beginning with a dedicated query called the *goal*, program rules and data items are recursively selected if they are relevant for "proving" that a query succeeds. The query is then replaced by the query part (possibly consisting of a conjunction or disjunction of smaller queries) of the selected rule, and the process is repeated until all queries can be evaluated against data items in the database ("facts"). Backward Chaining is useful when the expected result is small in comparison with the number of possible results of the program. Thus, backward chaining is goal-oriented. On the other hand, naïve backward chaining may not terminate even in cases where the fixpoint is finite and forward chaining is guaranteed to terminate.

Rule-based query languages for traditional database systems, like *Datalog* [101], often implement forward chaining, because forward chaining can be evaluated more efficiently and its implementation is usually straightforward. If recursion is not allowed (e.g. when materialising views), or if there are no so-called *dependency cycles* in the data considered, forward chaining is also unproblematic since it always terminates. On the other hand, logic programming languages usually implement backward chaining, as they are working with more complex data structures and thus allow recursion, and do not require dependency cycle freeness of the data.

On the Web, both a forward and a backward chaining approach appear to be desirable. Forward chaining is useful to materialise query results, e.g. for creating static Web pages from an XML document that specifies the content and a query program that adds styling information. Backward chaining is useful if the queried data is not a local resource but instead the Web itself, which is – although finite – very large and difficult to grasp as a whole, as pages might for instance not be available at the moment. Although possible in theory, considering the whole Web as a starting point for a forward chaining evaluation is not viable in practical applications, because the size of the Web exceeds the limit for reasonable response times.

This thesis mainly investigates backward chaining for evaluating query programs. This decision has several interesting consequences, most notably the introduction of a new, non-standard unification algorithm and a runtime engine based on constraint solving.

Non-Standard Unification

When using backward chaining, simple pattern matching of incomplete patterns with data items of a database is not sufficient: queries need to be evaluated against rule heads that might also contain variables, and variables thus cannot always be bound only to non-ground values. It is furthermore usually

```
<ul>
  { FOR $person in document("addressbook.xml")//person,
        $first  in $person/name/first/text(),
        $last   in $person/name/last/text()
    RETURN
      <li> { $first } { $last } </li>
  }
</ul>
```

Figure 1.2: XQuery uses subqueries for grouping: This query constructs an HTML list from an address book.

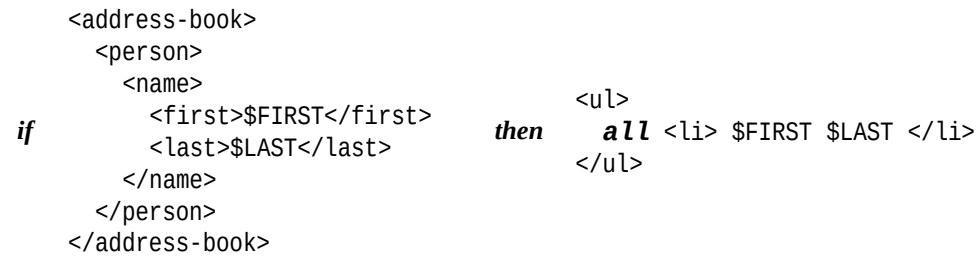

Figure 1.3: Grouping with *all* in a pattern and rule-based query language with strict separation of querying and construction

desirable to also bind (or *restrict*) the variables in the rule head so as to improve performance by restricting the set of possible answers early in the chaining process.

Therefore, logic programming languages that rely on backward chaining, like *Prolog*, use a method called *unification* instead of pattern matching, first introduced by Robinson in 1965 [91]. A unification yields variable bindings whose application makes the query and the rule head syntactically equal except for variable renaming. However, this *standard unification* does not take into account incomplete query patterns like those that are used in Xcerpt. An important part of this thesis is therefore to introduce a new, non-standard unification called *simulation unification* that respects incomplete query patterns.

1.3.7 Separation of Querying and Construction

A further salient property of the language Xcerpt, and one that distinguishes it from almost all other Web query languages, is that it strictly separates querying from construction. Such a separation is desirable as the data items that occur in a query pattern are those of the queried data, while the data items occurring in a construction pattern are those of the resulting data. In XML terms, this means that in general, the two belong to different schemas. Mixing them makes programs more difficult to conceive and thus can result in avoidable programming errors.

A fundamental aspect of this strict separation is that – unlike in other query languages – it is not possible to use embedded subqueries in the style of XQuery or SQL in Xcerpt. While rule chaining can be a substitute for many subqueries, it is not possible to use it for *grouping* several data items within a single Web page. Grouping is e.g. necessary to arrange all entries of an address book in an HTML list. Whereas other query languages like XQuery use a subquery and an iteration construct for this purpose (Figure 1.2), Xcerpt introduces a special construct called *all* that declaratively specifies that data items selected in a query pattern have to be grouped (Figure 1.3) in the separated construction pattern. Like in SQL, the grouping construct *all* can be used together with aggregation functions to compute aggregated values; unlike SQL, it can also be used without aggregation functions, in which case it serves to construct a list of subterms. Thus, *all* takes into account the nested term structure of XML documents.

1.3.8 Reasoning Capabilities

As already mentioned in the *Motivation*, the Semantic Web aims at adding meta-data to resources on the Web that allow to *reason* with data and thus provide a certain amount of "intelligence" to data processing on the Web. Current proposals for the specification of meta-data are the *Resource Description Framework* (RDF) [119], *Topic Maps* [85], and the *Ontology Web Language* (OWL) [118], which are all based on XML as a format for data representation. Nonetheless, most XML query languages are not well suited for querying such data, wherefore several new query and reasoning languages for Semantic Web data have been proposed (e.g. *RQL* [64] and *OWL-QL* [48]). However, all current proposals are special purpose languages that only allow to implement specific forms of reasoning, e.g. those of a certain description logic like $\mathcal{SHIQ}$ [58, 59], and are only capable of querying certain kinds of data, e.g. expressed in the *Resource Description Framework* RDF [119], or the *Web Ontology Language* OWL [118].

Such restrictions, while acceptable for research purposes, are not desirable in practise, as they artificially separate querying and working with data from querying and working with meta-data. However, XML query languages should be able to profit from semantic information in all possible formats and likewise, queries to Semantic Web data should be able to also query XML content.

The last principle is therefore to support the implementation of a wide range of different reasoning algorithms for the Web (without committing to a single formalism), while at the same time being capable of querying any kind of Web data. The rule-based approach used by Xcerpt is promising for this task, as the rules are very similar to the inference or deduction rules in logic programming, and rule chaining with recursion allows to build complex reasoners. At the same time, Xcerpt is developed as a query language for Web data and thus provides capabilities to easily retrieve both data and meta-data. In addition to an implementation in Xcerpt, certain constructs that are frequently needed for querying the Semantic Web might be built natively into the language Xcerpt for efficiency reasons (cf. Section 9.2).

The implementation of Semantic Web reasoning algorithms in Xcerpt is not covered extensively in this thesis, but some small examples of Semantic Web applications are provided. Applying Xcerpt to Semantic Web reasoning is, however, currently being investigated in several related projects.

CHAPTER

TWO

Data Representation on the Web

The Internet and the Web have changed the way of how information is authored and represented in many ways. While the only possibility to author text and make it available to a wider audience used to be to submit it to a journal or to publish a book, which usually meant a rather thorough reviewing process and possibly involved considerable expenses, the Web allows anyone to author, access and publish content very easily, which results in a huge amount of documents with usually differing structures. Likewise, whereas data mostly used to be stored in large, central databases with relatively homogeneous structure and restricted access, information on the Web is decentralised, heterogeneous and often allows access by anyone[1]. While there are also many social issues associated with this change, the main focus of this thesis is on the new way how data is represented on the Web.

Two initially independent developments contribute to this representation: the *extensible markup language* (XML), which has its roots in the document representation community, and the concept of *semistructured data* (SSD), which has been developed to represent heterogeneous data that is not well-suited for traditional database systems. Sections 2.1 and 2.2 give introductions into semistructured data and XML. They follow similar descriptions in [27] and [4]. XML and semistructured data have many concepts in common, and consequently, Section 2.3 tries to bridge the gap between the two. Three larger example scenarios for XML and semistructured data are given in Section 2.4, which will also be referred to in other parts of this thesis. Section 2.5 continues with a graph representation of semistructured data and of XML. Finally, Section 2.6 introduces the notion of *rooted graph simulation*, which is a similarity relation between two graphs that can serve as a foundation for both, querying graph structured data and validating graph structured data against a schema, and which is thus the base for the language Xcerpt presented in this thesis.

Beyond the information provided in this Chapter, interested readers might find a good introduction into the history of the Web in Tim Berners-Lee's book *Weaving the Web* [18]. A more thorough overview over semistructured data and the Web is provided by the book *Data on the Web* [4].

2.1 Semistructured Data

2.1.1 Traditional Database Systems

Traditional database management systems (DBMS) – i.e. object-oriented, relational, hierarchical or network database management systems – require to specify a rigid *schema* in advance of storing any data. A schema defines in which structures data items have to be arranged if they are to be stored in the DBMS. For instance, in relational DBMS, the schema definition specifies which relations are available and how many and what kinds of fields they allow.

Such a schema definition is *rigid* in the sense that (1) all data must adhere to it, and (2) it has to be defined in advance. Whereas this restriction might be sensible in traditional database systems that store a

[1] A premier example of this is *Wikipedia*, the online encyclopedia, to which anyone can contribute with minimum technical efforts. (`http://www.wikipedia.org`)

huge amount of very uniform data, it is undesirable on the Web, where the data is very heterogeneous and might come from different sources. It furthermore deters many users from authoring content, as they either fear or are not proficient with in-advance schema definitions and prefer to let the schema evolve during the authoring process.

A further disadvantage of this approach is that data is usually not self-explanatory and invalid without its schema information. For example, a tuple of `("Mickey","Mouse",19281118)` does not fully convey what kind of information it represents. The fact that "Mickey Mouse" is probably a name can be guessed from the data with reasonable certainty. But the meaning of the value 19281118 is unclear without further schema information, it might for example be the birth date, a social security number, or a telephone number[2]. However, self-explanatory data is desirable on the Web, as such data is often exchanged between different parties (and thus systems) not sharing a common schema or data representation format.

Of course, it has to be mentioned that a rigid schema definition also has many advantages, most of them of technical nature (optimisability, data storage) and some of social nature (author has to think about the schema in advance), which are the reasons why rigid schemas are well established in current database systems.

2.1.2 Semistructured Data

Semistructured data has been of interest in database research since the mid-nineties [3, 4, 30, 36, 62, 83]. In contrast to the traditional database management systems described above, semistructured data does not require a schema definition. For this reason, semistructured data was first referred to as *unstructured data* [30]. This term has been abandoned because it does not convey that semistructured data *does* have structure, the structure is merely not given separately and in advance, but instead is part of the data. The term *semistructured data* describes such data more adequately: neither is the data fully structured with a rigid, in-advance schema (like in traditional database management systems) nor is it completely unstructured (like raw images or plain text). Instead, it is "structure-carrying" or "self-describing" and thus allows very flexible structuring of the data.

Semistructured data is syntactically represented by *semistructured expressions*, which are very similar to term structures in logic or functional programming languages. The example above can be represented as a semistructured expression as follows:

```
person [
  name [
    first [ "Mickey" ],
    last  [ "Mouse"  ]
  ],
  phone [ "19281118" ]
]
```

This data item is self explanatory, as the structure is part of the data.

Semistructured data is not limited to flat tuples or tree structured data as the example above might imply. Graph structures can be represented in semistructured expressions by means of *object identifiers* and *references*. The following extension of the address book adds a subexpression `knows` to the two person entries so as to represent that `Mickey Mouse` knows `Donald Duck` and vice versa:

```
address-book [
  &o1 @ person [
    name [
      first [ "Mickey" ],
      last  [ "Mouse"  ]
    ],
    phone [ "19281118" ],
    knows [ &o2 ]
```

[2] It is, in fact, the (assumed) birth date (18th of November, 1928) of Mickey Mouse

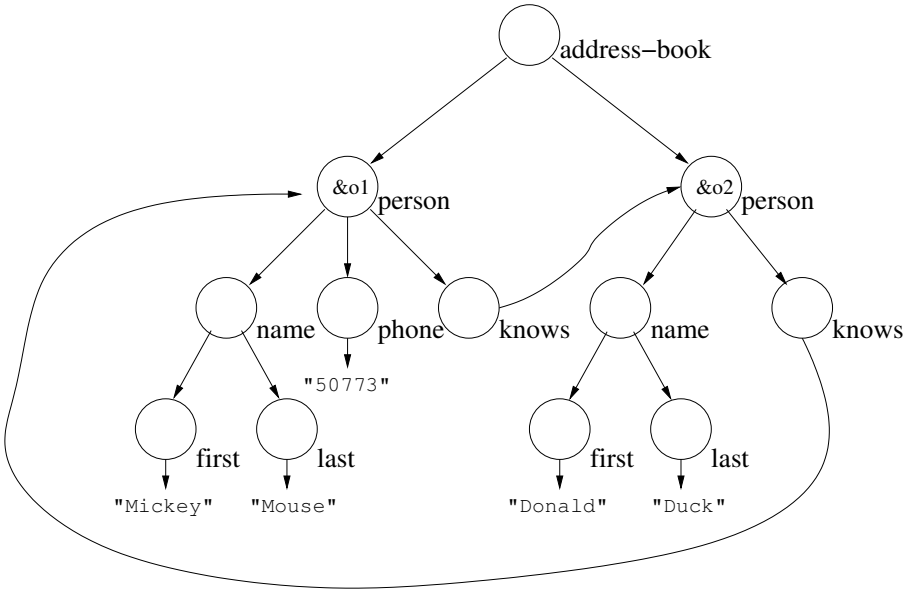

Figure 2.1: Graph induced by a semistructured expression

```
  ],
  &o2 @ person [
    name [
      first [ "Donald" ],
      last  [ "Duck"  ]
    ],
    knows [ ^&o1 ]
  ]
]
```

Note also, that this semistructured "database" has two entries with differing structure: whereas the first entry contains a phone number, the second does not.

Figure 2.1 illustrates the graph induced by the example above. Note that this thesis uses node-labelled graphs whereas in [4], graphs are edge-labelled.

Expressions of the form `&n` are called *references* or *object identifiers* (oid). Occurrences of the form `^&o1` are *referring occurrences*, occurrences of the form `&o1 @ person [ ... ]` are *defining occurrences* of an oid. An object identifier can be defined exactly once, but referred to 0 or more times. If an oid is referred to, then it has to be defined as well.

Furthermore, it is often useful to distinguish between *ordered* and *unordered* data. Consider for instance a publication list of the following form[3]:

```
publications [
  book [
    title [ "Folket i Birka på Vikingarnas Tid" ],
    authors [
      author [ "Mats Wahl" ],
      author [ "Sven Nordqvist" ]
      author [ "Björn Ambrosiani" ]
    ]
  ],
```

[3]The titles translate from Swedish to English as "The people of Birka in the Viking Age" and "The Book about Vikings"

```
  book [
    title [ "Boken Om Vikingarna" ],
    authors [
      author [ "Catharina Ingelman-Sundberg" ]
    ]
  ]
]
```

This semistructured expression contains both ordered and unordered content: whereas it might be irrelevant whether the first book occurs before the second, the order of authors might be significant for correct citations. Unordered content leaves a certain amount of freedom for storage systems, in that they can decide to rearrange data for more efficient storage or for building indexes. It is thus convenient to provide constructs for expressing ordered sequences (denoted by `[...]`) and unordered sequences (denoted by `{...}`). With this extension, the semistructured database above can be represented as follows:

```
publications {
  book {
    title [ "Folket i Birka på Vikingarnas Tid" ],
    authors [
      author [ "Mats Wahl" ],
      author [ "Sven Nordqvist" ]
      author [ "Björn Ambrosiani" ]
    ]
  },
  book {
    title [ "Boken Om Vikingarna" ],
    authors [
      author [ "Catharina Ingelman-Sundberg" ]
    ]
  }
}
```

The syntax chosen for semistructured expressions in this Section deliberately deviates from other syntaxes as described e.g. in [4] and is closer to the syntax of the language Xcerpt introduced later in this thesis. More formally, semistructured expressions are defined as follows using a context free grammar (following a Definition in [27] and [28]):

```
1              <sse> := ( oid "@" )? ( label "{}" | label <list>) .
2             <list> := <ordered-list> | <unordered-list> .
3    <ordered-list> := "[" <sse-or-reference> ( "," <sse-or-reference> )* "]" .
4  <unordered-list> := "{" <sse-or-reference> ( "," <sse-or-reference> )* "}" .
5 <sse-or-reference> := <sse> | '"' string  '"' | "^" oid .
```

In this grammar, expressions between < and > are non-terminal symbols (or variables). `@`, `^`, `"["` , `"{"`, `"]"` and `"}"` are terminal symbols. `oid` and `label` denote object identifiers and expression labels (tag names), respectively.

If a semistructured expression e is of the form $\texttt{label}[t_1, \ldots, t_n]$ or $\texttt{label}\{t_1, \ldots, t_n\}$, then the t_i are called *subexpressions of e*.

2.1.3 Other Languages for Representing Semistructured Data

Besides the semistructured expressions used above, several other formalisms have been proposed in the literature, most notably *OEM/Lore* and *ACeDB*, which are briefly introduced below:

OEM/Lore

OEM, the *object exchange model*, was developed as part of the *Tsimmis* project at Stanford [36, 83, 54], a project aiming at integrating heterogeneous information sources. According to [4], "an OEM object is a quadrupel (`label,oid,type,value`), where `label` is a character string, `oid` is the object's identifier, and `type` is either `complex` or some identifier denoting an atomic type (like `integer`, `string`, `gif-image`, etc.). When `type` is `complex`, then the object is called a *complex object*, and `value` is a set (or list) of oids. Otherwise, the object is an *atomic object* and `value` is an atomic value of that type." In the original OEM, graphs are thus node labelled, like in this thesis.

Most other representation formats are variants of OEM, so that OEM can be considered the de facto standard for representing semistructured data. In particular, Lore [2] is an edge-labelled variant of OEM, and used as the primary representation for semistructured data in [4] and [31].

In Lore, the address book example used above is expressed as follows:

```
{address-book: {
  person: &o1{
    name: { first: "Mickey", last: "Mouse" },
    phone: 19281118,
    knows: &o2
  },
  person: &o2{
    name: { first: "Donald", last: "Duck" },
    knows: &o1
  }
}
```

OEM/Lore does not differentiate between ordered and unordered content.

ACeDB

ACeDB (A *C. elegans* Database) [62, 4, 21] is a database system originally developed to store genetic data of a specific organism (*C. elegans*). As such data is usually rather heterogeneous and often incomplete, ACeDB is capable of dealing with missing parts or loose structure and can thus be considered as a general format for representing semistructured data, although ACeDB initially was not developed for this purpose.

Unlike other languages for semistructured data, ACeDB requires the definition of a schema. However, the data is not required to strictly conform to it, data items may be missing. Also, the data itself still carries the structure necessary to identify data items.

The address book example can be expressed using ACeDB as follows:

```
?Person name UNIQUE first Text
                    last  Text
        phone Text
        knows ?Person

&o1 name first "Mickey"
         last  "Mouse"
    phone "19281118"
    knows &o2

&o2 name first "Donald"
         last  "Duck"
    knows &o1
```

The first block defines the schema of the data. A person must have a unique name, and may have a phone number and know other persons. The second and third block comprise the data items representing the two persons.

2.2 XML – the *Extensible Markup Language*

This section introduces into XML, the *eXtensible Markup Language*. Section 2.2.1 begins with a short introduction into the concept of *markup languages* and Section 2.2.2 briefly summarises the history and motivation that lead to the development of XML. The *anatomy of an XML document* (Section 2.2.3) introduces into the building stones from which XML documents are composed, namely the *prologue*, *character sets*, *elements*, *character data*, *attributes* and *entities*. This introduction is not complete, but should provide a good understanding for XML. Section 2.2.4 briefly introduces two XML schema languages, *DTDs* and *Relax NG*. The XML reference mechanisms *ID/IDREF* is presented in Section 2.2.5. In order to use elements structured according to different schemas within a single document, XML supports so-called *namespaces*, which are discussed in Section 2.2.6.

2.2.1 Markup Languages

The notion *markup* originates from the verb "to mark up", which means to annotate text documents with formatting instructions for use in type setting. A *markup language* defines a set of valid instructions for marking up text documents. Three kinds of markup languages are usually differentiated: (1) *specific* or *layout oriented* markup languages, (2) *generalised* or *structure oriented* markup languages, and (3) *generic* or *meta* markup languages.

Specific or Layout Oriented Markup Languages

Specific or layout oriented markup languages contain a fixed set of *formatting instructions* that may be used to mark up text documents (e.g. *bold* or *italic*). Examples for such languages are *PostScript* [89], which provides instructions for typesetting a document on a printer, or *HTML* (the *hypertext markup language*) [109, 107], which is a markup language that provides instructions for rendering documents in a Web browser. For example, the HTML expression `<b>Mickey Mouse</b>` formats the text `Mickey Mouse` in bold font.

Generalised or Structure Oriented Markup Languages

Generalised or structure oriented markup languages contain a fixed set of instructions that allow to structure a document logically (e.g. in *chapters* and *sections*). Examples for such languages are the venerable *DCF GML* (*Document Composition Facility Generalized Markup Language*) [53] or *DocBook*[4], a language designed for structuring documents into chapters, sections, etc.

Generic or Meta Markup Languages

Generic or meta-markup languages allow to define custom markup languages. They do not specify any markup, but are instead a means for defining new markup, both for structure and presentation. The most widespread such language is *SGML*, the *Standard Generalized Markup Language* [1], but it is increasingly superseded by XML. In spite of its name[5], SGML is a meta markup language that originated in the document management community as a unified method for defining markup languages (called *SGML applications*) for structured documents. For example, both HTML and DocBook are specified as an SGML application.

An SGML application is defined in terms of a grammar specified in a *document type definition* (DTD), which is like a database schema definition, albeit for text markup. SGML schemas are mandatory: every SGML document is required to be associated with and conform to a schema.

[4] `http://www.docbook.org`

[5] SGML was developed as a refinement of DCF GML and kept the name "generalised" from that language

2.2.2 A Generic Markup Language for the Web

Although HTML has many deficiencies and has thus undergone significant changes, the Web of the year 2004 mostly consists of HTML documents containing text content with presentational markup. There are probably three reasons for this: (1) HTML allows users to easily define hyperlinks (i.e. clickable references to other documents), (2) HTML is *open*, i.e. it is not restricted to a specific product and available to anyone, and (3) HTML is *simple to use* and thus enables a wide variety of users to author content [18].

However, the limitation to presentational markup is one of HTML's most significant disadvantages: most data available on the Web is – although machine *processable* – not machine *understandable*, i.e. computers are able to process and display the data but have no means to reason with it. For example, a web shop might list articles with prices in an HTML table, but the browser of a user is not aware of the fact that certain numbers are prices whereas other numbers merely identify products.

A generic markup language like SGML would improve this situation significantly: authors of Web pages can "mark up" all prices with custom markup such that they can be automatically recognised by machines. The problem with SGML is that it is complicated to use, as it always requires to adhere to a possibly complex schema, and allows many abbreviations (like omitting of closing or opening tags or tag minimisation) that can lead to confusion or ambiguities.

Recognising the deficiencies of HTML and SGML, the *World Wide Web Consortium* (W3C) proposed in 1996 the language XML, the *eXtensible Markup Language* [116], which aims at unifying the advantages of SGML and HTML in providing a meta-markup language that allows to define custom markup, but which keeps the simplicity of HTML so as to enable a wide range of authors and application programmers to use it. In particular, XML simplifies SGML in the following aspects:

- it removes ambiguous constructs like *tag minimisation* (as in `<b>Mickey Mouse</>`), *interleaved opening and closing tags* (as in `<b><i>Mickey Mouse</b></i>`), etc.
- it allows documents without a schema definition
- it supports hyperlinks and references
- is is intended to be used not only for documents but also for data items

Or to quote Tim Berners-Lee in an *Scientific American* article entitled "XML and the Second-Generation Web"[6]:

> "Just as HTML created a way for every computer user to read Internet documents, XML makes it possible, despite the Babel of incompatible computer systems, to create an Esperanto that all can read and write. Unlike most computer data formats, XML markup also makes sense to humans, because it consists of nothing more than ordinary text."

Although initially developed primarily as a document representation format, "to meet the challenges of large-scale electronic publishing"[7], XML is now increasingly being used for data exchange and storage. The development of native XML database systems like Xindice[8], eXist[9] or Tamino[10], and data exchange formats like SOAP[11] bear testimony of this development.

Nowadays, it is necessary to differentiate between XML as a *language* and XML as an *activity*, which contains a plethora of different developments centered around XML, like *XML Schema*[12], *XML Linking*[13] and *XML Query* [14]. While many of these activities are often criticised for their complexity and redundancy, XML as a language is mostly considered to be well established.

[6] http://www.sciam.com/article.cfm?articleID=0008C786-91DB-1CD6-B4A8809EC588EEDF
[7] http://www.w3.org/XML/
[8] http://xml.apache.org/xindice/
[9] http://exist.sourceforge.net/
[10] http://www.softwareag.com/tamino/
[11] http://www.w3.org/TR/soap12-part0/
[12] http://www.w3.org/XML/Schema
[13] http://www.w3.org/XML/Linking
[14] http://www.w3.org/XML/Query

2.2.3 Anatomy of an XML Document

An "XML document" is a file, or collection of files, that adheres to the general syntax specified in the XML Recommendation [116], independent of the concrete application. XML documents consist of an optional *document prologue* and a *document tree* containing *elements*, *character data* and *attributes*, with a distinguished root element.

Document Prologue

The document prologue is used to define properties of an XML document, like the version of XML used, the character encoding, processing instructions and schema information. It consists of the following parts:

- a mandatory *XML declaration* denoted by `<?xml version="1.0" ...?>` which specifies the version of XML used, and optionally the encoding of the document.
- zero or more application specific *processing instructions* that may be evaluated when loading an XML document denoted by `<?target data?>`, where `target` identifies the application to which the instruction is directed, and `data` represents additional information for the application
- an optional schema declaration in terms of a DTD, defined either internally, or system file, or as a public identifier associated with a DTD which is assumed to be known to the processing program[15].

Example 2.1 (XML Document Prologue)
The following document prologue initiates an XML document in DocBook format (the DTD of which is identified by a public identifier), to be processed with a stylesheet `stylesheet.css` and an encoding of `ISO-8859-15` (Western Europe with Euro):

```
<?xml version="1.0" encoding="ISO-8859-15"?>

<?stylesheet href="stylesheet.css"?>

<!DOCTYPE book PUBLIC "-//Norman Walsh//DTD DocBk XML V1.4/EN"
                      "http://docbook.org/docbook/xml/1.4/db3xml.dtd">
```

Although several improved schema languages like XML Schema [111] and Relax NG [39] for XML exist, both XML 1.0 and the recently released XML 1.1 only support the declaration of DTD schemas in the document prologue (see schema languages below).

Character Set and Encodings

Since the Web is a place containing documents in many different languages, XML has been designed as an internationalised language from the beginning. XML supports all characters defined in ISO/IEC 10646 (a superset of Unicode[16]), amounting to approximately 4 billion. To represent these characters in concrete documents, XML supports a variety of encodings, which can be specified in the XML declaration of the document prologue. Table 2.1 lists some of the more frequent character encodings[17]. Of these encodings, XML language processors need to implement at least `UTF-8` and `UTF-16`.

Elements

Elements are used to "mark up" the document. They are identified by a label (called *tag name*) and specified by opening and closing tags that enclose the element content. Opening tags are of the form `<label ...>` and contain the label and optionally a set of attributes (see below). Closing tags are of the form `</label>` and contain only the label. Labels start with either an alphabetical character (with respect to the defined character encoding) or with underscore _. They may contain any alphanumeric characters, and the signs _, `-`, `:` and `.`. The character `:` is reserved for separating namespace prefixes from element names.

[15] public identifiers are commonly used for widespread XML applications like XHTML or DocBook
[16] `http://www.unicode.org`
[17] A comprehensive list can be found at WikiPedia: `http://en.wikipedia.org/wiki/Character_encoding`

ASCII	*American Standard for Character Information Interchange*, 7-bit
Big5	Traditional Chinese, Hong-Kong and Taiwan, 2 byte
GB2312	Simplified Chinese (*Guójiā Biāozhǔn Mǎ*), People's Republic of China, 2 byte
ISO-2022-JP	Japanese, 1-2 bytes variable length (compatible to ASCII)
ISO-8859-1	Latin, Western European without Euro, 8-bit
ISO-8859-2	Latin, East European, 8-bit
ISO-8859-15	Latin, Western European with Euro, 8-bit
KOI8-R	Cyrillic, Russian, 8-bit
UTF-8	Unicode, 1-4 bytes variable length (compatible to ASCII)
UTF-16	Unicode, 2 byte

Table 2.1: Frequently used character encodings in XML

Example 2.2 (XML Elements)

```
<address-book>
  content
</address-book>
```

Elements may contain either other elements, character data, or both (*mixed content*). In analogy with the document tree, such content is often referred to as *children* of an element. Interleaving of the opening and closing tags of different elements (e.g. `<b><i>Text</b></i>`) is forbidden. The order of elements is relevant (so-called *document order*). This is a reasonable requirement for storing text data, but might be too restrictive when storing data items of a database. Applications working with XML data thus often ignore the document order. If an element contains no content, it may be abbreviated as `<label/>`, i.e. the "closing slash" is contained in the start tag.

Example 2.3 (Empty Elements)
In HTML, line breaks are indicated by an empty element with label `br`. In XML syntax, this is specified as

```
<br/>
```

An XML document always contains a distinguished element called the *root element* that encloses all other content of the document. If a schema is associated with an XML document, then the root element has to be an instance of this schema in order for a document to be *valid* . Documents that do not conform to a specified schema, but otherwise adhere to the XML specification are *invalid*, but *well-formed*.

Character Data

Besides elements, XML documents may contain character data. In general, character data is written "as-is", i.e. it is not enclosed in special symbols like in many programming languages or the semistructured expressions above.

Example 2.4 (Character Data)
The following XML document contains character data mixed with element content:

```
<document>
  ...
  The quick brown fox <highlight>jumps</highlight> over the lazy dog.
  ...
</document>
```

Whitespace in character data is ignored and certain reserved characters (like <) are disallowed. Therefore, XML provides an additional construct for escaping character data, so-called *CDATA sections*. CDATA sections are enclosed in `<![CDATA[` and `]]>`.

Example 2.5 (CDATA Sections)
The following is only character data and does not contain markup:

```
<![CDATA[The quick brown fox <highlight>jumps</highlight> over the lazy dog.]]>
```

Attributes

Opening tags of elements may contain a set of key/value pairs called *attributes*. Attributes are of the form `name = "value"` where `name` may contain the same characters as element labels and value is a character sequence which is always enclosed in quotes `"` and in which white space is insignificant. An opening tag may contain attributes in any order, but each attribute name can occur at most once.

Example 2.6 (XML Attributes)

```
<person id="mickey mouse">
  <name>
    <first>Mickey</first>
    <last>Mouse</last>
  </name>
  <phone type="home">19281118</phone>
</person>
```

XML defines certain reserved attributes, currently `xml:lang` (which defines the language of the element content) and `xml:space` (which in XML 1.1 defines that whitespace is significant). Furthermore, certain extensions of XML, like *XLink* [110] and *XML Namespaces* [117], reserve attributes prefixed by `xlink:` and `xmlns:`.

Example 2.7 (`xml:lang`)
The reserved attribute `xml:lang` may be used to specify the language of element contents. This may e.g. be used to specify two different titles for a book:

```
<book>
  <title xml:lang="sv">
    Folket i Birka på Vikingarnas Tid
  </title>
  <title xml:lang="de">
    Die Leute von Birka. So lebten die Wikinger.
  </title>
  <title xml:lang="en">
    The people of Birka in the Viking Age
  </title>
</book>
```

Entities

XML entities are a macro mechanism for reusing commonly used content. In particular, the reserved characters `<` and `&` can be expressed using entities. Note that, unlike many other macro mechanisms, XML entities cannot be parametrised.

Entities are defined in the document type definition in the prologue of an XML document (or in an external DTD) with the construct `<!ENTITY name "value">`, which defines the entity `name` to be an abbreviation for `value`. `value` may contain any content, including markup. *Entity references* have the form `&name;`, where `name` is the name of an either predefined or previously defined entity. The occurrence of `&name;` is then literally replaced by the value of the entity.

Example 2.8 (Entities)
The following example defines an entity `warn` to be the data `<bold>Warning:</bold>` (i.e. the word “Warning” printed in bold face) and refers to it later by `&warn;`:

```
<?xml version="1.0" encoding="ISO-8859-15"?>

<!DOCTYPE paragraph [
  <!ELEMENT paragraph ANY >
  <!ENTITY warn "<bold>Warning:</bold>">
```

`&amp;`	`&`
`&lt;`	`<`
`&gt;`	`>`
`&apos;`	’
`&quot;`	"
`&#x;`	the ISO/IEC 10646 character with hexadecimal number x

Table 2.2: Predefined character references available in XML

```
]>

<paragraph>
  &warn; Don’t ever try this out yourself.
</paragraph>
```

Entities can also be used for *character references*. For example, `<` refers to the letter `<`, which is otherwise not allowed in character data. Table 2.2 summarises character references that may be used in XML.

Example 2.9 (Character References)
The following character reference includes the character α, which has the hexadecimal number 0x03B1 (or 945 in decimal format): `The character &#03B1;` is rendered as *The character* α.

A third application of entities that is of interest is the possibility to include binary data in an XML document, like a PNG (*Portable Network Graphics*) image with so-called *external entities*.

Example 2.10 (External Entities with Binary Content)
The following external entity includes the PNG image `figure.png` in the XML document:

```
<!ENTITY figure SYSTEM "./figure.png" NDATA png>
```

2.2.4 XML Schema Languages

An XML schema language describes what structure an XML document is allowed to have. For example, the schema of an address book might specify that all entries are required to contain a name, but the phone number is optional. Unlike SGML or relational databases, an XML document is not required to *conform* to a schema, or even to *have* a schema, but if it does, it is called *valid* with respect to its schema. However, schema definitions are advantageous as they allow for automatic optimisations, may be used to communicate the admissible structure between authors and may support the authoring process if schema-aware editors are used.

Several languages for defining schemas are available. This section briefly introduces the languages *DTD* (as it is part of the XML specification) and *Relax NG* (as it is more flexible than DTD and used for schema specifications in this thesis). Other approaches (like *XML Schema* [111] or *Schematron* [63]) are not discussed here, as schema languages are outside the scope of this thesis.

DTD

The specification for XML DTDs, or *document type definitions*, is included in the W3C XML recommendation [116]. DTDs allow to define the possible structure of XML documents in terms of a tree grammar. DTDs allow four kinds of *markup declarations*:

Element Declarations have the form `<!ELEMENT label content>`, where `label` is the element label and `content` defines what kind of content an element may have. The content definition has the following structure:

- `(child`$_1$`,...,child`$_n$`)` denotes an ordered sequence of child elements of types `child`$_1$`, ..., child`$_n$.
- `(child`$_1$`|...|child`$_n$`)` denotes *n* alternatives of child elements of types `child`$_1$`, ..., child`$_n$.
- `child?` denotes optional child elements, sequences or alternatives
- `child*` denotes repetition of child elements, sequences or alternatives (0 or more)
- `child+` denotes repetition of child elements, sequences or alternatives (1 or more)
- `(#PCDATA)` denotes character content
- `ANY` denotes that the element may contain any content
- `EMPTY` denotes that the element may not contain content

If an element is allowed to contain both other elements and character data, it is said to be of *mixed content*.

In DTDs, it is not possible to associate types like *integer* or *float* with character content.

Attribute List Declarations have the form

```
<!ATTLIST element-label att-name1 att-type1 # qualifier
                        ...
                        att-namen att-typen # qualifier>
```

where `element-label` is the label of the element to which the attributes belong, `att-name`$_i$ is the name of the i^{th} attribute and `att-type`$_i$ is the type definition of content of the i^{th} attribute. The most important type definitions are[18]:

- `CDATA` for character content
- `ID` for defining occurrences of identifiers (see ID/IDREF in Section 2.2.5)
- `IDREF` for referring occurrences of identifiers (see ID/IDREF in Section 2.2.5)
- `(token`$_1$`,...,token`$_n$`)` for defining *n* alternative attribute values `token`$_1$`,...,token`$_n$, where a *token* conforms to the same syntax requirements as an element label.

Each attribute pair definition has an additional `qualifier`, which is one of `REQUIRED` (all instance elements must have the attribute), `IMPLIED` (instance elements may have the attribute) or `FIXED value`, where `value` is the fixed value of the attribute.

Entity Declarations have already been described in Section 2.2.3.

Notation Declarations are used to identify external binary formats and specify helper applications that can be used to process the format. For example, the notation declaration

```
<!NOTATION png SYSTEM "/usr/bin/gimp">
<!ATTLIST picture format NOTATION (png | jpg)>
```

specifies that pictures take a format attribute with values of either `png` or `jpg` and that the helper application for the notation `png` is called via the system call `/usr/bin/gimp`. Notation declarations are apparently only rarely used.

Example 2.11 (DTD)
The following DTD defines a grammar for the address book used earlier:

[18] other types are `NMTOKENS`, `ENTITY` and `NOTATION`

```
<!ELEMENT address-book (person*)>
<!ELEMENT person (name,phone?)>
<!ELEMENT name   (first,last)>
<!ELEMENT first  (#PCDATA)>
<!ELEMENT last   (#PCDATA)>
<!ELEMENT phone  (#PCDATA)>

<!ATTLIST person oid   ID    # REQUIRED
                 knows IDREF # IMPLIED  >
```

Although DTDs are very widespread, they have several significant deficiencies:

- they do not allow context dependent definitions of elements; it is thus not possible to define that the `name` child element of a `person` has a different structure than the `name` child element of a `company` in the same document.
- they do not support typed content; for instance, it might be desirable to restrict a phone number to only digits and dashes.
- it is difficult to express repetitions of cardinality `n-m`.
- their syntax differs from the syntax used for XML documents.

A further criticism of DTDs is that they are not restricted to specifying the schema of a document or database, but also allow to define content, e.g. in entities, in ID/IDREF, or in attribute default values (qualifier `FIXED` above). Thus, a DTD can be considered a *preprocessing specification* rather than a mere *schema specification*.

XML DTDs differ from SGML DTDs in various aspects. Most importantly, XML DTDs are in contrast to SGML DTDs *case sensitive* and allow all ISO/IEC 10464 characters. On the other hand, XML DTDs do not support unordered content specification (SGML's & operator)[19].

Relax NG

Relax NG [39, 102] is a schema language defined by the *Oasis Open* consortium and thus developed independently from XML Schema and the W3C. It has recently been adopted as an ISO/IEC standard and is used by IETF[20] as the "official" XML schema language for defining the schema of IETF publications in XML format. It has a solid formal foundation in the theory of tree automata (Relax NG uses so-called *hedge automata*), and is rather easy to learn and use, while still being flexible and more expressive than DTD. Relax NG has an XML-based syntax and a so-called *compact* notation, which resembles the syntax for semistructured expressions used in this thesis. Whereas the XML syntax allows to use many existing tools like editors or browsers, the compact notation is much terser and easy to read for human users.

This section provides an short intuitive introduction into Relax NG's compact notation. Readers interested in the XML notation or in a more thorough description of Relax NG should refer to [39] or [102], or to Relax NG's website[21].

A Relax NG schema is defined in terms of production rules in a *regular tree grammar*, where the left hand side is always a non-terminal symbol (where a distinguished non-terminal symbol called `start` is the start symbol of the grammar) and the right hand side is either an element definition, an attribute definition, text, a datatype from an external library (e.g. XML Schema Datatypes [112]) or an ordered or unordered list of the former constructs. Elements are introduced by the keyword `element`, followed by the label and the specification for the admissible children. For example, the grammar rule

```
nphone = element phone { text }
```

[19] a good summary can be found at `http://www.xml.com/pub/a/98/07/dtd/`

[20] IETF is an abbreviation for the so-called *Internet Engineering Task Force*, which defines many of the standards the Internet builds upon, e.g. TCP/IP, HTTP or the various email standards (`http://www.ietf.org`)

[21] `http://www.relaxng.org`

defines the non-terminal symbol `nphone` to describe elements with label “phone” and only text content.

The specification of admissible children is a list of further definitions or non-terminals separated by `,` (ordered sequence), by `&` (unordered set), or by `|` (alternatives). For instance, the following grammar defines the non-terminal `name` to be elements with label “name” and unordered child elements with labels “first” and “last”:

```
nname =
  element name {
    element first { text } &
    element last  { text }
  }
```

In the specification of admissile children, it is (as usual in grammars but in contrast to DTD) possible to replace any non-terminal by its definition and vice versa to improve readability. For example, in the definition of `name` above, it would be possible to replace the definitions of the elements `first` and `last` by non-terminals as follows:

```
nname  = element name { nfirst & nlast }
nfirst = element first { text }
nlast  = element last  { text }
```

Repetitions of elements may be specified using the operators `*` and `+` like in regular expressions and DTDs. The following grammar rule specifies that an `address-book` element may contain an arbitrary number of children defined by the non-terminal `person`:

```
start = element address-book { person* }
```

Attributes are defined using the keyword `attribute`, followed by the attribute name and the specification of the admissible values. They are defined in the same manner as elements. The following grammar rule completes the addressbook schema by providing the definition for persons. Note that Relax NG supports to import external datatype libraries, in this case *XML Schema Datatypes* (prefix `xsd:`).

```
person =
  element person {
    attribute oid { xsd:ID } &
    attribute knows { xsd:IDREF } &
    name &
    phone*
  }
```

Example 2.12 (Relax NG)
The following Relax NG document again summarises the schema for the address book (in compact notation).

```
1  datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"
2
3  start = element address-book { person* }
4  person =
5    element person {
6      attribute oid { xsd:ID } &
7      attribute knows { xsd:IDREF } &
8      element name {
9        element first { text } &
10       element last  { text }
11     } &
12     element phone { text }*
13   }
```

For many applications, Relax NG is more suitable than DTD, because it is more expressive than DTD (in DTD, it is e.g. not possible to define two elements with label "name" but differing structure, once for a person, once for a company). In addition, Relax NG grammars are easy to read and maintain, as their syntax is close to the usual syntax of grammars, and they are more flexible than DTDs as they allow to structure a grammar independent from the structure described by the grammar. For these reasons, Relax NG is used for XML schema definitions in this thesis.

2.2.5 XML References: ID/IDREF

References in XML serve two purposes. The first, implemented by *ID/IDREF*, is to support cross references within a document. In the document context, this may be used to refer to the bibliography, or to other sections in the text. In the same manner, it may also be used within data items of a database to form graph structures. Thus *ID/IDREF* is a *structural* reference mechanism. The second, implemented by *XLink*, is to connect several documents – possibly even at different locations – with so-called *hyperlinks*. Such references cannot be considered as structural, they are rather at the level of the content and typically only resolved by explicit user interaction (e.g. clicking on a link in a browser). Since this thesis does not investigate browsing aspects, only ID/IDREF is described here.

ID/IDREF references are implemented by using two special types of attributes: *identifiers* (denoted by `ID`) are used to specify that the attribute value is a unique identifier for the element containing the attribute, and *identifier references* (denoted by `IDREF`) are used to specify that the attribute value is a reference to an element identified by a unique identifier. Both types need to be associated to attributes by a DTD or other schema definition.

Example 2.13 (ID/IDREF)
The following XML document models the address book of Section 2.1.2. Note that the DTD is used to identify the attributes that are used as identifiers and references.

```
<?xml version="1.0" encoding="iso-8859-15"?>

<!DOCTYPE address-book [
  <!ELEMENT address-book (person*)>
  <!ELEMENT person (name,phone?)>
  ...

  <!ATTLIST person oid   ID    # REQUIRED
                   knows IDREF # IMPLIED  >
]>

<address-book>
  <person oid="o1" knows="o2">
    <name>
      <first>Mickey</first>
      <last>Mouse</last>
    </name>
    <phone type="home">19281118</phone>
  </person>
  <person oid="o2" knows="o1">
    <name>
      <first>Donald</first>
      <last>Duck</last>
    </name>
  </person>
</address-book>
```

Although this reference mechanism is very similar to the OID references of semistructured expressions introduced in Section 2.1.2, the limitation to attributes does not allow to position references beneath the

children of an element in case the order is significant. For example, in the address book above, it is not possible to position the reference `knows` as the *last* child of the `person` elements using ID/IDREF.

2.2.6 XML Namespaces

In many documents, it is desirable to combine parts of several schemas. For instance, the adress book might contain an element `remarks` for each entry, in which it is possible to write free text annotated with certain parts of (X)HTML, like bold or italic face. Unfortunately, such combinations often result in naming conflicts or ambiguities ("does the `name` element refer to company names from one schema or to person names from another?").

Therefore, XML supports so-called namespaces [117] that uniquely associate elements with namespaces. Namespaces are identified by so-called *internationalised resource identifiers* (IRI, [61])[22] Elements in an XML document can be associated with a namespace IRI by so-called *namespace prefixes*, which appear in the opening and closing tags and are separated from the element label by a colon. Namespace prefixes are defined by certain attribute-value pairs and are valid in the scope of the element they are defined in (i.e. in the element itself and in all descendants). Namespace prefixes can be chosen arbitrarily and always resolve to the IRI they are associated with.

It is also possible that two different prefixes are associated with the same namespaces, in which case the elements prefixed with either of them are obviously in the same namespace.

Example 2.14 (XML Namespaces)
In the address book, it might be desirable to complement all entries by an `remarks` element that contains free text remarks, possibly marked up with certain HTML elements, about the entry. The following XML document shows how namespaces can serve this purpose. It uses the namespace prefix `a` to refer to the address book schema, and the namespace prefix `b` to refer to the XHTML schema:

```
<a:address-book xmlns:a="http://www.myschemas.org/address-book"
                xmlns:b="http://www.w3.org/2002/06/xhtml2">
   <a:person a:oid="&o1"a:knows="&o2">
     <a:name>
       <a:first>Mickey</a:first>
       <a:last>Mouse</a:last>
     </a:name>
     <a:phone a:type="home">19281118</a:phone>
     <a:remarks>
       <b:strong>Note:</b:strong> The phone number is also the
       <b:em>birthday</b:em>!
     </a:remarks>
   </a:person>
   <a:person a:oid="&o2"a:knows="&o1">
     <a:name>
       <a:first>Donald</a:first>
       <a:last>Duck</a:last>
     </a:name>
   </a:person>
</a:address-book>
```

In this example, an address book browser might render XHTML elements using an HTML component (like Java's HTML component in the package `javax.swing.text.html`) while displaying all elements of the `http://www.myschemas.org/address-book` namespace in an application specific manner (e.g. in a table).

[22] IRIs are introduced in XML 1.1 and replace the formerly used URIs, or *uniform resource identifiers*. IRIs differ from URIs in that they can contain characters from any character set.

XML namespaces are always declared by using the `xmlns` attribute, either followed by a colon and a namespace prefix (as in the example above), or as a stand-alone attribute (in the latter case, the so-called default namespace is defined for elements that have no prefix). Namespace prefix declarations may appear inside any element and thus have a clearly defined scope. Furthermore, namespace prefix declarations may be shadowed by redefining a namespace prefix within the scope of an already defined prefix with the same name.

2.3 XML, Semistructured Expressions and Semistructured Data

Although XML and semistructured data have initially been independent developments, and initially have been aimed at different application areas (i.e. document representation vs. databases), they have much in common: both represent rooted graph structured data, both are structure-carrying and schema independent and both allow to represent very heterogeneous data items. In fact, XML can be seen as just another syntax for semistructured expressions. For instance, the XML document

```
<publications>
  <book>
    <title>Folket i Birka på Vikingarnas Tid</title>
    <authors>
      <author>Mats Wahl</author>
      <author>Sven Nordqvist</author>
      <author>Björn Ambrosiani</author>
    </authors>
  </book>
  <book>
    <title>Boken Om Vikingarna</title>
    <authors>
      <author>Catharina Ingelman-Sundberg</author>
    </authors>
  </book>
</publications>
```

corresponds to the following semistructured expression:

```
publications [
  book [
    title [ "Folket i Birka på Vikingarnas Tid" ],
    authors [
      author [ "Mats Wahl" ],
      author [ "Sven Nordqvist" ]
      author [ "Björn Ambrosiani" ]
    ]
  ],
  book [
    title [ "Boken Om Vikingarna" ],
    authors [
      author [ "Catharina Ingelman-Sundberg" ]
    ]
  ]
]
```

Beyond this straightforward correspondence, however, a translation of XML into semistructured expressions and vice versa is not always possible. Whereas semistructured expressions can contain both ordered and unordered data (or in the case of OEM only unordered data), XML elements are always ordered. Furthermore, references in XML behave slightly differently than in semistructured expressions (see

Section 2.2.5). In particular, object identifiers in semistructured expressions must be known, i.e. the data must be managed centrally. This restriction is impossible on an open Web. Likewise, semistructured expressions do not have a counterpart for XML attributes, processing instructions, entities, document type definitions, schema languages, XLink hyperlinks, and similar features.

Despite these differences, semistructured expressions are a useful "abstraction" of XML documents. On the one hand, the deficiencies mentioned above are either rarely used (processing instructions, entities) or can be overcome in a straightforward manner (attributes can be seen as flat, unordered elements). On the other hand, semistructured expression provide with a concise representation of semistructured data that is capable of differentiating between ordered and unordered content and has a flexible reference mechanism while at the same time avoiding many constructs that are redundant or add unnecessary complexity.

2.4 Three Scenarios for Querying Semistructured Data

Throughout this document, many examples will be illustrated based on three scenarios introduced in this section. The first two scenarios are examples for semistructured databases: a *student database* represents information gathered during a course and two *book databases* represent the databases of two online bookstores. The last example is more document oriented: it is this *thesis* itself (but for obvious reasons only incomplete). All databases and documents are given both in form of an XML document and in form of a semistructured expression, and are accompanied by a schema definition in Relax NG compact notation, which serves to further illustrate the structure of the data but is not itself used in further examples. In the two database examples, the semistructured expression is given using unordered specification of subexpressions, since this is reasonable in a database context. Note, however, that the XML document representation is always ordered.

2.4.1 Student Database

Imagine a lecture that is accompanied by a course management system (CMS), which manages all kinds of data about students. At the beginning of a teaching term, students have to register with their name, student id and email address. Of course, students are students, and some do not fill in their student id – maybe they have not remembered it and do not have their student registration card ready. During the teaching term, students have to submit their solutions to the weekly exercises before certain deadlines. Since students are not obliged to submit all exercises, each student might have missing exercise entries. Teaching staff correct the solutions and assign scores to each. As teachers usually also have other things to do, a teacher only corrects some exercises at a time and updates the database accordingly, while other solutions might still be uncorrected.

The XML document `students.xml` (given in Figure 2.3) contains the data of such a course management system. It represents a snapshot taken at a certain time: some students have not (yet) submitted their solutions, some submitted solutions have not yet been corrected. Furthermore, information is in parts incomplete, as the student id is missing for some students.

The student database uses the following intuitive schema (a formal schema definition in Relax NG compact notation is given in Figure 2.2):

- each student is represented by a `student` element and its subtree
- each student has at least a name (element `name`), an arbitrary number of email addresses (element `email`) and an optional inscription number (element `matrnr`), which might be missing in case a student did not specify it at the registration.
- for each solution to an exercise a student submitted, the respective entry contains an `exercise` element with at least the exercise number (element `number`), and optionally a score (element `score`), which is used to represent the result of correction.

Naturally, in such a system it is necessary to be able to query certain kinds of data and generate summaries. For example, the CMS might provide a page listing all students with name and email, but the email

```
stud-db = element students {
  element student {
    element name   { text },
    element email  { text }*,
    element matrnr { text }?,
    element exercise {
      element number { text },
      element score  { text }?
    }*
  }*
}
```

Figure 2.2: Schema of the student database in Relax NG [39] compact notation

should be in a format that is not easily parsable by spam address harvesters (so-called "spamvertised" addresses). Also, teachers might be interested in the total scores for each student, in information like "all students that did not submit a solution to exercise 2", or "all students whose submission of the solutions to exercise 3 have not yet been corrected". All of these queries (and some more) will be presented in the remainder of this thesis.

This database is representative for a large number of semistructured databases. Information might be missing (like email address or inscription number), or explicitly not given (exercises that have not been submitted, or are not yet corrected). The individual entries are heterogeneous, for instance there might be several email addresses for a student, but only one for others. Order in such databases is usually irrelevant (indicated by curly braces in the Xcerpt syntax).

2.4.2 Bookstore

The second scenario considers two online bookstores that represent their data in an XML database[23]. Since both use customised applications, the structures and content of the two databases differ: whereas the first bookstore (bookstore A) stores information about title, authors, price and publisher, the second bookstore (bookstore B) does not have information about authors and publisher but instead provides for each book a review part that contains comments made by readers of a book. Furthermore, the data provided by bookstore A is not homogeneous: while some books have a list of authors, others have instead an editor, which contains in addition to first and last name also an affiliation. Figure 2.5 shows the database of bookstore A and Figure 2.6 shows the database of bookstore B.

The Relax NG compact notation of the schemas for both bookstores is given in Figure 2.4. Note that this scenario is in analogy to a use case in the *XQuery Use Cases* [34], from which it differs in two aspects: (1) book entries are augmented by an `authors` element to group the authors. This is useful to illustrate that a database may contain both ordered and unordered content. (2) Instead of the by now rather well known Computer Science books, this thesis uses a set of books about the Viking Age to make reading a little more diversified. Note that parts of the XQuery use case are discussed in more detail later in this thesis (Section 5.1).

As in the student database above, the XML document representation differs slightly from the semistructured expression. One aspect is that the semistructured expressions are unordered (except the list of authors), the other is that attributes have to be represented as subexpressions in a semistructured expression. To this aim, they are grouped within a subexpression with label `attributes` that is always the first subexpression of an expression.

Many queries are conceivable in this scenario. Common queries are to list all titles for an author, or to list books of a certain publication year, with a certain string in the title, and so on. More complex queries could create a summary of prices for both bookstores, or create a mediated list of books with the minimum

[23] Such a database is e.g. accessible at `http://www.amazon.com/webservices`

```xml
<students>
  <student>
    <name>Donald Duck</name>
    <email>donald@duck.org</email>
    <matrnr>123456789</matrnr>
    <exercise>
      <number>1</number>
      <score>15</score>
    </exercise>
    <exercise>
      <number>2</number>
      <score>7</score>
    </exercise>
    <exercise>
      <number>3</number>
    </exercise>
  </student>
  <student>
    <name>Mickey Mouse</name>
    <email>mickey@mouse.org</email>
    <matrnr>987654321</matrnr>
    <exercise>
      <number>1</number>
      <score>3</score>
    </exercise>
    <exercise>
      <number>3</number>
      <score>14</score>
    </exercise>
  </student>
  <student>
    <name>Goofy</name>
    <email>goofy@goofy.org</email>
    <email>goofy@disney.com</email>
    <exercise>
      <number>2</number>
      <score>13</score>
    </exercise>
    <exercise>
      <number>3</number>
    </exercise>
  </student>
</students>
```

```
students {
  student {
    name { "Donald Duck" },
    email { "donald@duck.org" },
    matrnr { "123456789" }
    exercise {
      number { 1 },
      score { 15 }
    },
    exercise {
      number { 2 },
      score { 7 }
    },
    exercise {
      number { 3 }
    }
  },
  student {
    name { "Mickey Mouse" },
    email { "mickey@mouse.org" },
    matrnr { "987654321" },
    exercise {
      number { 1 },
      score { 3 }
    },
    exercise {
      number { 3 },
      score { 14 }
    }
  },
  student {
    name { "Goofy" },
    email { "goofy@goofy.org" },
    email { "goofy@disney.com" },
    exercise {
      number { 2 },
      score { 13 }
    },
    exercise {
      number { 3 }
    }
  }
}
```

Figure 2.3: The student database as an XML document and as a semistructured expression. Note that the semistructured expression is unordered whereas the XML document is ordered.

```
bib  = element bib { book* }
book =
  element book {
    attribute year { text },
    element title { text },
    (authors | editor),
    element publisher { text },
    element price { text }
  }
authors =
  element authors {
    element author { last, first }*
  }
editor =
  element editor { last, first, affil }
last   = element last { text }
first  = element first { text }
affil  = element affiliation { text }
```

```
reviews = element reviews { entry* }
entry =
  element entry {
    element title { text },
    element price { text },
    element review { text }
  }
```

Figure 2.4: Schemas of the two bookstore databases in Relax NG [39] compact notation

price and the name of the bookstore where we can get this minimum price. Also, one could be interested in "which books that A sells are not sold by B".

2.4.3 Document-Centric: PhD Thesis

The last of the three scenarios is this PhD thesis itself, which is an good representative for document-centric data. Not only is the data in such a document usually ordered, it also contains a very heterogeneous structure with deep levels of nesting, and also cross references to bibliographic information and other parts of the document.

The following Relax NG grammar provides a simplified, incomplete definition of the schema for this thesis (incomplete parts are indicated by . . .):

```
report    = element report { abstract, part*, appendix? }
part      = element part { chapter* }
appendix  = element appendix { chapter* }
chapter   = element chapter { title, scontent }
section   = element section { title, scontent }
paragraph = element paragraph { title?, pcontent }
scontent  = paragraph* & section* & bibliography
pcontent  = text* & cite* & emph* & strong*  ...
...
```

For obvious reasons, the following XML document only contains a fragment of this thesis:

```
<report>
  <abstract xml:lang="en">
  ...
  </abstract>
  <abstract xml:lang="de">
  ...
  </abstract>
  <part>
    <title>Introduction and Motivation</title>
  </part>
```

```xml
<bib>
  <book year="1995">
    <title>Vikinga Blot</title>
    <authors>
      <author>
        <last>Ingelman-Sundberg</last>
        <first>Catharina</first>
      </author>
    </authors>
    <publisher>Richters</publisher>
    <price>5.95</price>
  </book>
  <book year="1998">
    <title>Boken Om Vikingarna</title>
    <authors>
      <author>
        <last>Ingelman-Sundberg</last>
        <first>Catharina</first>
      </author>
    </authors>
    <publisher>Prisma</publisher>
    <price>22.95</price>
  </book>
  <book year="1999">
    <title>Folket i Birka på Vikingarnas Tid</title>
    <authors>
      <author>
        <last>Wahl</last>
        <first>Mats</first>
      </author>
      <author>
        <last>Nordqvist</last>
        <first>Sven</first>
      </author>
      <author>
        <last>Ambrosiani</last>
        <first>Björn</first>
      </author>
    </authors>
    <publisher>BonnierCarlsen</publisher>
    <price>39.95</price>
  </book>
  <book year="1997">
    <title>Vikingar i Österled</title>
    <editor>
      <last>Larsson</last>
      <first>Mats</first>
      <affiliation>Lunds universitet</affiliation>
    </editor>
    <publisher>Atlantis</publisher>
    <price>49.95</price>
  </book>
</bib>
```

```
bib {
  book { attributes { year { "1995" } },
    title { "Vikinga Blot" },
    authors [
      author {
        last { "Ingelman-Sundberg" },
        first { "Catharina" }
      }
    ],
    publisher { "Richters" },
    price { "5.95" }
  },
  book { attributes { year { "1992" } },
    title { "Boken Om Vikingarna" },
    authors [
      author {
        last { "Ingelman-Sundberg" },
        first { "Catharina" }
      }
    ],
    publisher { "Prisma" },
    price { "22.95" }
  },
  book { attributes { year { "1999" } },
    title { "Folket i Birka på Vikingarnas Tid" },
    authors [
      author {
        last { "Wahl" },
        first { "Mats" }
      },
      author {
        last { "Nordqvist" },
        first { "Sven" }
      },
      author {
        last { "Ambrosiani" },
        first { "Björn" }
      }
    ],
    publisher { "BonnierCarlsen" },
    price { "39.95" }
  },
  book { attributes { year { "1997" } },
    title { "Vikingar i Österled" },
    editor {
      last { "Larsson" },
      first { "Mats" },
      affiliation { "Lunds universitet" }
    },
    publisher { "Atlantis" },
    price { "49.95" }
  }
}
```

Figure 2.5: The Bookstore Database `bib.xml` of bookstore A as XML document and as semistructured expression.

```
<reviews>
  <entry>
    <title>Folket i Birka på Vikingarnas Tid</title>
    <price>34.95</price>
    <review>
      A children's book telling the story of two siblings
      in the Viking town of Birka; nicely illustrated.
    </review>
  </entry>
  <entry>
    <title>
      Boken Om Vikingarna
    </title>
    <price>24.95</price>
    <review>
      A good description of Viking culture.
    </review>
  </entry>
  <entry>
    <title>Vikingar i Österled</title>
    <price>49.95</price>
    <review>
      History of the Viking travels to Byzantine (Miklagård).
    </review>
  </entry>
</reviews>
```

```
reviews {
  entry {
    title { "Folket i Birka på Vikingarnas Tid" }
    price { "34.95" }
    review {
      "A children's book telling the story of two siblings
      in the Viking town of Birka; nicely painted illustrated."
    }
  }
  entry {
    title {
      "Boken Om Vikingarna"
    }
    price { "24.95" }
    review {
      "A good description of Viking culture."
    }
  }
  entry {
    title { "Vikingar i Österled" }
    price { "45.95" }
    review {
      "History of the Viking travels to Byzantine (Miklagård)."
    }
  }
}
```

Figure 2.6: The Bookstore Database `reviews.xml` of bookstore B as XML document and as semistructured expression.

```
<part>
  <title>The Language Xcerpt</title>
  <chapter>
    <title>Xcerpt: Core Language</title>
    <section>
      <title>Xcerpt Terms</title>
      <paragraph>
        A term in Xcerpt is a representation of a semistructured database, or
        a pattern for querying or constructing such a database. In particular,
        terms may be used for representing XML documents, but they are also
        suited for other semistructured data formats like OEM <cite ref="oem95"/>
        or RDF <cite ref="rdf"/>.
      </paragraph>
      ...
      <section>
        <title>Data Terms</section>
        ...
      </section>
      <section>
        <title>Query Terms</section>
        ...
      </section>
      ...
    </section>
    <section>
      <title>Xcerpt Programs</title>
      ...
    </section>
  </chapter>
</part>
...
<appendix>
```

```
43      <chapter>
44        <title>Full Grammar</title>
45        <section>
46          <title>Data Terms</title>
47          ...
48        </section>
49        ...
50      </chapter>
51      <chapter>
52        <title>XML Syntax</title>
53        <section>
54          <title>Core Xcerpt Terms</title>
55          ...
56        </section>
57        ...
58      </chapter>
59      <chapter>
60        <title>Bibliography</title>
61        <bibliography>
62          <entry id="oem95">
63            <title>Object Exchange across Heterogeneous Information Sources</title>
64            <booktitle>11th Conference on Data Engineering</booktitle>
65            <author>Yannis Papakonstantinou</author>
66            <author>Hector Garcia-Molina</author>
67            <author>Jennifer Widom</author>
68            <year>1995</year>
69          </entry>
70          ...
71        </bibliography>
72      </chapter>
73    </appendix>
74  </report>
```

2.5 Graph Representation of Semistructured Data

Semistructured expressions and XML documents induce graphs in a straightforward manner, which has already been introduced intuitively in Section 2.1, and is further elaborated below. Formally, semistructured expressions can be represented either as an edge-labelled or as a node-labelled graph. This thesis always represents a semistructured expression as a node-labelled graph, as this representation is closer to the graph model of XML specified in the *XML Information Set*[120]. A similar, node-labelled graph representation is also used in the *document object model*, which is a uniform application programming interface for accessing and manipulating XML data. Note, however, that [4] uses edge-labelled graphs instead.

Definition 2.1 (Graph Induced by a Semistructured Expression)
Given a semistructured expression e. The *graph induced by* e is a tuple $G_e = (V, E, r)$, with:

1. a set of *vertices* (or *nodes*) V defined as the set of all subexpressions of e (including e itself)
2. a set of *edges* $E \subseteq V \times V \times \mathbb{N}$ characterised as follows:
   - for all expressions $e_1, e_2, e_3 \in V$: if e_2 is the i^{th} subexpression of e_1 and of the form `^oid` (a reference), and e_3 is of the form `oid @ e'`, with `oid` an identifier and `e'` an expression, then $(e_1, e_3, i) \in E$
   - for all expressions $e_1, e_2 \in V$: if e_2 is the i^{th} subexpression of e_1 and *not* of the form `^oid`, then $(e_1, e_2, i) \in E$

3. a distinguished vertice $r \in V$ called the *root node* with $r = e$

The *label* of a vertice is the label of the subexpression represented by it.

This definition differs in two aspects from the intuitive graph representation used in Section 2.1: (1) vertices represent complete subexpressions instead of only labels, and (2) edges are associated with the respective positions of the subexpressions within the parent. Both properties are necessary to distinguish between ordered and unordered content. Note that for semistructured expressions with unordered content, the position has to be ignored.

Example 2.15 (Graph Induced by a Semistructured Expression)
Consider again the semistructured expression representing an address book with two entries.

```
address-book {
  &o1 @ person {
    name [
      first [ "Mickey" ],
      last  [ "Mouse"  ]
    ],
    phone [ "19281118" ],
    knows [ &o2 ]
  },
  &o2 @ person {
    name [
      first [ "Donald" ],
      last  [ "Duck"  ]
    ],
    knows [ ^&o1 ]
  }
}
```

Note that this expression differs from the expression used in Section 2.1.2 in that certain subexpressions contain unordered content.

Figure 2.7 illustrates the graph induced by this semistructured expression. Note that in contrast to the graph of Figure 2.1 the vertices now comprise subexpressions instead of labels (although space restrictions require that subexpressions are abbreviated by ...) and edges are labelled with the position of the subexpression (indicated in red colour).

As usual, graphs are represented in this thesis with the root at the top and the leaves at the bottom. In this context, the maximum number of immediate subexpressions of a (sub)expression is called the *breadth* and the maximal number of edges from the root to a leaf is called the *depth* of the tree, semistructured expression, or XML document.

2.6 Rooted Graph Simulation – A Similarity Relation for Rooted Graphs

The language Xcerpt presented in this thesis uses *pattern matching* to select data items in a semistructured database (or XML document). A pattern can be considered as an *example* of the data in the database, albeit one that usually is augmented by variables and omits much of the structure that is irrelevant for the selection. A pattern thus has to be *similar* to the queried data.

Pattern matching in Xcerpt (and UnQL, for that matter) is based on a similarity relation between the graphs induced by two semistructured expressions, which is called *graph simulation* [56, 77]. Graph simulation is a relation very similar to graph homomorphisms, but more general in the sense that it allows to match two nodes in one graph with a single node in the other graph and vice versa.

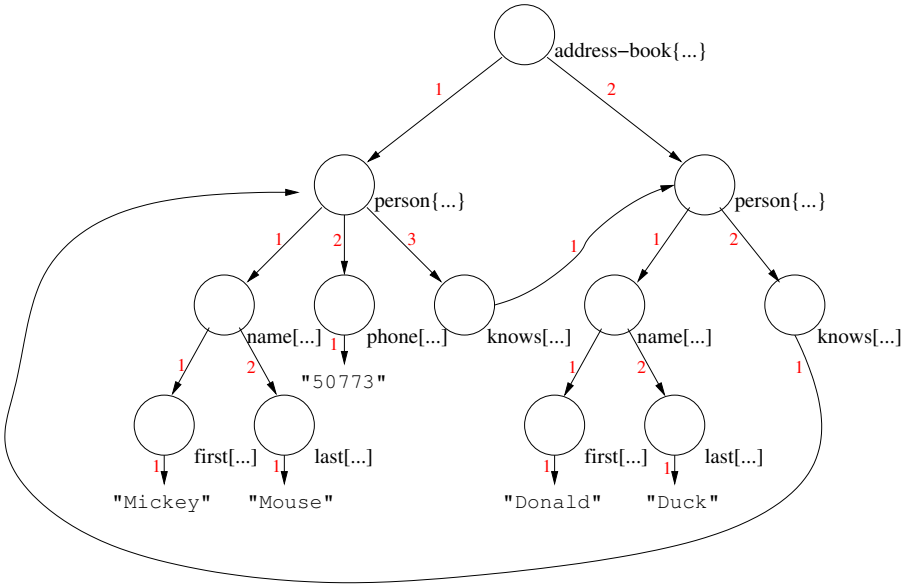

Figure 2.7: Graph Induced by the semistructured expression of Example 2.15.

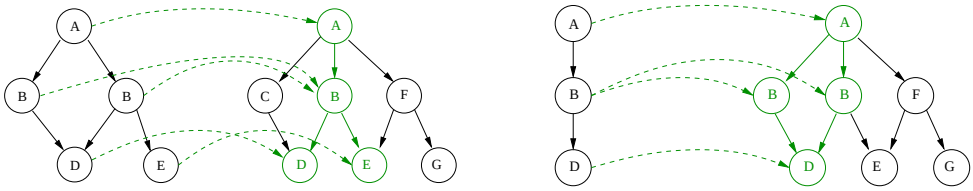

Figure 2.8: Rooted Graph Simulations (with respect to vertex adornment equality)

The following definition is inspired from [56, 77] and refines the simulation considered in [24]. Recall that a (directed) rooted graph $G = (V, E, r)$ consists in a set V of vertices, a set E of edges (i.e. ordered pairs of vertices), and a vertex r called the root of G such that G contains a path from r to each vertex of G. Note that the initial definition of a rooted graph simulation does not take into account the edge labels of graphs induced by a semistructured expression, it is defined on generic, node labelled and rooted graphs. Note furthermore, that in general, there might be more than one simulation between two graphs, which leads to the notion of *minimal* simulations also defined below.

Definition 2.2 (Rooted Graph Simulation)
Let $G_1 = (V_1, E_1, r_1)$ and $G_2 = (V_2, E_2, r_2)$ be two rooted graphs and let $\sim \subseteq V_1 \times V_2$ be an order or equivalence relation. A relation $\mathcal{S} \subseteq V_1 \times V_2$ is a *rooted simulation* of G_1 in G_2 with respect to $\sim$ if:

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

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

Definition 2.2 does not preclude that two distinct vertices v_1 and v_1' of G_1 are simulated by the same vertex v_2 of G_2, i.e. $v_1 \mathcal{S} v_2$ and $v_1' \mathcal{S} v_2$. Figure 2.8 gives examples of simulations with respect to the equality of vertex adornments. The simulation of the right example is not minimal.

The *existance* of a simulation relation between two graphs (without variables) can be computed efficiently: results presented in [67] give rise to the assumption that such problems can generally be solved in polynomial time and space. However, computation of pattern matching usually requires to compute not only one, but all minimal simulations between two graphs, in which case the complexity increases with the size of the "answer".

Interestingly, graph simulation can also be used for schema validation (cf. e.g. [4]). In this case, a schema is considered as a graph in which all instances have to simulate. This suggests that schema validation and querying are closely related: schema validation can be considered as querying the schema with a semistructured expression. If the query succeeds, the expression is an instance of the schema (i.e. valid). If the query fails, the expression is no instance of the schema (i.e. invalid).

CHAPTER

THREE

Web Query Languages

As we have seen in Chapter 2, XML is increasingly used not only as a format for representing text documents, but also as a format for representing semistructured databases and for exchange of data on the Web. As such, it becomes more and more important to be able to *query* XML data. Obviously, query languages for XML need to respect the peculiarities of the data and thus differ from traditional query languages. Likewise, a query language for the *Web* needs not only to be capable of querying XML data, it also needs to be able to perform network operations, and — following the *Reasoning Capabilities* design principle of Section 1.3.8 — support reasoning mechanisms for the Semantic Web.

This Chapter first argues why Web query languages need to provide a higher expressive power than traditional database query languages (Section 3.1). It then continues with an overview of desirable characteristics of Web query languages following [73] (Section 3.2). Finally, existing Web query languages are summarised (Section 3.3), with a focus on the predominant languages *XPath*, *XSLT* and *XQuery*.

3.1 Database vs. Web Query Languages

Traditionally, access to a database management system is realised using a query language (the so-called *data manipulation language*) embedded in a so-called *host language* (which can be any programming language available on a system, e.g. *Java* or *C*). In this setup, the query language only has limited expressive power, whereas more complex computations are performed in the host language [100]. For example, in relational database systems, query languages are usually *relationally complete* (i.e. they support all of the operations of the relational algebra, like *projection*, *selection* and *joins*), but exclude recursion and thus do not provide the same expressive power as general purpose programming languages.

Example 3.1
The original versions of SQL, e.g. did not allow to compute the transitive closure of a relation (this functionality has later been added to SQL'99 [6], but is not part of the core standard). Consider e.g. a binary relation `uncle` that relates nephews with their (immediate) uncles:

uncle	*nephew*	*uncle*
	Donald Duck	Scrooge Duck
	Huey Duck	Donald Duck
	Dewey Duck	Donald Duck
	Louie Duck	Donald Duck

Note that the (transitive) `uncle` relationship between e.g. Dewey Duck and Scrooge is not directly represented in the table. Query languages like SQL (earlier than SQL3) are in general (i.e. if the number of transitive steps is not known in advance) not capable of retrieving this information. In contrast, more expressive languages like *Datalog* are capable of doing this by using recursion. The following recursive Datalog query describes this transitive closure:

```
uncle(X,Y) :- uncle(X,Z), uncle(Z,Y).
```

This restriction is deliberate as it allows for many automatic optimisations that are much more difficult or even not possible in more expressive languages. However, since many applications need to perform more powerful query tasks while at the same time making use of the advantages of database management systems (like efficient storage and access, concurrency, etc.), database management systems have been combined with more expressive languages (usually languages providing the full expressiveness of first order logic, or at least the expressiveness of the *Horn*-fragment of first order logic). Such systems are often referred to as *knowledge base systems* [100]. According to [100], "applications require a knowledge-base system if they have a recursive or nested structure that needs to be queried" (p.983).

Since XML documents and semistructured databases often comprise such nested structures, a query language for such data consequently needs to be more expressive than traditional database query languages like SQL. Consider for example an XML document containing a large text (e.g. a book or this thesis) structured in chapters and sections (for an example, cf. Section 2.4.3). A typical query could be to retrieve all sections (at arbitrary level of nesting) where the title contains the substring "XML". Since this query needs to consider sections at an depth unknown by the query author, such queries cannot be expressed in languages that do not support recursion. In general, restructuring of graph structured data also requires languages with a higher expressive power than is available in traditional query languages ([4], p.54).

Also, embedding a Web query language into a host language is often not feasible: queries might be exchanged between different Web sites and processed in a distributed manner (e.g. either on the client, or on the server, or on both); relying on a host language would require that all participating Web sites are able to evaluate the query language as well as the host language. Consequently, a Web query language needs to be *self-contained* (cf. characteristics 3 in Section 3.2 below).

When querying on the *Semantic Web* (cf. Section 1.3.8), a higher expressiveness is even more important. Reasoners like FaCT [58] or RACER [55] need the expressive power of the description logic $\mathcal{SHIQ}$ [59], but the Semantic Web is still in development and more powerful reasoners are conceivable. To support arbitrary Semantic Web reasoners, it is thus desirable to provide query languages that have the same power as general purpose programming languages.

3.2 Desirable Characteristics of Web Query Languages

From Section 3.1, it is already possible to see that Web query languages need to be different from traditional database query languages in terms of expressive power. This section introduces further characteristics that have been deemed desirable for Web query languages. In the article *Database Desiderata for an XML Query Language* [73], David Maier summarises 13 such characteristics for XML query languages (David Maier uses the term *XQuery* to refer to any XML query language; at the time of publication, the language now called XQuery did not exist). The following is a quote from Section 2 of this article. References have been adapted to point to the correct items and emphasis has been added to improve readability.

1. **XML Output**

   An XQuery should yield XML output. Such a closure property has many benefits. Derived databases (views) can be defined via a single XQuery. Query composition and decomposition is aided. It is transparent to applications whether they are looking at base data or a query result.

2. **Server-side Processing**

   XQuery should be suitable for server-side processing. Thus, an XQuery should be self-contained, and not dependent on resources in its creation context for evaluation. While an XQuery might incorporate variables from a local context, there should be a "bound" form of the XQuery that can be remotely executed without needing to communicate with its point of origin. An example of local content that should be "bound away" is the local alias for a namespace. (See Requirement 11.)

3. **Query Operations**

Selection, extraction, reduction, restructuring and combination should all be possible in a single XQuery. This requirement is a consequence of Requirement 2, really. It should not be necessary to resort to another language or multiple XQueries to perform these operations. One reason is that an XQuery server might not understand the other language, necessitating moving fragments of the desired result back to the sender for final processing. Some of these operations greatly reduce data volumes, so are highly desirable to perform on the server side to reduce network requirements. Further, efficient query optimization and evaluation depends on having as much data access and manipulation described in advance as possible, to plan the best data retrieval, movement and processing strategies.

What I mean my these different operations, briefly:

- *Selection*: Choosing a document or document element based on content, structure or attributes.
- *Extraction*: Pulling out particular elements of a document.
- *Reduction*: Removing selected sub-elements of an element.
- *Restructuring*: Constructing a new set of element instances to hold queried data.
- *Combination*: Merging two or more elements into one.

[. . .]

4. **No Schema Required**

   XQuery should be usable on XML data when there is no schema (DTD) known in advance. XML data is structurally self-describing, and it should be possible to an XQuery to rely on such "just-in-time" schema information in its evaluation. This capability means XQueries can be used against an XML source with limited knowledge of its documents' precise structures.

5. **Exploit Available Schema**

   Conversely, when DTDs are available for a data source, it should be possible to judge whether an XQuery is correctly formed relative to the DTDs, and to calculate a DTD for the output. This capability can detect errors at compile time rather than run time, and allows a simpler interface for applications to manipulate a query result.

6. **Preserve Order and Association**

   XQueries should preserve order and association of elements in XML data. The order of elements in an XML document can contain important information – a query shouldn't lose that information. Similarly, the grouping of sub-elements within elements is usually significant. For example, if an XQuery extracts `<title>` and `<author>` sub-elements from `<book>` elements in a bibliographic data source, it should preserve the `<title>`-`<author>` associations.

7. **Programmatic Manipulation**

   XQueries should be amenable to creation and manipulation by programs. Most queries will not be written directly by users or programmers. Rather, they will be constructed through user-interfaces or tools in application development environments.

8. **XML Representation**

   An XQuery should be representable in XML. While there may be more than one syntax for XQuery, one should be as XML data. (Note that XSL is written in XML.) This property means that there do not need to be special mechanisms to store and transport XQueries, beyond what is required for XML itself. It also helps satisfy Requirement 7.

9. **Mutually Embedding with XML**

   XQueries should be mutually embedding with XML. That is, an XQuery should be able to contain arbitrary XML data, and an XML document should be able to hold arbitrary XQueries. The latter capability allows XML document to contain both stored and virtual data. The former capability allows an XQuery to hold arbitrary constants, and allows for

partial evaluation of XQueries. Representation of arbitrary constants helps with Requirement 2. Partial evaluation is useful in a distributed environment where data selected at one source is sent to another source and combined with data there.

10. **XLink and XPointer Cognizant**
    XQuery should provide for following XLinks and XPointers. One expects much XML will contain external and internal cross-references, which a query should be able to traverse.
11. **Namespace Alias Independence**
    An XQuery should not be dependent on namespace aliases local to an XML document. An XQuery of course needs to disambiguate elements names that occur in more than one DTD for a document. However, it is unreasonable to expect the query creator to know the internal aliases that a document uses for those DTDs. Also, if a query is issued over a group of documents, they may well use different aliases for the same DTD.
12. **Support for New Datatypes**
    XQuery should have an extension mechanism for conditions and operations specific to a particular datatypes. I am thinking mainly of specialized operations for selecting different kinds of multimedia content.
13. **Suitable for Metadata**
    XQuery should be useful as a part of metadata descriptions. For example, a metadata interchange format for data warehousing transformations or business rules might have components that are queries. It would good if XQuery could be used in such cases, rather than defining an additional query language. Another possible metadata use would be in conjunction with XMI for expressing data model constraints. An implication is that queries should be able to stand alone, and not have to be appended to a URL or URI.

3.3 Existing Web Query Languages

3.3.1 XPath

XPath, the *XML Path Language* [108], is a selection language aiming at addressing parts of an XML document. As it lacks capabilities for restructuring data items, it cannot be considered a true query language, it is rather a *selection language*. However, many other query languages are based on XPath, in particular the two most prominent query languages XSLT and XQuery, which are presented below.

Data Model: Ordered Tree

XPath models an XML document as an *ordered tree*. XPath differentiates between several kinds of nodes, including *document nodes*, *element nodes*, *attribute nodes* and *text nodes*. This document tree induces the so-called document order, which is obtained by traversing the document tree in a depth-first, left-to-right manner. XPath does not consider non-tree graph structures like semistructured expressions, and ID/IDREF are only supported by explicit dereferencing.

Navigation Steps

An XPath expression specifies a sequence of *navigation* or *location steps* (separated by and beginning with "`/`") in this tree, similar to what a car navigation system might provide to locate a certain address. For example, to select the phone number of `Mickey Mouse` in the address book used in Chapter 2, an XPath expression would specify to start at the document node, proceed to the element node `address-book`, from there move to each of the children, and for each child to the name to determine whether the name is `Mickey Mouse`. In this case, it would select in the next step the child node with label `phone`:

```
/child::address-book/child::person[
    child::name[child::first = "Mickey" and child::last = "Mouse"]]/child:phone
```

axis	description
`/`	select the document root (which is considered the parent of the document element)
`ancestor`	proper ancestor of current node
`ancestor-or-self`	current node or proper ancestor of current node
`attribute`	attribute of current node
`child`	immediate descendant (child) of current node
`descendant`	proper descendant of current node
`descendant-or-self`	current node or proper descendant of current node
`following`	node following the current node in document oder
`following-sibling`	node following the current node in document oder and at the same depth as the current node
`preceding`	node preceding the current node in document oder
`preceding-sibling`	node preceding the current node in document oder and at the same depth as the current node
`namespace`	namespace node of the current node
`parent`	immediate ancestor (parent) of current node
`self`	current node

Table 3.1: Axis Specifications available in XPath

The result of such a selection is always a sequence of nodes. XPath does not differentiate between a single value and the sequence consisting only of that value. This has serious implications, for instance, the `=` operator is not true equality but only "existential" equality, i.e. it tests whether the intersection of two sequences is non-empty.

Axis Specifications

The navigation steps in XPath expressions contain so-called *axis specifications* that specify the "direction" of the traversal in the document tree. In the example above, the only axis specifier used was `child`. Other frequently used axis specifiers are `descendant`, which selects not only immediate child nodes but also child nodes of child nodes and so forth, and `following-siblings` selects all siblings that come after the currently selected node in document order. Axis specifications are separated from node tests by `::`. Table 3.1 summarises the axis specifications available in XPath.

An XPath expression beginning with a forward slash (i.e. `/`) always specifies a traversal anchored at the root, and is thus called an *absolute* XPath expression. An XPath expression beginning with any other axis specifications is *relative* to the current *context node*.

Node Tests

Navigation steps consist of *node tests* that specify what kinds of nodes to select. XPath supports, among others, the following node tests:

`name`	matches elements of type *`name`*
`*`	matches every element
`namespace:name`	matches elements of type *`name`* from the given namespace
`namespace:*`	matches every element from the given namespace
`comment()`	matches comment nodes
`text()`	matches text nodes
`node()`	matches every node

The most common form of node test is to specify the element name, as in the example above.

Predicates

Predicates express further conditions on node tests that go beyond the capabilities of simple matching. For example, they may be used to select every second element node, or all `person` element nodes that contain a child node with label `first` and further text child `Mickey`, together with a child node with label `last` and further text child `Mouse`. Predicates are enclosed in square brackets `[ ]` and follow the node test (or other predicates). Predicates may contain:

location path	the predicate succeeds if the evaluation of the location path returns a non-empty sequence
exp OP *exp*	compares two expressions, which may either be atomic values, location paths or function calls, with OP. The following comparison operators are supported: • `=` tests whether the intersection of two sequences is non-empty • `!=` tests whether the intersection of two sequences is empty • `>`, `>=`, `<` and `<=` convert the two expressions to numbers and compare them accordingly
pred `and` *pred*	connects two predicates with `and`
pred `or` *pred*	connects two predicates with `or`

Abbreviated Syntax

Those axis specifications that are most frequently used (e.g. `child` and `descendant-or-self`) can also be expressed using an *abbreviated syntax*, which closely resembles path specifications for directories and files in UNIX. The following table summarises the available abbreviations:

Expression	Abbreviation
`child::`*name*	*name*
`/descendant-or-self::`*name*	`//`*name*
`self::node()`	`.`
`parent::node()`	`..`
`attribute::`*name*	`@`*name*
`[position()=`*n*`]`	`[`*n*`]`

All other axes have no counterpart in the abbreviated syntax, but it is possible to mix abbreviated and non-abbreviated syntax as required.

In the abbreviated syntax, the selection of the phone number of “Mickey Mouse” is more conveniently expressed as:

```
/address-book/person[name[first = "Mickey" and last = "Mouse"]]/phone
```

3.3.2 XSL/XSLT

The *Extensible Stylesheet Language*[1] includes both a transformation language (called *XSL Transformations* or XSLT) and a formatting language (called *XSL Formatting Objects* or XSL-FO). Both are specified as XML applications (i.e. they use an XML syntax) and are developed by the *World Wide Web Consortium* (W3C), where they have achieved the status of *W3C Recommendation*, which in W3C terms is the equivalent of a standard.

Whereas XSL-FO is merely a language containing instructions for *formatting* documents (similar to HTML), XSLT [106] can be considered a query language, as it allows to select data from an XML document and rearrange it in a new structure. XSLT is currently mainly used to transform XML documents into HTML, but other applications exist.

[1] `http://www.w3.org/Style/XSL/`

XML-based Syntax

XSLT *stylesheets* (i.e. query or transformation programs) are themselves XML documents. An XSLT stylesheet can be seen as a *form* or *template* for the resulting XML document, augmented by data selection expressions. This is advantageous for two reasons:

- it allows to easily embed queries in documents that contain mostly static content, similar to other Web languages like PHP or JSP.
- it allows to treat XSLT stylesheets as data, i.e. XSLT stylesheets can be input as well as output of another XSLT stylesheet.

XSLT stylesheets have the following XML structure:

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  template rules

</xsl:stylesheet>
```

Template Rules

An XSLT stylesheet is always given in terms of a sequence of *template rules*. A template rule consists of an XPath expression (called the *pattern*) functioning as a *guard* to the rule (i.e. a condition that specifies when the rule is applicable), and an XML fragment used as a *template* for the output that may contain either fixed markup, XPath expressions for data selection, or recursive applications of template rules.

Example: Recall the address book example used earlier. The following XSLT stylesheet creates an HTML document summarising the entries of the address book in a table.

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/address-book">
    <html>
      <head><title>Address Book</title></head>
      <body>
        <table>
          <tr><td>Name</td><td>Phone</td><td>Email</td></tr>
          <xsl:apply-templates select="./person"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="person">
    <tr>
      <td><xsl:value-of select="./name/first"/>
          <xsl:value-of select="./name/last"/></td>
      <td><xsl:value-of select="./phone"/></td>
      <td><xsl:value-of select="./email"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

The first template rule matches the root element (specified by `/`) if it has the label `address-book`. It contains a template that creates some static HTML markup, and recursively applies the stylesheet to all `person` child elements relative to the current element (i.e. the root element). The second template rule matches only elements that have a label of `person`, but at arbitrary depth. It constructs a table row and fills it with values relative to the current element.

Structural Recursion

The fundamental computation model of XSLT is structural recursion over the structure of a single, fixed input document, beginning with the root element and traversing to the leaves. In the course of this traversal, template rules are applied in case their patterns match and content is immediately written to the result document. Since rules have to adhere to this traversal, they in general cannot serve to structure a stylesheet into logical components. As a consequence, it is neither possible to query more than one document nor to query the results of other rule applications for further processing within a single stylesheet.

However, the author of a stylesheet can deviate from the root-to-leaves traversal by specifying explicit selections in recursive calls to `apply-templates` or by using absolute or backward XPath expressions in a rule pattern or selection. The following template rule lists for each person the names of person that he/she knows. It first selects the value of the attribute `knows` into the variable `knows`, then outputs the person name, then applies the stylesheet to all names of persons that this person knows. The latter are selected beginning again at the root element, and consequently, this selection deviates from the default traversal of the tree.

```
<xsl:template match="person">
  <xsl:variable name="knows" select="@knows"/>
  <p>
    Person Name: <xsl:value-of select="name/first"/>
                 <xsl:value-of select="name/last"/> <br/>
    Knows: <xsl:apply-templates select="//person[@oid = $knows]/name"/>
  </p>
</xsl:template>
```

Imperative Constructs

Besides the recursive application of template rules, XSLT provides a set of imperative constructs that may be used inside of template rules:

- `xsl:for-each` may be used to iterate over all elements of a sequence selected by an XPath expression
- `xsl:if` may be used to output certain parts only if some condition succeeds (no `else`)
- `xsl:choose` is a generalisation of `xsl:if` that allows to specify several alternatives guarded by conditions and works like `case` or `switch` in other programming languages

Example: The example in the previous paragraph listed all known person, but the output does not contain commas to separate the different names. The following refinement adds commas after a name in case it is not the last name of the list. `xsl:for-each` is used to iterate over the sequence of person names that are known, and an `xsl:if` inserts a comma in case the name is not the last element in the sequence:

```
<xsl:template match="person">
  <xsl:variable name="knows" select="@knows"/>
  <p>
    Person Name: <xsl:value-of select="name/first"/>
                 <xsl:value-of select="name/last"/> <br/>
    Knows: <xsl:for-each select="//person[@oid = $knows]/name"/>
             <xsl:value-of select="first"/>
```

```
            <xsl:value-of select="last"/>
            <xsl:if test="position() = last()">,</xsl:if>
          </xsl:for-each>
  </p>
</xsl:template>
```

Named Templates

XSLT allows to add names to template rules by using a `name` attribute, in which case the template rule is also called a *named template*. Named templates can then be called explicitly (as opposed to the implicit `xsl:apply-templates`) by `xsl:call-template`, similar to function calls in other programming languages. Such calls can take an arbitrary number of named parameters (using `xsl:with-param`) and can also be recursive. This feature provides XSLT with the capability to express any kind of computation [66].

Example: The following XSLT template recursively calculates the faculty of the parameter `n`. The parameter `akk` is an accumulator, which is necessary because all construction of a template is immediately written to the output and cannot be bound to variables. If `n` is greater than 1, then the template is called recursively, with `n` decreased by 1 and the accumulator multiplied by `n`. Otherwise, the accumulator is returned. Both parameters are assigned a default value of `1`.

```
<xsl:template name="fac">

  <xsl:param name="n" select="1"/>
  <xsl:param name="akk" select="1"/>

  <xsl:choose>
    <xsl:when test="$n > 1">
      <xsl:call-template name="fac">
        <xsl:with-param name="n" select="$n - 1"/>
        <xsl:with-param name="akk" select="$n * $akk"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$akk"/>
    </xsl:otherwise>
  </xsl:choose>

</xsl:template>
```

Due to its verbose XML syntax, XSLT programs often appear more complicated than they are. Also, the built-in recursion over the tree structure of the input document – while being very powerful – often confuses users, in particular beginners.

3.3.3 XQuery

XQuery [113] is an XML query language developed by the *XML Query Working Group*[2] at W3C. It is currently a *W3C Working Draft*, but scheduled to become a *W3C Recommendation* (i.e. the W3C equivalent of a standard) soon. The *XML Query Working Group* is a committee with participants from both, academia and industry, and the design of the language is influenced by many different groups, which sometimes gives the impression that XQuery tries to solve all problems at once (a phenomenon often referred to as "design-by-committee"[3])

The following introduction is inspired by the recently published book *XQuery from the Experts* [65], the first chapter of which is also available online[4], and by the lecture notes found at [27].

[2] `http://www.w3.org/XML/Query`
[3] `http://c2.com/cgi/wiki?DesignByCommittee`
[4] `http://www.datadirect.com/techzone/xml/xquery/docs/katz_c01.pdf`

Data Model

Like XSLT, XQuery regards every XML document as an *ordered tree*, consisting of (among others) *element nodes*, *attribute nodes*, and *text nodes*, with a particular node called the *root node*, which is parent of the node corresponding to the outermost element and represents the whole document.

Every *value* in XQuery is a sequence of nodes or atomic values, there is no distinction between a single element and the sequence consisting only of this element. A sequence is always ordered, usually (if not explicitly sorted differently) in so-called *document order*, i.e. in order of appearance in the XML document. Atomic values can be typed using a type system that is similar to XML schema's basic types. Also, the document itself can be associated with a schema definition.

Path Expressions

XQuery uses XPath for selecting nodes in this tree. Any XPath expression is itself an XQuery program. The following XQuery program thus retrieves all authors in the bibliography example of Section 2.4.2 (abbreviated XPath syntax):

```
/bib/book/authors/author
```

The result of an XQuery program is in general not a tree like in XSLT but instead a *forest*, i.e. a sequence of trees. In the example above, the result would be:

```
<author>
  <last>Ingelman-Sundberg</last>
  <first>Catharina</first>
</author>

<author>
  <last>Ingelman-Sundberg</last>
  <first>Catharina</first>
</author>

<author>
  <last>Wahl</last>
  <first>Mats</first>
</author>

<author>
  <last>Nordqvist</last>
  <first>Sven</first>
</author>

<author>
  <last>Ambrosiani</last>
  <first>Björn</first>
</author>
```

Constructing Nodes

Similar to XSLT, queries can be embedded in construction patterns that resemble the intended result. A very simple XQuery program is thus just an XML document without any XQuery specific expressions:

```
<result>
  Hello World!
</result>
```

When embedding XQuery expressions in construction patterns, they are enclosed in curly braces { }. Anything not enclosed in curly braces is written to the output and not evaluated. The following XQuery program groups the forest created above under an `authors` element:

```
<authors>
  { /bib/book/authors/author }
</authors>
```

Variables

An expression of the form $*name* is called a *variable reference*. Variables may be bound to *values*, i.e. sequences of subtrees selected by XPath expressions (see `FOR` and `LET` below). Variable references may be used in XPath expressions where they are substituted by their binding. If the value (a sequence!) of the binding contains more than one item, then each of these items is substituted in turn, building a union of all selected data items.

Example: Assume, the variable `$b` is bound to a sequence of `book` subtrees in the bibliography. The following XQuery expression creates a sequence of authors for the books contained in `$b`:

```
$b/authors/author
```

Variables may not only occur at the beginning of an expression but in certain cases also within it. The following example assumes that the variable `$b` is bound to the (sequence containing only the) string `book` (in contrast to the example above, where `$b` was bound to the sequence of subtrees with label `book`). This expression again creates a sequence of authors:

```
/bib/$b/authors/author
```

FLWOR Expressions

Variables are bound in so-called FLWOR (read "flower") expressions. FLWOR is an abbreviation composed of the initial letters of the five fundamental XQuery keywords `FOR ... LET ... WHERE ... ORDER BY ... RETURN`.

`FOR` and `LET` serve to bind variables in different manners. Whereas `FOR` iterates over all items in a sequence and binds a variable successively to each item (or rather to the singleton sequence containing only that item), `LET` binds the variable once to the complete sequence.

Example: The following expression binds the variable `$b` successively to each book in the bibliography (selected by the XPath expression `/bib/book`):

```
FOR $b IN /bib/book
```

whereas the following expression binds the variable `$b` to the sequence of all books:

```
LET $b := /bib/book
```

`FOR` loops may also iterate over several sequences, which effectively is a shortcut for nested loops and allows to compute e.g. the cross-product of two sequences. For instance, the following expression builds the cross-product of all books and all authors:

```
FOR $b IN /bib/book,
    $a IN /bib//author
```

`WHERE` allows to attach conditions to filter the admissible bindings of a variable in a `FOR` expression. For instance, the following expression binds the variable `$b` only to books published after 1997:

```
FOR $b IN /bib/book
WHERE $b/@year > 1997
```

ORDER BY specifies an ordering for the variable bindings in a `FOR` expression. `ORDER BY` is followed by an XPath expression selecting the nodes based on which the sequence should be ordered. The following XQuery expression successively binds the variable `$b` to all books of the bibliography sorted by title:

```
FOR $b IN /bib/book
ORDER BY $b/title
```

RETURN returns the result of an XQuery expression. The result is again specified by an XQuery (sub-)expression, i.e. it may contain construction templates, XPath expressions, or nested FLWOR expressions. For instance, the following XQuery expression (example XMP-Q3 from the *XQuery Use Cases* [34]) lists for each book in the bibliography the title and authors, grouped inside a `result` element:

```
<results>
{
    FOR $b IN doc("file:bib.xml")/bib/book
    RETURN
        <result>
            { $b/title }
            { $b/author  }
        </result>
}
</results>
```

FLWOR expressions can also be nested. The following XQuery expression (example XMP-Q4 from the *XQuery Use Cases*) lists for each author in the bibliography the author's name and the titles of all books by that author, grouped inside a `result` element, i.e. the symmetric case for the example above. It also uses a function called `distinct-values` that eliminates duplicates in a sequence:

```
<results>
{
  LET $a := doc("file:bib.xml")//author
  FOR $last IN distinct-values($a/last),
      $first IN distinct-values($a[last=$last]/first)
  ORDER BY $last, $first
  RETURN
    <result>
      <author>
        <last>{ $last }</last>
        <first>{ $first }</first>
      </author>
      {
        FOR $b IN doc("file:bib.xml")/bib/book
        WHERE some $ba IN $b/author
              satisfies ($ba/last = $last and $ba/first=$first)
        RETURN $b/title
      }
    </result>
}
</results>
```

The two previous examples also show a major deficiency of XQuery: although both examples query exactly the same data, the second query expression is much more complicated, as it has to arrange the data in a new structure, whereas in the first example the result quite similar in structure to the original document.

The Positional Variable `at`

Iterations with `FOR` also allow to simultaneously bind a variable to the position of the current item in the sequence. This is achieved by the construct `at`. The following XQuery expression selects books together with their position in the bibliography, and outputs this in an appropriate XML representation:

```
<results>
{
  FOR $b at $p IN /bib/book
  RETURN
    <result>
      <position>{ $p }</position>
      <title>{ $b/title }</title>
    </result>
}
</results>
```

Selecting the position can be important as the position often conveys meaning. For example, in an HTML table, the position is the only way to refer to a certain column or row.

Joins

It is often useful to combine data from different sources. To this aim, XQuery supports not only to bind multiple variables in a `FOR` loop (which computes the cross product), but also to join them based on certain join conditions. Join conditions are added to the `WHERE` clause of an XQuery expression, like any other condition. For example, the following query expression selects books from both databases of Section 2.4.2 to combine them in a unified representation:

```
<books>
{
  FOR $b IN doc("file:bib.xml")/bib/book,
      $r IN doc("file:reviews.xml")/reviews/entry
  WHERE $b/title = $r/title
  RETURN
    <book>
      { $b/title   }
      { $b/authors }
      { $r/review  }
    </book>
}
</books>
```

Using nesting of query expressions, XQuery is capable of expressing a wide range of different join conditions, like equijoin, left outer join, etc.

Quantifiers

Sometimes it is necessary to determine whether *all* items or *at least one* item in a sequence satisfy a certain condition. This is achieved using the quantifiers *all* and *some* in a `WHERE` clause. For instance, the following XQuery expression selects only such books where one of the authors is “Sven Nordqvist”:

```
FOR $b IN doc("file:bib.xml")/bib/book
WHERE some $a in $b/authors/author
      satisfies ($a/last = "Nordqvist" and $a/first = "Sven")
RETURN $b
```

Conditional Expressions: if . . . then . . . else . . .

XQuery uses `if ...then ...else ...` expressions similar to conditional expressions in other languages. Both the `then` and the `else` branch are required, but it is possible to use `()` as return value to denote empty content. The following XQuery expression lists for each book having at least one author the title and the first two authors, and adds an empty `et-al` element in case there are more than two authors (example XMP-Q6 of [34]):

```
<bib>
  {
    for $b in doc("file:bib.xml")//book
    where count($b/author) > 0
    return
        <book>
            { $b/title }
            {
                for $a in $b/author[position()<=2]
                return $a
            }
            {
                if (count($b/author) > 2)
                 then <et-al/>
                 else ()
            }
        </book>
  }
</bib>
```

Operators and Functions

XQuery provides a variety of pre-defined operators and functions that can be used in XQuery expressions. These include arithmetic operators (like `+`, `-`, or `*`), comparison operators (like `=`, `!=`, or `>`), sequence operators (like `union`, `intersect`, or `except`), and a function library containing functions for many tasks that occur frequently (like date/time conversion, string search, etc.).

User-Defined Functions

It is also possible to define own functions that can be used to reuse frequently used query expressions, or to define recursive traversals of a data structure. Function definitions in XQuery have the following form (*`fname`* is the function name, `$`*`param1`*`, $`*`param2`*`, ...` is a list of parameters with names `param1`, `param2`, etc):

```
define function fname ($param1, $param2, ...)
  as element()*
{
  XQuery expression
}
```

The expression `as element()*` indicates that the return type of the function is an arbitrary number of elements (see types below).

3.3.4 Survey over other Web Query Languages

The remainder of this section gives a brief survey over other XML query languages that are relevant to this thesis because they have properties that are interesting for the development of the language Xcerpt. Besides

the languages listed below, there exist many other XML query languages that are not discussed here, e.g. *XQL* [90] and *XirQL* [52] (both are similar to XPath), *Quilt* [35] (the main predecessor of XQuery), *Lorel* [2] (which is an extension of the Object Query Language OQL to semistructured data), *fxt* [17] (the *Functional XML Transformer*, similar to XSLT), *FnQuery* [96] (adds XPath-like constructs for querying XML to Datalog and/or Prolog), *XPathLog/LoPiX* [75] and *X-DEVICE* [10] (both are deductive path-based query languages for XML), *Elog* [11] and *CXQuery* [38] (the *Constraint XML Query Language*), are extension of Datalog with XPath expressions, *WebLog* [68], *XML-RL* [70] (the *XML Rule Language*), and *XET/XDD* [7] (*XML Equivalent Transformations*). In addition to these pure query languages, there are furthermore the languages *XDuce* [32] and *CDuce* [13], which extend the functional programming language *ML* by constructs for XML processing, and the library *HaXml* [121], which adds XML processing support to the functional programming language *Haskell*.

All of these languages are *textual* languages aimed at querying XML or semistructured data. Besides this, a number of *visual* query languages exist (e.g. *XML-GL* [33], *Xing* [47], *Complete Answer Aggregates* [76], *BBQ* [78], *QURSED* [84], *VXT* [86], and *Lixto* [12]) that allow to compose query programs using a visual interface. Since visual querying is not the focus of this thesis, none of them are described here in detail. Furthermore, a number of query languages aiming at querying Semantic Web data are proposed (e.g. *RDQL* [95], *OWL-QL* [48], and *TRIPLE* [99]), which are not capable of querying plain XML data and thus also not discussed here.

UnQL

UnQL [31] (the *Unstructured Query Language*) is a query language originally developed for querying semistructured data and precedes the development of XML. It has later been adopted to querying XML, but the origins are still apparent in many language properties (for example, UnQL has a non-XML syntax that is very similar to the syntax of OEM presented in Section 2.1.3). UnQL uses *query patterns* and *construction patterns* and a query consists of a single `select ...where ...` rule that separates construction from querying. Queries may be nested, in which case the separation of querying and construction is abandoned.

Example 3.2 (UnQL)
Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element. Note the use of nested `select ...where ...` statements to group titles and authors.

```
select { results: (
  select { result: { title: T,
                     ( select { author: A }
                       where { author: A } in Book )
                   }
         }
  where  { book: Book } in Bib,
         { year: Y, title: T } in Book ) },
         Y > 1991
where { bib: Bib } in db
```

Since UnQL has originally not been developed for the Web, it is apparently not possible to address arbitrary Web documents. Instead, the example above uses an identifier called `db` to refer to the semistructured database represented by the document `bib.xml`.

Pattern-Based Querying and Simulation. UnQL is to the best of our knowledge the first language to propose a pattern-based querying (albeit with subqueries instead of rule chaining) for semistructured data (including XML). It furthermore uses *graph simulation* as its foundation for evaluation, which inspired the usage of simulation for the evaluation of Xcerpt.

XML-QL

XML-QL [45] is a pattern-based, rule-based query language for XML designed at AT&T Labs. Like UnQL, it uses *query patterns* initiated by `WHERE` and augmented by variables for selecting data and *construction patterns* initiated by `CONSTRUCT` for reassembling selected data in new structures. An XML-QL query always consists of a single `WHERE-CONSTRUCT` rule, which may be divided into several subqueries. The following example gives a flavour of how XML-QL queries are constructed using nested subqueries. Like in XQuery and XSLT, expressions beginning with `$` are variables. Note also that XML-QL uses tag-minimisation to abbreviate closing tags.

Example 3.3 (XML-QL)
Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element. Like in UnQL, subqueries are used to group titles and authors.

```
WHERE
  <bib>
    $book
  </> IN "bib.xml"
CONSTRUCT <results>
            WHERE <book year=$y>
                    <title>$t</>
                    <author>$a</>
                  </book> IN $book, y > 1991
            CONSTRUCT <result>
                        <title>$t</>
                        WHERE $a2 IN $a
                        CONSTRUCT <author>$a2</>
                      </result>
          </results>
```

Logic Variables. One of the main characteristics of XML-QL is that it uses query patterns containing multiple variables that may select several data items at a time instead of path selections that may only select one data item at a time. Furthermore, variables are similar to the variables of logic programming, i.e. "joins" can be evaluated over variable name equality. Since XML-QL does not allow to use more than one separate rule, it is often necessary to employ subqueries to perform complex queries.

XMAS

XMAS [72], (the *XML Matching And Structuring language*), is an XML query language that builds upon XML-QL. Like XML-QL, XMAS uses *query patterns* and *construction patterns*, and rules of the form `CONSTRUCT ...WHERE ....` However, XMAS extends XML-QL in that it provides a powerful *grouping construct* instead of relying on subqueries for grouping data items within an element. It furthermore supports *pattern restrictions* that allow to restrict the admissible bindings of a variable.

Example 3.4 (XMAS)
Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element. Note that XMAS provides special grouping constructs for grouping titles and authors and thus avoids nested subqueries.

```
CONSTRUCT
  <results>
    <result>
      $T
      $A {$A}
    </result> {$T}
  <results>
```

```
WHERE
  <bib>
    <book year=$Y>
      $T: <title/>
      $A: <author/>
    </>
  </> IN "bib.xml"
  AND $Y > 1991
```

Grouping Constructs. In any kind of tree or graph structured databases, it is desirable to *group* data items beneath a node in the construction of the result. For example, when constructing the list of authors with all titles in the example used in this Section, it is necessary to group all `result` elements that can be created for different authors beneath the `results` element. Likewise, it is necessary to group all book titles of an author beneath the `result` element corresponding to that author.

Many query languages (like XQuery, UnQL, and XML-QL) implement grouping by reverting to *subqueries* that are embedded in the construction pattern, which leads to a close intertwining of query and construction patterns (as shown in the examples for XQuery, UnQL, and XML-QL above). In contrast, languages like XMAS (and Xcerpt) provide high-level *grouping constructs* that allow to collect data items that are bound in a separate query pattern. Such grouping constructs usually specify a set of variables on whose bindings the grouping is performed. A data item is created for each different combination of bindings for these variables.

In XMAS, grouping is expressed by enclosing the variables on whose bindings the grouping is performed in curly braces and attaching them to the end of the subpattern that specifies the structure of the resulting instances. In Example 3.4 above, a `result` element is created for every instance of `$T` (indicated by `{$T}` after the closing tag of the element `result`). Within every such result element, all authors are collected (indicated by `{$A}`).

When comparing this XMAS query with the XML-QL query above, it is easy to see that grouping constructs that result in a separation of querying and construction are a desirable property for a query language, as the query is more declarative and therefore easier to grasp.

Pattern Restrictions. A *pattern restriction* restricts the admissible bindings of a variable to such data items that match a certain query pattern. In XMAS, pattern restrictions may be attached to variables in the `WHERE` part of a rule and are denoted by `:`, followed by the restricting pattern. In Example 3.4 above, the variables `$T` and `$A` are restricted to certain patterns.

Arguably, a pattern restriction is a very declarative means to specify restrictions for variables. In languages that do not support pattern restrictions, it is necessary to add the restriction by using additional external constraints that are not part of the query pattern, and thus break up the query pattern. As a consequence, queries in such languages are not only less concise, but often also less efficient to evaluate.

Part II

The Language Xcerpt

CHAPTER

FOUR

Xcerpt: A New Programming Paradigm for Querying the Web

This Chapter introduces the syntactical constructs of the language Xcerpt and gives an intuitive meaning to them based on many examples without immediately providing the formal definition. Similar descriptions have also been published in [23, 93]. More extensive examples can be found in the next chapter.

This Chapter is structured as follows: Section 4.1 describes two different syntaxes that are available for Xcerpt programs, one based on XML and the other using a more compact term syntax. Section 4.2 describes *data terms*, which are Xcerpt's means to represent semistructured data and closely resemble the semistructured expressions of Section 2.1. Section 4.3 introduces *query terms*, which are patterns used to select subterms from data terms by binding variables. They resemble data terms, but provide advanced constructs for querying. Section 4.4 formally describes matching of query with data terms and introduces the notion of *ground query term simulation*, which is central to this thesis and replaces the syntactical equivalence used in most standard pattern matching approaches. Next, Section 4.5 describes how query terms can be combined to form more complex *queries*. Section 4.6 then introduces *construct terms*, which serve to reassemble variable bindings gathered in queries into new structures. Of particular interest here are the powerful grouping constructs *all* and *some* that serve to create nested lists of subterms. Finally, Section 4.7 introduces *construct-query rules* that combine queries with construct terms.

4.1 Two Syntaxes

Xcerpt uses two different syntaxes for programs and semistructured data, an XML syntax and a compact term syntax (called the Xcerpt syntax). The XML syntax allows to use standard XML tools like parsers, editors or browsers (in particular, the visXcerpt prototype [14, 16, 15] is based on a rendering of the XML syntax in Mozilla). The Xcerpt syntax, which is used in most parts of this thesis, is a more compact representation of Xcerpt programs. It is more convenient for both presentation and editing of Xcerpt programs. Furthermore, to emphasise that XML is not the only representation format for semistructured data (see Section 2.1 above), the Xcerpt syntax also provides a more generic format with a deliberate abstraction from XML. The XML syntax of Xcerpt is not presented in this thesis. Regular updates are available at `http://www.xcerpt.org`.

Besides such "convenience features", the Xcerpt syntax allows to express language constructs that have no direct counterpart in XML and thus can only be represented in the XML syntax by using Xcerpt-specific attributes and elements:

- *graph-structured data:* Semistructured data is in general graph structured (cf. Section 2.1). While several linking and reference mechanisms for XML exist (e.g. XLink [110] and ID/IDREF [116]), they privilege hierarchical data, as all of them need explicit dereferencing.

  In contrast, references in Xcerpt terms are treated as equal to a parent-child relationship when matching a query pattern against a "database".

- *unordered/ordered content:* In XML documents, content is always considered as being *ordered* (the so-called *document order*). In many applications, particularly in semistructured databases, it is however desirable to be able to consider data as *unordered*, i.e. the order in which data items occur is irrelevant.

  Xcerpt allows to mix both ordered and unordered content.

- *query specific constructs:* As Xcerpt is a pattern-based language, it is necessary to enrich term patterns with certain query-specific constructs like variables or partial/total and ordered/unordered term specification (see *Query Terms* below), but nonetheless stay as close as possible to the representation of data items.

4.2 Data Terms: An Abstraction for Data on the Web

Data terms represent XML documents and data items in semistructured databases. Data terms correspond to *ground* functional programming expressions and *ground* logical atoms. Syntactically, they are very similar to the semistructured expressions introduced in 2.1, but they contain additional constructs that allow to represent peculiarities of XML (like attributes). Apart from the special constructs for ordered/unordered term specification and the Xcerpt reference mechanism, data terms are thus just a simplified syntax for XML, or "XML in disguise". Data terms are not restricted to representing XML data or semistructured expressions: they are meant as an abstraction of many of the available formalisms for rooted, graph structured data like data represented in OEM or ACeDB, but also Lisp S-expressions or RDF graphs.

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

Like in the grammar of Section 2.1, expressions between < and > are non-terminal symbols (or variables). Expressions enclosed in the quotation characters " or ' are terminal symbols. `oid` and `label` denote object identifiers and expression labels (tag names), respectively. `oid`, `label`, and `string` are character sequences corresponding to XML identifiers, tag names, and text content. `number` is an arbitrary integer or floating point number. `iri` is an *internationalised resource identifier* as defined in [61]. In this thesis, the symbol ^ is often replaced by the more concise symbol ↑, which is unfortunately not available in ASCII.

If a data term t is of the form $\texttt{label}[\texttt{t}_1,\ldots,\texttt{t}_n]$ or $\texttt{label}\{\texttt{t}_1,\ldots,\texttt{t}_n\}$, then the $\texttt{t}_i$ are called *immediate subterms of* t. Subterms of the $\texttt{t}_i$ are called *indirect subterms* of t. If neither "immediate" nor "indirect" is specified, the term *subterm* usually only refers to the immediate subterms of a term. In analogy to the XML terminology, t is the *parent term* of its subterms, (immediate) subterms are sometimes also referred to as *child terms*, and the topmost parent term is called the *root term*. In an ¡attributes¿expression of the form $\texttt{attributes}\{\texttt{label}_1\{\ldots\},\ldots,\texttt{label}_n\{\ldots\}\}$, the labels must be different, because XML attributes need to have different names.

Example 4.1
Consider again the publication list from Section 2.1. The representation of this semistructured data item as a data term (or semistructured expression) is shown on the left. An equivalent representation (except subterm ordering) as an XML document is shown on the right. Note that the document prologue is omitted for brevity.

```
publications {
  book {
    title [ "Folket i Birka på Vikingarnas Tid" ],
    authors [
      author [ "Mats Wahl" ],
      author [ "Sven Nordqvist" ]
      author [ "Björn Ambrosiani" ]
    ]
  },
  book {
    title [ "Boken Om Vikingarna" ],
    authors [
      author [ "Catharina Ingelman-Sundberg" ]
    ]
  }
}
```

```
<publications>
  <book>
    <title>Folket i Birka på Vikingarnas Tid</title>
    <authors>
      <author>Mats Wahl</author>
      <author>Sven Nordqvist</author>
      <author>Björn Ambrosiani</author>
    </authors>
  </book>
  <book>
    <title>Boken Om Vikingarna</title>
    <authors>
      <author>Catharina Ingelman-Sundberg</author>
    </authors>
  </book>
</publications>
```

In this example, the terms with label `book` are *immediate subterms* or *child terms* of the term with label `publications`, which is also the *root term*. The term with label `publications` is thus the *parent term* of the terms with label `book`. The terms labelled `author` are immediate subterms of the respective terms labelled `authors`, and *indirect subterms* of e.g. the respective terms labelled `book`.

Data terms may be used as an abstraction for many other formalisms that represent hierarchical or graph structured data. The following two examples show the publication list as a *Lisp S-expression* and in the *Object Exchange Model* (OEM).

```
(publications
  (book
    (title "Folket i Birka på Vikingarnas Tid")
    (authors
      (author "Mats Wahl")
      (author "Sven Nordqvist")
      (author "Björn Ambrosiani")
    )
  )
  (book
    (title "Boken Om Vikingarna")
    (authors
      (author "Catharina Ingelman-Sundberg")
    )
  )
)
```

```
{ publications:
  { book:
     { title: "Folket i Birka på Vikingarnas Tid",
       authors:
         { author: "Mats Wahl",
           author: "Sven Nordqvist",
           author: "Björn Ambrosiani"
         },
     },
    book:
      { title: "Boken Om Vikingarna",
        authors:
          { author: "Catharina Ingelman-Sundberg" }
      }
  }
}
```

4.2.1 Term Specifications

Like semistructured expressions, data terms allow the specification of *ordered* and *unordered* lists of subterms. These properties are expressed by using different kinds of braces to parenthesise the subterms.

- Square brackets (i.e. [ ]) denote *ordered term specification*, i.e. the order of subterms in the list is significant. An ordered term specification allows to select subterms by position and is important e.g. in text documents.
- Curly braces (i.e. { }) denote *unordered term specification*, i.e. the order of subterms in the term is insignificant, although they are stored in a particular sequence. An unordered term specification allows to rearrange subterms in the list e.g. for building an index for faster access, or for more efficient use of a storage system (like grouping several small subterms in a single page of background memory while storing large subterms in an individual page each). Unordered term specification is commonly found in semistructured databases.

In Example 4.1 above, the term with label `publications` has an unordered term specification, meaning that the order of the `book` subterms is irrelevant, i.e. the storage system might choose to rearrange them in a different order. The terms with label `authors` have ordered term specification, meaning that the order of the list of `author` elements is significant (e.g. for proper citing).

Terms with different term specifications may be nested (i.e. subterms of a term may have a term specification different from the parent term's), but nesting of term specifications within the same list of subterms is not permitted. For example, the term `f{g["a","b"],h{"c","d"}}` is a data term, but `f{"a",["b","c"],"d"}` is not.

4.2.2 References

References are used for representing graph structures in a textual syntax. In Xcerpt data terms, subterms of the form `oid @ t` (read: "oid at t") are *defining occurrences* of `oid` and associate the identifier `oid` with the subterm `t`. Subterms of the form `^oid` (or `↑oid`, read: "reference to oid") are *referring occurrences* of `oid` and refer to the subterm associated with the identifier `oid`. As with semistructured expressions, every identifier may occur at most once in a defining occurrence, and an identifier used in a referring occurrence must also occur in a defining occurrence somewhere.

References in data terms are a unified representation for the various linking mechanisms available for XML (and other formalisms), like ID/IDREF, XPointer, XLink and URIs, and serve to simplify their representation in Xcerpt.[1] Unlike other query languages, Xcerpt automatically dereferences such references when querying, i.e. a reference can be treated like a parent-child relationship.

Example 4.2
The following two terms are considered to be equivalent:

```
f {                                  f {
  b { &o1 @ d {} },                    b { ↑&o1 },
  c { ↑&o1 }                           c { &o1 @ d {} }
}                                    }
```

4.2.3 Attributes

Unlike XML, Xcerpt does not have a special representation for attributes. Instead, XML attributes are treated as subterms of a term with the specific restriction that the value may not be structured content. An attribute of the form `key = "value"` is represented in Xcerpt as a term of the form `key{"value"}`

In order to separate attributes from child elements and thus retain the possibility to perform one-to-one transformations between Xcerpt and XML, Xcerpt groups them in a special subterm with the label `attributes`. Since attributes in XML are always unordered, this special subterm always has an unordered term specification (see above). As a convention, every data term should contain *at most one* `attributes` subterm, and this subterm, if existent, should be the *first* subterm in the list of subterms (even in case the parent term is unordered). Also, all attributes of a term need to have different labels.

Example 4.3
Each book in the `bib.xml` database of Section 2.4.2 contains an attribute `year` in the XML syntax. Consider for example the following book:

```
<book year="1995">
  <title>Vikinga Blot</title>
  <authors>
    <author>
      <last>Ingelman-Sundberg</last>
      <first>Catharina</first>
    </author>
  </authors>
  <publisher>Richters</publisher>
  <price>5.95</price>
</book>
```

In Xcerpt syntax, this book can be represented as follows. Note in particular that the element itself is ordered (as it is a representation of an XML document) while the attributes are unordered:

[1] Note that Xcerpt is not limited to its own reference mechanism: e.g. ID/IDREF can easily be dereferenced using an appropriate query (cf. Section 5.1.2).

```
book [
  attributes { year { "1995" } },
  title [ "Vikinga Blot" ],
  authors [
    author [
      last [ "Ingelman-Sundberg" ],
      first [ "Catharina" ]
    }
  ],
  publisher [ "Richters" ],
  price [ "5.95" ]
]
```

This treatment of attributes has the main advantage that no exceptions are needed in the definition of Xcerpt extensions like variables or regular expressions. Instead, since attributes are represented in the same term structure as elements, it is possible to use the standard constructs for all occurrences of attributes.

4.2.4 Namespaces

Xcerpt supports namespaces in a straightforward manner that follows closely the use of namespaces in XML (cf. Section 2.2.6). Like in XML, namespaces are URIs (*uniform resource identifiers*) or IRIs (*internationalised resource identifiers*). Namespace prefixes can be declared and are then separated from term labels by a colon. As an extension to XML namespaces, it is also possible to use the namespace URI as a prefix [2].

Namespace Declarations

Namespace prefixes are declared with the keyword `ns-prefix` followed by the defined prefix, a `=` and the namespace IRI. The default namespace (i.e. the namespace of all subterms that do not have an explicit namespace prefix) can be defined with the keyword `ns-default`, followed by `=` and the namespace IRI of the default namespace.

```
1 <ns-declaration> ::= "ns-prefix" <ns-prefix> "=" '"' iri '"'
2                    | "ns-default" "=" '"' iri '"'    .
```

As a simplification over XML namespaces, this thesis allows namespace declarations only outside terms. This restriction obviously anticipates nested namespace declarations and shadowing, and thus a *syntactic* one-to-one mapping between XML documents and Xcerpt terms preserving the namespace prefixes is not always possible, although the two approaches have equivalent expressiveness (both allow to associate namespace IRIs with term/element labels). Transforming XML documents that use nested namespace declarations into data terms and vice versa is nevertheless possible as the *namespaces* themselves are preserved and just the *namespace prefixes* might get lost. Further refinements of namespaces that take into account both nested declarations and shadowing are currently being investigated.

Namespaces in Data Terms

In Xcerpt terms, namespaces are used almost as in XML. The most significant difference to XML is that the namespace IRI may also be used as a namespace prefix. In this case, it is not necessary to define the namespace in advance.

[2] In XML, this is not admissible due to syntactic restrictions. Xcerpt does not need to adhere to such restrictions as it is not necessary to retain backwards compatibility with applications that are not namespace aware.

```
1 <ns-prefix> = label | '"' uri '"' .
```

Example 4.4 (Namespaces in Xcerpt)
Consider again Example 2.14 on page 30, which illustrated the use of namespaces in XML by adding a `remarks` element to address book entries that might contain HTML elements for markup. It uses the namespace prefix `a` to refer to the address book schema, and the namespace prefix `b` to refer to the XHTML schema. As a data term, this document might be represented as follows:

```
ns-prefix a = "http://www.myschemas.org/address-book"
ns-prefix b = "http://www.w3.org/2002/06/xhtml2"

a:address-book {
  &o1 @ a:person {
    a:name {
      a:first { "Mickey" },
      a:last { "Mouse" }
    },
    a:phone {
     attributes {
       a:type { "home" }
     },
     "19281118"
    },
    a:knows { ↑&o2 },
    a:remarks {
      b:strong{"Note:"}, "The phone number is also the", b:em{"birthday"},"!"
    }
  },
  &o2 @ a:person {
    a:name {
      a:first { "Donald" },
      a:last { "Duck" }
    }
  }
}
```

Instead of declaring the namespace prefix `b`, it would also be possible to use the namespace URI directly, as in the following example. Note also the use of the default namespace declaration.

```
ns-default = "http://www.myschemas.org/address-book"

address-book {
  &o1 @ person {
    name {
      first { "Mickey" },
      last { "Mouse" }
    },
    phone {
     attributes {
       type { "home" }
     },
     "19281118"
```

```
    },
    knows { ↑&o2 },
    remarks {
      "http://www.w3.org/2002/06/xhtml2":strong{"Note:"},
      "The phone number is also the",
      "http://www.w3.org/2002/06/xhtml2":em{"birthday"},"!"
    }
  },
  &o2 @ person {
    name {
      first { "Donald" },
      last { "Duck" }
    }
  }
}
```

4.3 Query Terms: Patterns for Selecting Data

Query terms are (possibly incomplete) patterns matched against Web resources represented by data terms. A pattern is like a *form* augmented by variables acting as place holders for data retrieved from data terms (cf. Section 1.3.3), very similar to (non-ground) atoms in logic programming. Query terms build upon data terms, but may contain *variables*, constructs for expressing *incompleteness* (cf. Section 1.3.4), as well as *position specifications*, *subterm negation*, and subterm exclusion.

4.3.1 Incompleteness

As discussed in Section 1.3.4, query patterns need to support incomplete query specifications, because data represented on the Web has a much more flexible schema than data represented e.g. in relational databases. Query terms may contain constructs for expressing incompleteness in *breadth*, in *depth*, with respect to *order*, and with respect to *optional subterms*. The terms "breadth" and "depth" refer to the graph induced by a data term or semistructured expression (cf. Sections 2.5 and 4.4.1). Note that the constructs described here together realise requirement 4 ("no schema required") of David Maier's database desiderata (cf. Section 3.2).

Incompleteness in Breadth: Partial Term Specifications

Incompleteness in breadth (i.e. within the subterms of the same parent term) is expressed by using so-called *partial* and *total term specifications*:

- double square or curly braces (i.e. `[[ ]]` or `{{ }}`) denote *partial term specifications*, i.e. a data term matched by the query term may contain additional subterms not matched by subterms of the query term.
- single square or curly braces (i.e. `[ ]` or `{ }` as in data terms) denote *total term specifications*, i.e. a data term matched by the query term must not contain additional subterms that are not matched by subterms of the query term.

Consequently, a data term that is used as a query term matches only itself (and all such terms that are equivalent with respect to subterm ordering in case of unordered term specifications), whereas a query term containing partial term specifications matches possibly infinitely many data terms. As with ordered/unordered term specifications, subterms with different term specifications may be nested, but nesting within the same list of subterms is disallowed.

Example 4.5 (Total/Partial Term Specifications)
Consider the `bib.xml` document of the bookstore example from Section 2.4.2. The following two are query terms for this database:

```
bib {
  book {
    title { "Boken Om Vikingarna" }
  }
}
```

This query term does not match with the data term, as its total term specification requires that there is exactly one book with exactly one `title` element.

```
bib {{
  book {{
    title {{ "Boken Om Vikingarna" }}
  }}
}}
```

This query term will match with the data term, as it allows for additional books and additional elements inside the `book` element.

Incompleteness wrt. Order: Unordered Term Specifications

Like data terms, query terms may contain both *ordered term specifications* (square brackets `[ ]` and `[[ ]]`), and *unordered term specifications* (curly braces `{ }` and `{{ }}`). Let t_1 be a query term and let t_2 be a data term:

- if t_1 has an ordered term specification, then it matches with t_2 only if t_2 also has an ordered term specification. Furthermore, all subterms of t_1 must match subterms in t_2 in the same order of appearance.
- if t_2 has an unordered term specification, then it matches with t_2, if t_2 has either an ordered term specification or an unordered term specification. All subterms of t_1 must match subterms in t_2 in arbitrary order.

In case a query term uses ordered and partial term specification, the matched data term has to contain corresponding subterms in the same order as the subterms of the query term, but there may be additional subterms in between.

Example 4.6 (Ordered/Unordered Term Specifications)
Consider the `bib.xml` example of Section 2.4.2. Recall that in this example the list of authors for each book uses an *ordered* term specification. The following two query terms show the difference between ordered and unordered term specifications in query terms:

```
bib {{
  book {{
    authors [[
      author {
        first [ "Björn" ],
        last [ "Ambrosiani" ]  },
      author {
        first [ "Sven" ],
        last [ "Nordqvist"]    }
    ]]
  }}
}}
```

Match with all books where the author “Björn Ambrosiani” appears before the author “Sven Nordqvist”.
This query term does not match with the data term, as the authors in the database do not have the same order as in the query term.

```
bib {{
  book {{
    authors {{
      author {
        first [ "Björn" ],
        last [ "Ambrosiani" ]        },
      author {
        first [ "Sven"],
        last [ "Nordqvist"]    }
    }}
  }}
}}
```

Match with all books that have (at least) the two authors “Björn Ambrosiani” and “Sven Nordqvist” in any order.
This query term will match with the database, as the query term does not enforce a particular order on authors.

Incompleteness in Depth: Descendant

Incompleteness in depth is expressed using the `descendant` construct. A query term of the form `desc t` (read: "descendant t") matches with all data terms that contain a subterm that is matched by `t` at an arbitrary depth (including zero). It is the counterpart to the Kleene star operator of regular path expressions and to XPath's descendant (in short notation: `//`) construct (cf. Section 3.3.1).

Example 4.7 (Descendant)
The following query term matches with a text document (like the one introduced in Section 2.4.3), if at arbitrary depth below the root term, the data term representing the text document contains a `section` term with a `title` subterm containing the string "Data Terms", i.e. either a section, a subsection, a sub-subsection, etc.

```
report {{
 desc section {{
    title {{ "Data Terms" }},
  }}
}}
```

Currently, the descendant construct is unrestricted, i.e. it "matches" with any path. Extensions are being considered that allow restrictions to these paths, e.g. using regular expressions over labels, or sets of admissible term labels.

Incompleteness wrt. Optional Subterms: Optional

Terms containing a subterm of the form `optional t` specify to match the subterm `t` with a subterm of the data term if possible (and yield variable bindings for the variables in `t` accordingly); otherwise, the evaluation of the query does not fail, but does not yield any bindings for the variables in `t`.

Example 4.8
Consider in the following the student database example introduced in Section 2.4.1. The following query term retrieves student names (variable `Name`) and student ids (variable `MatrNr`). If both exist, both are returned. If only the name exists, the evaluation does not fail (i.e. the query term still matches), but binds only the variable `Name`. If there is no name in the data term, the query term fails to match it.

```
students {{
  student {{
    name { var Name },
    optional matrnr { var MatrNr }
  }}
}}
```

The construct `optional` is not strictly necessary as the same queries can be expressed by using several query terms instead of only one. However, it is a convenient construct in many practical examples of semistructured databases and XML documents, as the schema languages of such formats often allow optional elements.

4.3.2 Term Variables, Label/Namespace Variables and the →-Construct

Variables act as "handles" for those subterms of the data term that match with the subterm the variable is "attached to". If a query term matches with a data term, the variables are bound to the corresponding subterms. They can thus be used to retrieve data from a data term and assemble it in a new structure (with the help of construct terms, Section 4.6 below). As in logic programming, a single variable can occur at

several positions in a term. Of course, bindings to such variables have to be consistent for all occurrences, i.e. all occurrences of the same variable must have the same binding.

Matching a query term with a data term yields a set of alternative substitutions, each of which represents a possible binding for the variables in the query term such that the resulting ground instance matches with the data term (see Section 4.4 below). Obviously, the use of unordered and partial term specifications allows several alternative bindings for the variables that all fulfill this requirement.

In Xcerpt query terms, the following variable notions are used:

- *Variables without restriction* are expressed using the keyword `var` followed by an identifier (variable name). They can be bound to any subterm in the data term and are thus very similar to the variables in logic programming, i.e. they act as place holders.

- *Variables with restriction* are expressed like a variable without restriction followed by the symbol `->` or → (read "as") and a query term. They can only be bound to subterms of the data term that match with the pattern they are restricted to. Note that variable restrictions are also used in the language XMAS (cf. Section 3.3.4).

- *Label Variables* are, like variables without restrictions, expressed by using the keyword `var` followed by an identifier, but they occur at the position of a label in a query term. They can be bound to any label of a subterm of the data term that matches with the remaining term specification.

- *Namespace Variables* are similar to label variables. They occur at the position of a namespace prefix in a query term. Namespace variables are always bound to the namespace URI/IRI, not to the namespace prefix.

Note that in logic programming, variable restrictions are represented using external constraints. The advantage of constraining a variable to certain subterms *within* a query term instead of *outside* the query is to better convey the overall structure of the considered query. Arguably, restricting variables inside the query term more appropriately realises the concept of *query patterns*.

Example 4.9 (Substitutions)
In the student database (Section 2.4.1), the query term given on the left hand side matches the variable `Name` with the student name and variable `Email` with the email address. The right hand side lists different substitutions that yield ground instances of the query term that match with the data term given in Figure 2.3 on page 34.

```
students {{
  student {{
    name  {{ var Name }}
    email {{ var Email }}
  }}
}}
```

Substitution σ_1:
```
Name   Donald Duck
Email  donald@duck.org
```

Substitution σ_2:
```
Name   Mickey Mouse
Email  mickey@mouse.org
```

Substitution σ_3:
```
Name   Goofy
Email  goofy@goofy.org
```

Substitution σ_4:
```
Name   Goofy
Email  goofy@disney.com
```

Note in particular that `Goofy` is listed twice as the data term contains two possible email addresses that can be bound to the variable `Email`.

Example 4.10 (Pattern Restrictions)
The following query terms for the `bib.xml` database of Section 2.4.2 illustrate the difference between variables without and with pattern restrictions.

```
bib {{
  book {{
    var X,
    authors {{ var AUTHOR }}
  }}
}}
```

In this query term, the occurrence of the variable X is unrestricted. Thus, the variable X might be bound to any subterm of the book element (besides authors), e.g. to price or title, since the variable X occurs without restriction.

```
bib {{
  book {{
    var X → title {{ }},
    authors {{ var AUTHOR }}
  }}
}}
```

In this query term, the occurrence of the variable X is restricted to such subterms that are matched by the query term title {{ }}. Thus, the variable X can only be bound to the title element.

The use of the keyword var to introduce a variable is not strictly necessary. It is often possible to determine from the context whether a term is a variable or not. In particular, extensions of the Xcerpt syntax are investigated that allow to declare variables in a context block. However, using the keyword var simplifies the syntax in particular for the programmer, as it allows to easily identify variables without having to look at the context.

Label variables are useful to retrieve structural information that is unknown in advance, e.g. when transforming an XML document into an HTML representation displaying the structure of the XML document (as e.g. in the implementation of the visual language *visXcerpt* [14, 16, 15]).

Example 4.11 (Label Variables)
Consider the student database of figure 2.3. The following query term retrieves the label of the element containing the string "Goofy" in the variable X:

```
students {{
  student {{
    var X {{ "Goofy" }},
  }}
}}
```

4.3.3 Position Specification and Positional Variables

In some applications it is desirable to query subterms only at a certain position while still being able to use partial query patterns, or to query the position of subterms in the database. For example, a query to a report in XML format could select the second paragraph of all sections.

In query terms, subterms of the form position X t denote that the query term only matches with data terms that have at position X a subterm t′ that is matched by t. The position specification X is either a positive number, or a negative number, or a variable:

- a *positive number* specifies the position of the matched subterm below its parent term, where 1 is the position of the first subterm
- a *negative number* specifies the position relative to the last subterm of the parent, where -1 is the position of the last subterm.
- a *variable* matches with a subterm with any position and binds the variable to the position of this subterm as a positive integer number.

Position specification is admissible in all kinds of term specifications, i.e. ordered and unordered as well as total and partial query terms. Note, however, that it is possible to express patterns that are contradictory and thus impossible to match, as position specifications might conflict with ordered or total term specifications (e.g. f[[position 2 a, position 1 b]] or f{position 2 a}).

Note that a term containing a subterm with position specification can never match against a data term with unordered term specification, as in such cases there is no information available about the position of elements.

Example 4.12 (Position Specification)
Consider an HTML document containing a table with books and prices, like the following:

```
<table>
  <th>
    <td>No.</td>
    <td>Title</td>
    <td>Price</td>
  </th>
  <tr>
    <td>3675</td>
    <td>Vikinga Blot</td>
    <td>5.95</td>
  </tr>
  <tr>
    <td>6743</td>
    <td>Boken Om Vikingarna</td>
    <td>22.95</td>
  </tr>
</table>
```

```
table [
  th [
    td ["No."],
    td ["Title"],
    td ["Price"]
  ],
  tr [
    td ["3675"],
    td ["Vikinga Blot"],
    td ["5.95"]
  ],
  tr [
    td ["6743"],
    td ["Boken Om Vikingarna"],
    td ["22.95"]
  ]
]
```

Now suppose you want to select the titles and prices of books in this HTML table. Since there is no possibility to determine this via subterm labels, it is necessary to explicitly specify the position in the selection, as in the following query term:

```
table {{
  td {{
    position 2 td { var Title },
    position 3 td { var Price }
  }}
}}
```

A solution that is even more flexible takes advantage of the column labels in the table headings and uses variables in the position specification to select the positions of the columns with label “Title” and “Price”. The same variables are then used in place of the positions 2 and 3 of the example above.

```
table {{
  th {{
    position var TPos td { "Title" },
    position var PPos td { "Price" }
  }},
  td {{
    position var TPos td { var Title },
    position var PPos td { var Price }
  }}
}}
```

Note that this query term does not assume that the price column comes after the title column!

4.3.4 Subterm Negation: without

Subterms of the form `without t` denote so-called *subterm negation*. Subterm negation allows to express that a data term should *not* contain subterms matching a certain query pattern. It is only applicable to subterms and may not be used at the root level. Furthermore, subterm negation is only reasonable in partial term specifications, and order does not have influence on the negated subterms (only on all positive subterms).

This kind of negation is useful in semistructured data, as the schema of such data often allows to omit subterms. For example, a query might ask for "all students that did not submit their homework" (i.e. all student elements that do not contain an element indicating that they submitted their homework). Note that in relational database systems, this negation is very similar to querying for NULL values.

Example 4.13
Recall the student database from Section 2.4.1. The following query term retrieves students that did not submit exercise 2 in the variable S.

```
students {{
  var S → student {{
    without exercise {{ number { 2 } }}
  }}
}}
```

The query matches if there is at least one student that does not have an `exercise` element with number 2.

Subterms negated by `without` may contain variables, but such an occurrence can never yield variable bindings, i.e. it is not possible to retrieve all subterms that do *not* occur. Accordingly, all variables that occur in the scope of a `without` have to appear elsewhere outside the scope of a negation construct (cf. Section 6.2). Nonetheless, using variables in negated subterms can be useful, as shown in the following example.

Example 4.14
Given a text document like the PhD thesis described in Section 2.4.3. The following query term uses subterm negation to retrieve all references to citations that have no corresponding entry in the bibliography in the variable `Citation` (note the representation of attributes in the Xcerpt term notation):

```
report {{
  desc var Citation → cite {
    attributes { ref { var Ref } }
  },
  desc bibliography {{
    without entry {{
      attributes {{ id { var Ref } }}
    }}
  }}
}}
END
```

As there is no bibliography entry with an `id` of `rdf`, the result of evaluating this rule against the sample document of page 35 is:

```
var Citation =  cite { attributes { ref{"rdf"} } }
```

Subterm negation is an *existential* negation: As long as there exists at least one term which does not contain the negated subterm (and matches with the remainder of the pattern), all other terms are irrelevant. There might be terms that contain the negated subterm, those simply do not match. Note that although subterm negation might appear less expressive than full negation as failure, it does in fact share the same problems if it occurs in combination with the `all` construct introduced below.

4.3.5 Regular Expressions

Query terms provide advanced text processing capabilities using *regular expressions* (abbreviated: *RE*). In language theory, a regular expression is a means to define a regular language (see e.g. [57]) and matches with a character sequence, if the character sequence is a word of the language defined by the regular expression. In Xcerpt, regular expressions may be used either in place of strings or in place of subterm labels, and take the form

`/<regexp>/`

where `<regexp>` is a regular expression based on the syntax defined in *POSIX* [60] (*Portable Operating System Interface*) with some Xcerpt-specific extensions (see below). POSIX regular expressions are very widespread (they are e.g. used in the languages *Perl*, *Python*, and *Java*) and thus well known to many programmers.

Example 4.15 (Regular Expressions)
The following query term against a text document (like the document described in Section 2.4.3) selects all sections that contain the substring "XML" in their title, i.e. where an arbitrary number of characters appears before and after a substring "XML" (expressed by `.*`).

```
report {{
  desc var S → section {{
    title {{ /.*XML.*/ }}
  }}
}}
```

POSIX Regular Expressions

As POSIX regular expressions are very well-known, this thesis only provides a brief summary over the major constructs used for building regular expressions. The language definition is available at [60], and many introductory books into programming languages provide a thorough treatment of the topic (see e.g. [49]).

In POSIX, an underlying character set is assumed (e.g. Unicode). Valid characters are all characters of the character set, where an ordinary (i.e. not special) character usually matches only itself, and the special character `.` matches all characters.

Special Characters Special characters are not matched. The following special characters are used in POSIX regular expressions:

character(s)	description
`*`	arbitrary repetition (0–) of the preceding character or subexpression
`+`	arbitrary repetition, but at least one (1–) of the preceding character or subexpression
`?`	optional occurrence (0–1) of the preceding character or subexpression
`|`	separates alternatives
`{n}`	exactly *n* occurrences of the preceding character or subexpression
`{n,m}`	between *n* and *m* occurences of the preceding character or subexpression
`^`	anchor (beginning of line)
`$`	anchor (end of line)
`(` and `)`	enclose subexpressions (see below)
`[` and `]`	define character classes (see below)
`\`	quote special characters

If a special character is to match instead of being interpreted, it has to be quoted using the prefix symbol `\`. For instance, `\.` matches the point and `\+` matches the plus character.

Character Classes Square brackets are used to define character classes, i.e. sets of characters. A character class matches with all characters that are part of the class. The following character classes are predefined by POSIX:

```
[:alnum:]   [:cntrl:]   [:lower:]   [:space:]
[:alpha:]   [:digit:]   [:print:]   [:upper:]
[:blank:]   [:graph:]   [:punct:]   [:xdigit:]
```

For example, the character class `[:alnum:]` contains all alphanumeric characters, `[:upper:]` contains all upper case characters, and `[:blank:]` contains all whitespace characters of the character set.

It is also possible to define new character classes by enclosing all matched characters in square brackets. The character class `[abc]` for instance matches with the characters a, b, and c. Character classes may contain range expressions using the hyphen character, like in `[a-zA-Z]` (matching all Latin lower/upper case letters), where the range is defined depending on the underlying character set (e.g. ASCII or Unicode).

Character classes are negated if the first character after the opening bracket is `^`. For instance, `[^-+]` denotes all characters except - and +. Note the different meaning of `^` in regular expressions as negation of character classes and as beginning of line anchor, and in Xcerpt terms as reference to an identifier.

Subexpressions. POSIX allows to specify subexpressions in regular expressions in order to retrieve specific parts from the matched text. A subexpression is enclosed in parentheses (and) (often also \(and \), e.g. in the search function of the editor *Emacs*). Subexpressions can later be referred to by their position. If subexpressions are nested, the position is determined by counting the opening parentheses.

Example 4.16 (POSIX Regular Expressions)
The following regular expression matches with date strings of the form "1999-12-23" (i.e. in ISO syntax) and retrieves the year, month and day in the subexpressions 1, 2 and 3.

```
([1-9][0-9]{3})-([01][0-9])-([0-3][0-9])
```

Backreferences. Backreferences are denoted by `\n`, where `n` is a single digit other than 0. A backreference matches a literal copy of whatever was matched by the corresponding n'th subexpression of the pattern.[3]

The regular expression	matches e.g. the strings
`(.*)-\1`	`a-a`
	`go-go`
	`wiki-wiki`

Note that "regular expressions with backreferences" are (strictly speaking) not regular, they describe a subset of context free languages.

Xcerpt Extensions

In POSIX, the (substrings matched by) subexpressions are referred to by position after the matching is evaluated. This approach is well suited for imperative languages like *Perl* or *Java*, where the evaluation is sequential. For example, the following *Perl* program retrieves the year, month, and day from the string `"2004-03-04"` by referring to the positions of the subexpressions after the regular expression is matched. The overbraces highlight the respective subexpressions.

```
                                  1          2          3
if("2004-03-04" =~ /(([1-9][0-9]{3})-([01][0-9])-([0-3][0-9])/) {
  $year = $1;
  $month = $2;
  $day = $3;
}
```

[3] Note that matching with backreferences is NP-hard (`http://perl.plover.com/NPC/`), as it is possible to encode the 3-SAT problem in a regular expression with backreferences.

In Xcerpt, referring to subexpressions by position is not feasible: it is incompatible with pattern-matching as it requires a specific control flow and does not fit with Xcerpt's notion of variable binding. Instead, Xcerpt introduces variables (→ restrictions) into regular expression patterns, similar to the way variables are part of term patterns. With this extension, subexpressions can take the form

```
(var <name> →...)
```

where ... denotes the regular expression pattern and <name> is the name of the variable restricted by this pattern. When the regular expression is matched against a character sequence, the variable is bound to the part of the character sequence that is matched by the subexpression.[4] For example, the following query term binds the year, month and day of a date string to the variables Y, M and D:

```
/(var Y →[1-9][0-9]{3})-(var M →[01][0-9])-(var D →[0-3][0-9])/
```

Note that subexpressions of the form (...) are still possible as a means for structuring the expression, but the character sequences they matched with cannot be retrieved.

Example 4.17 (Variables in Regular Expressions)
The following query term retrieves student names and email addresses from the file `students.xml` and separates the local name (`User`) from the domain name (`Domain`) of the email addresses. The first subexpression binds everything from the beginning of the string (indicated by ^) up till the first appearance of `@` (indicated by the character class `[^@]` matching all characters except `@`) to the variable `User`. The second subexpression binds every alphanumeric character (including `.` and `-`) after the `@` to the variable `Domain`.

```
in { resource { "file:students.xml" },
  students {{
    student {{
      name { var Name},
      email {
        /^(var User →[^@]+)@(var Domain →[a-zA-Z0-9.-]+)/
      }
    }}
  }}
}
```

Such a separation could be useful for rendering email addresses on Web pages in a "spamvertised form", i.e. not easily recognisable by automatic email address harvesters: the variable bindings for `User` and `Domain` could be reassembled in a construct term (see below) with a suitable representation (e.g. separate `User` and `Domain` by <at>).

4.4 Query Evaluation: Ground Query Term Simulation

Matching query terms with data terms is based on the notion of *rooted graph simulations* introduced in Section 2.6. Intuitively, a query term matches with a data term, if there exists at least one substitution for the variables in the query term (called *answer substitution* of the query term) such that the corresponding graph induced by the resulting *ground* query term simulates in the graph induced by the data term. Of course, graph simulation needs to be modified to take into account the different term specifications, descendant construct, optional subterms, subterm negation, and regular expressions.

It might appear that it would suffice to restrict simulation to matching a ground query term with a data term instead of allowing to match two ground query terms; however, a relation on arbitrary combinations of ground query terms is useful as ground query term simulation is later used to define *simulation equivalence*

[4] Note that this approach is similar to the extensions of regular expressions in the language Python [104], where a group may have the form (?P<name>...). The substring matched by this group is later accessible via the symbolic name <name>.

and a (partial) ordering on the set of ground query terms . This ordering is used in the definition of answers below to ensure that a variable is always bound to the maximal possible value.

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

4.4.1 Ground Query Terms and Ground Query Term Graphs

Let $\mathcal{T}^q$ be the set of all query terms.

Definition 4.1 (Ground Query Term)

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

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

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

The *graph induced by a ground query term* (or short: *ground query term graph*) is defined in analogy to the graph induced by a semistructured expression (cf. Section 2.6) as follows.

Definition 4.2 (Graph Induced by a Ground Query Term)
Given a ground query term t. The *graph induced by* t is a tuple $G_t = (V,E,r)$, with:

1. a set of *vertices* (or *nodes*) V defined as the set of all (immediate and indirect) subterms of t (including t itself).

2. a set of *edges* $E \subseteq V \times V \times \mathbb{N}$ characterised as follows:

   - for all terms $t_1,t_2,t_3 \in V$: if t_2 is the subexpression of t_1 at position i and of the form `^oid` (a referring occurrence), and t_3 is of the form `oid @ t'` (a defining occurrence), with `oid` an identifier and `t'` a term ($\in V$), then $(t_1,t_3,i) \in E$.
   - for all terms $t_1,t_2 \in V$: if t_2 is the subexpression of t_1 at position i and *not* of the form `^oid`, then $(t_1,t_2,i) \in E$.

3. a distinguished vertex $r \in V$ called the *root node* with $r = t$.

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

Like for semistructured expressions in Section 2.6, representing vertices as complete subterms and edges with positions is necessary for the definition of the simulation relation, as it conveys information about ordered/unordered and partial/total term specifications, and the respective positions of subterms in a term. Figure 4.1 illustrates this definition on two ground query terms. Note that for space reasons, the vertices in both graphs do not contain the subterms, but only the term labels and specifications.

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

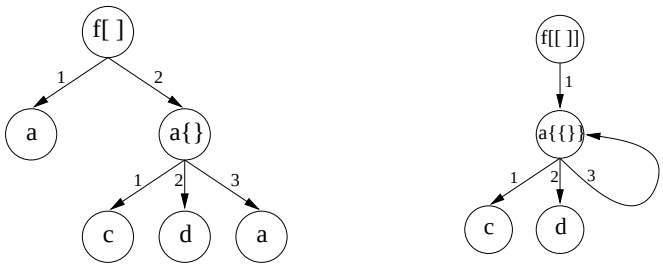

Figure 4.1: Graphs induced by $f[a,a[c,d,a]]$ and $f[[\&1 \ @ \ a\{\{c,d,\uparrow \&1\}\}]]$

4.4.2 Term Sequences and Successors

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

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

Definition 4.3 (Term Sequence)

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

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

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

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

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

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

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

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

Definition 4.4 (Sequence of Successors)
Let t be a ground query term, let $G_t = (V,E,t)$ be the graph induced by t, and let $v \in V$ be a node in G_t (i.e. subterm of t). The *sequence of successors* of v, denoted $Succ(v)$, is defined as

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

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

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

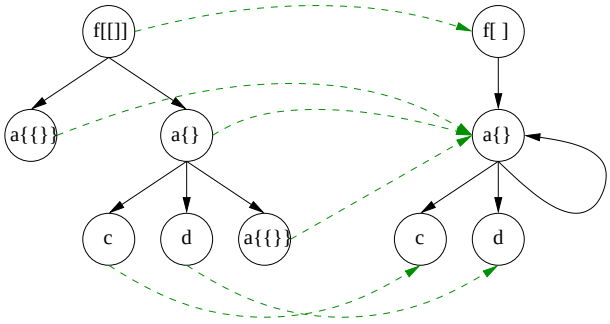

Figure 4.2: Minimal simulation of $f[[\ a\{\{\ \}\}, a\{\{c,d,a\{\{\ \}\}\ \}\}\ ]]$ in $f[\&1\ @\ a\{c,d,\uparrow \&1\}]$

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

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

In Chapter 7, the following additional notions of subsequences and concatenation of sequences are needed. Both definitions are straightforward. In order to distinguish subsequences from subsets, we usually write $S' \sqsubseteq S$.

Definition 4.5 (Subsequences, Concatenation of Sequences)
Let $S = \langle s_1,\ldots,s_m\rangle$ and $T = \langle t_1,\ldots,t_n\rangle$ be term sequences.

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

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

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

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

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

4.4.3 Ground Query Term Simulation

Using the graphs induced by ground query terms, the notion of rooted simulation almost immediately extends to all ground query terms: intuitively, there exists a simulation of a ground query term t_1 in a ground query term t_2 if the labels and the structure of (the graph induced by) t_1 can be found in (the graph induced by) t_2 (see Figure 4.2). So as to define an ordering on the set of all ground query terms, ground query term simulation is designed to be transitive and reflexive.

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

The definition of *ground query term simulation* is characterised using a mapping between the sequences of successors (i.e. subterms) of two ground terms with one or more of the following properties, depending

on the kinds of subterm specifications and occurrences of the constructs `without` and `optional`. Recall that a mapping is called total if it is defined on all elements of a set and partial if it is defined on some elements of a set.

Definition 4.6
Given two term sequences $M = \langle s_1, \ldots, s_m \rangle$ and $N = \langle t_1, \ldots, t_n \rangle$.

1. A partial or total mapping $\pi : M \to N$ is called
   - *index injective*, if for all $s_i, s_j \in M$ with $index(s_i) \neq index(s_j)$ holds that $index(\pi(s_i)) \neq index(\pi(s_j))$
   - *index monotonic*, if for all $s_i, s_j \in M$ with $index(s_i) < index(s_j)$ holds that $index(\pi(s_i)) < index(\pi(s_j))$
   - *index bijective*, if it is index injective and for all $t_k \in N$ exists an $s_i \in M$ such that $\pi(s_i) = t_k$.
   - *position respecting*, if for all $s_i \in M$ such that s_i is of the form `position` j s_i' holds that $index(\pi(s_i)) = j$
   - *position preserving*, if for all $s_i \in M$ such that s_i is of the form `position` j s_i' holds that $\pi(s_i)$ is of the form `position` l t_k' and $j = l$.
2. A partial mapping $\pi : M \to N$ is called *completable* with respect to some property P, if there exists a partial or total mapping $\pi' : M \to N$ such that
   - $\pi(s_i) = \pi'(s_i)$ for all $s_i \in M$ on which π is defined, and
   - there exists at least one term $s_j \in M$ on which π is undefined and π' is defined and
   - P holds for π'

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

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

A *position preserving* mapping maps a term with position specification to a term with the same position specification; it is applicable in case the sequence of successors of the second term N is incomplete with respect to order or breadth, as the exact position cannot be determined otherwise in these cases. In particular, this ensures the reflexivity and transitivity of the ground query term simulation (see Theorem 4.9 below). E.g. given the terms $f\{\{position\ 2\ b\}\}$ and $f\{a,b,position\ 2\ b\}$, the subterm $position\ 2\ b$ of the first term needs to be mapped to the subterm $position\ 2\ b$ of the second term, but cannot be mapped to the first b because its position is not "guaranteed".

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

The *completable* property is used for optional and negated terms. If a term has a negated successor, the mapping of its sequence of successors to the successors of a second term has to be defined on all successors that are not negated, but must not be completable to any of the negated successors. For example, given the terms $f\{\{a, without\ b\}\}$ and $f\{a,b,c\}$, all positive subterms of the first term can be mapped to subterms of the second term, but this mapping is completable to the negated subterm $without\ b$, causing the match to fail. For optional successors, the *completable* property is used to ensure that the simulation is maximal with respect to optional successors, i.e. all successors for which it is possible need to participate in the simulation.

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

Definition 4.7 (Label Match)
A term label l_1 matches with a term label l_2, if

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

l_1 does not match with l_2 in all other cases.

Example 4.18

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

The following definition characterising a ground query term simulation of a ground query term t_1 into a ground query term t_2 is divided into several parts. The first part (points 1 and 2) describes simulation for terms not containing the subterm negation `without`, which is rather straightforward. Subsequent parts extend the notion of simulation by introducing `without` first only into the term t_1 and then also into the term t_2 (points 4 and 5), and the last part (point 6) describes simulation in case of `optional` subterms. Both extensions are rather complex and therefore treated separately.

Let $G = (V,E,t)$ be the graph induced by a ground query term t. In the following, $Succ(t')$ denotes the sequence of all successors (i.e. immediate subterms) of t' in G, $Succ^+(t') \subseteq Succ(t')$ denotes the sequence of all successors of a term t' in G that are not of the form `without` t'', and $Succ^-(t)$ denotes the sequence of all successors of a term t' in G that are of the form `without` t'' (i.e. $Succ^+(t') \uplus Succ^-(t') \equiv Succ(t')$). Furthermore, $Succ^!(t') \subseteq Succ(t')$ denotes the sequence of all successors of a term t' in G that are not of the form `optional` t'', and $Succ^?(t') \subseteq Succ(t')$ denotes the sequence of all successors of a term t' that are of the form `optional` t'' (i.e. $Succ^!(t') \uplus Succ^?(t') \equiv Succ(t')$). Note that $Succ^- \subseteq Succ^!$, because a combination of `without` and `optional` is not reasonable.[5]

Definition 4.8 (Ground Query Term Simulaton)
Let r_1 and r_2 be ground (query) terms, and let $G_1 = (V_1,E_1,r_1)$ and $G_2 = (V_2,E_2,r_2)$ be the graphs induced by r_1 and r_2. A relation $\preceq \subseteq V_1 \times V_2$ on the sets V_1 and V_2 of immediate and indirect subterms of r_1 and r_2 is called a *ground query term simulation*, if and only if:

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

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

3. if $v_1 \preceq v_2$ and v_1 is of the form *desc* t_1, then
   - v_2 is of the form *desc* t_2 and $t_1 \preceq t_2$ (*descendant preserving*, or

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

- $t_1 \preceq v_2$ (*descendant shallow*), or
- there exists a $v_2' \in SubT(v_2)$ such that $v_1 \preceq v_2'$ (*descendant deep*)

4. if $v_1 \preceq v_2$, v_1 has successors of the form `without` t, and v_2 is either of the form $l_2\{t_1,\ldots,t_m\}$ or of the form $l_2[t_1,\ldots,t_m]$, then the labels l_1 and l_2 of v_1 and v_2 match, and there exists a *total, index injective mapping* $\pi : Succ^+(v_1) \to Succ(v_2)$ such that for all $s \in Succ^+(v_1)$ holds that $s \preceq \pi(s)$. Depending on the kinds of subterm specifications of v_1 and v_2, π in addition satisfies the following requirements:

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

Furthermore, π is not *completable* with respect to the above mentioned properties to a (partial or total) mapping $\pi' : Succ(v_1) \to Succ(v_2)$ such that there exists a successor $t \in Succ^-(v_1)$ with t of the form `without` t' and $t' \preceq \pi(t)$.

In this case, the simulation is called *negation respecting*.

5. if $v_1 \preceq v_2$, both v_1 and v_2 have successors of the form `without` t, and v_2 is either of the form $l_2\{\{t_1,\ldots,t_m\}\}$ or of the form $l_2[[t_1,\ldots,t_m]]$, then the labels l_1 and l_2 of v_1 and v_2 match, and there exists a *total, index injective mapping* $\pi : Succ(v_1) \to Succ(v_2)$ such that

   - for all $s \in Succ^+(v_1)$ holds that $s \preceq \pi(s)$
   - for all $s \in Succ^-(v_1)$ such that s is of the form `without` s' holds that $\pi(s)$ is of the form `without` t' and $t' \preceq s'$ (*negation preserving*)[6]

   Depending on the kinds of subterm specifications of v_1 and v_2, π in addition satisfies the following requirements:

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

6. if $v_1 \preceq v_2$, and v_1 or v_2 have successors of the form `optional` t, then the labels l_1 and l_2 of v_1 and v_2 match and there exists a *partial or total, index injective mapping* $\pi : Succ(v_1) \to Succ(v_2)$ such that

   - π is total on $Succ^!(v_1)$
   - depending on the kinds of subterm specifications and occurrences of subterm negations in v_1 and v_2, π satisfies the requirements listed in the tables above
   - for all $s \in Succ^!(v_1)$ holds that $\pi(s) \in Succ^!(v_2)$ and $s \preceq \pi(s)$
   - for all $s \in Succ^?(v_1)$ such that s is of the form `optional` s' for which π is defined holds that either
     (a) $\pi(s) \in Succ^!(v_2)$ and $s' \preceq \pi(s)$, or
     (b) $\pi(s) \in Succ^?(v_2)$, $\pi(s)$ is of the form `optional` t', and $s' \preceq t'$
   - π is not completable to a mapping π' that also satisfies these requirements[7]

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

[6] Note that this property requires $t' \preceq s'$ although one might expect $s' \preceq t'$ on a first glance. The reason is that s' needs to exclude at least the same subterms as t' and therefore needs to be more general

[7] This restriction, while not strictly necessary for ground terms, ensures that always a maximal number of optional subterms participates in a simulation and thus yields variable bindings.

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

Example 4.19
The following comprehensive list of examples illustrates the different requirements for a ground query term simulation. They are grouped in categories, each referring to the relevant requirement in Definition 4.8.

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

1. **total ordered term specification (cf. requirement 2)**
   Let $t_1 = f[a_1, b_2, c_3]$, $t_2 = f[a_1, b_2, c_3, d_4]$, $t_3 = f[a_1, c_2, b_3]$, $t_4 = f\{a_1, b_2, c_3\}$, and $t_5 = g[a_1, b_2, c_3]$
   - $t_1 \preceq t_1$: there exists a total, index bijective, and index monotonic mapping π from $\langle a_1, b_2, c_3\rangle$ to $\langle a_1, b_2, c_3\rangle$ with $s \preceq \pi(s)$, mapping each subterm to itself.
   - $t_1 \not\preceq t_2$: there exists no index bijective mapping from $\langle a_1, b_2, c_3\rangle$ to $\langle a_1, b_2, c_3, d_4\rangle$, as the two sets have different cardinality.
   - $t_1 \not\preceq t_3$: there exists no index monotonic mapping from $\langle a_1, b_2, c_3\rangle$ to $\langle a_1, c_2, b_3\rangle$ with $s \preceq \pi(s)$; the only mapping that would satisfy $s \preceq \pi(s)$, i.e. $\{a_1 \mapsto a_1, b_2 \mapsto b_3, c_3 \mapsto c_2\}$, is not index monotonic.
   - $t_1 \not\preceq t_4$: the braces of t_1 and t_4 are incompatible.
   - $t_1 \not\preceq t_5$: the labels of t_1 and t_5 do not match.
2. **total unordered term specification (cf. requirement 2)**
   Let $t_1 = f\{a_1, b_2, c_3\}$, $t_2 = f[a_1, b_2, c_3, d_4]$, $t_3 = f[a_1, c_2, b_3]$, $t_4 = f\{a_1, b_2, c_3\}$, and $t_5 = g[a_1, b_2, c_3]$
   - $t_1 \preceq t_1$: there exists a total and index bijective mapping π from $\langle a_1, b_2, c_3\rangle$ to $\langle a_1, b_2, c_3\rangle$ with $s \preceq \pi(s)$, mapping each subterm to itself, thus being position preserving.
   - $t_1 \not\preceq t_2$: there exists no index bijective mapping from $\langle a_1, b_2, c_3\rangle$ to $\langle a_1, b_2, c_3, d_4\rangle$, as the two sets have different cardinality.
   - $t_1 \preceq t_3$: there exists a total and index bijective mapping π from $\langle a_1, b_2, c_3\rangle$ to $\langle a_1, c_2, b_3\rangle$ with $s \preceq \pi(s)$, the mapping $\{a_1 \mapsto a_1, b_2 \mapsto b_3, c_3 \mapsto c_2\}$ (it does not need to be index monotonic) and it is trivially position respecting, because t_1 does not contain position subterms.
   - $t_1 \preceq t_4$: there exists a total and index bijective mapping π from $\langle a_1, b_2, c_3\rangle$ to $\langle a_1, b_2, c_3\rangle$ with $s \preceq \pi(s)$, mapping each subterm to itself, thus being position preserving.
   - $t_1 \not\preceq t_5$: the labels of t_1 and t_5 do not match
3. **partial ordered term specification (cf. requirement 2)**
   Let $t_1 = f[[b_1, c_2]]$, $t_2 = f[a_1, b_2, c_3, d_4]$, $t_3 = f[a_1, c_2, b_3]$, $t_4 = f\{a_1, b_2, c_3\}$, and $t_5 = f[b_1, a_2, c_3]$
   - $t_1 \preceq t_1$
   - $t_1 \preceq t_2$: there exists a total, index injective, and index monotonic mapping $\pi = \{b_1 \mapsto b_2, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$. It is trivially position respecting.
   - $t_1 \not\preceq t_3$: there exists no mapping π with $s \preceq \pi(s)$ that is also index monotonic, because t_3 does not contain b and c in the right order.
   - $t_1 \not\preceq t_4$: the braces of t_1 and t_4 are incompatible.
   - $t_1 \preceq t_5$: there exists a total, index injective, and index monotonic mapping $\pi = \{b_1 \mapsto b_1, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$. It is trivially position respecting.
4. **partial unordered term specification (cf. requirement 2)**
   Let $t_1 = f\{\{b_1, c_2\}\}$, $t_2 = f[a_1, b_2, c_3, d_4]$, $t_3 = f[a_1, c_2, b_3]$, $t_4 = f\{a_1, b_2, c_3\}$, $t_5 = f[b_1, a_2, c_3]$, and $t_6 = f[a_1, b_2, d_3]$. All mappings π on $Succ(t_1)$ are trivially position respecting and position preserving.
   - $t_1 \preceq t_1$
   - $t_1 \preceq t_2$: there exists a total, index injective mapping $\pi = \{b_1 \mapsto b_2, c_2 \mapsto c_3\}$ with $s \preceq \pi(s)$

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

5. **position specification (cf. requirement 2)**
   Let $t_1 = f\{\{c_1, \text{pos } 2\ b_2\}\}$, $t_2 = f[a_1, b_2, c_3]$, $t_3 = f[b_1, c_2, a_3]$, $t_4 = f[[a_1, b_2, c_3]]$ and $t_5 = f[[a_1, \text{pos } 2\ b_2, c_3]]$
   - $t_1 \preceq t_1$: there exists a total, index injective, position preserving mapping $\pi = \{c_1 \mapsto c_1, \text{pos } 2\ b_2 \mapsto \text{pos } 2\ b_2\}$ with $s \preceq \pi(s)$
   - $t_1 \preceq t_2$: there exists a total, index injective, position respecting mapping $\pi = \{c_1 \mapsto c_3), \text{pos } 2\ b_2 \mapsto b_2\}$ with $s \preceq \pi(s)$
   - $t_1 \not\preceq t_3$: there exists no position respecting mapping π with $s \preceq \pi(s)$; the only mapping with $s \preceq \pi(s)$ is not position respecting, as it contains pos $2\ b_2 \mapsto b_1$.
   - $t_1 \not\preceq t_4$: there exists no position preserving mapping π with $s \preceq \pi(s)$, because t_4 contains no subterm of the form `pos 2 t'`; position *respecting* is not sufficient, as t_4 is incomplete and might match further terms with b at a different position than 2, e.g. the term $f[a_1, d_2, b_3, c_4]$, in which case $\preceq$ would not be transitive.
   - $t_1 \preceq t_5$: there exists a total, index injective, position preserving mapping $\pi = \{c_1 \mapsto c_3), \text{pos } 2\ b_2 \mapsto \text{pos } 2\ b_)\}$ with $s \preceq \pi(s)$; in contrast to t_4, the term t_5 "preserves transitivity" of $\preceq$.

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

7. **unordered term specification: subterm negation (cf. requirements 4 and 5)**
   Let $t_1 = f\{\{a_1, \neg b_2\{\{d_1\}\}\}\}$, $t_2 = f\{a_1, c_2\}$, $t_3 = f\{a_1, b_2\{d_1, e_2\}, c_3\}$, $t_4 = f\{\{a_1, c_2\}\}$, $t_5 = f\{\{a_1, \neg b_2\{\{\ \}\}, c_3\}\}$, and $t_6 = f\{\{a_1, \neg b_2\{\{d_1, e_2\}\}, c_3\}\}$. All mappings π on $Succ(t_1)$ are trivially position respecting and position preserving.
   - $t_1 \preceq t_2$: there exists a (partial) mapping $\pi = \{a_1 \mapsto a_1\}$ that is total on $Succ^+(t_1) = a_1$ and for all $s \in Succ^+(t_1)$ holds that $s \preceq \pi(s)$, and π cannot be completed to a mapping π' such that there exists a $t \in Succ^-(t_1)$ of the form $\neg t'$ with $t' \preceq \pi'(t)$, because t_2 does not contain a subterm matching $b_2\{\{d_1\}\}$
   - $t_1 \not\preceq t_3$: every partial mapping π with $s \preceq \pi(s)$ for all $s \in Succ^+(t_1)$, i.e. only the mapping $\pi = \{a_1 \mapsto a_1\}$, can be completed to a mapping π', i.e. the mapping $\pi' = \{a_1 \mapsto a_1, \neg b_2\{\{d_1\}\} \mapsto b_2\{d_1, e_2\}\}$, such that there exists a $t \in Succ^-(t_1)$ of the form $\neg t'$ (i.e. $\neg b_2\{\{d_1\}\}$) with $t' \preceq \pi'(t)$ (i.e. $b_2\{\{d_1\}\} \preceq b_2\{d_1, e_2\}$)
   - $t_1 \not\preceq t_4$: there exists no mapping π such that all $t \in Succ^-(t_1)$ are mapped on $\pi(t)$ of the form $\neg t'$, as $Succ^-(t_4) = \emptyset$. Note that t_4 is of a form defined in requirement 4.
   - $t_1 \preceq t_5$: there exists a total index injective (and vacuously position respecting) mapping $\pi = \{a_1 \mapsto a_1, \neg b_2\{\{d_1\}\} \mapsto \neg b_2\{\{\ \}\})\}$ such that for all $t \in Succ^+(t_1)$ holds that $t \preceq \pi(t)$ (i.e. $a_1 \preceq a_2$), and for all $\neg t \in Succ^-(t_1)$ with $\neg t' = \pi(\neg t)$ holds that $t' \preceq t$, i.e. $b\{\{\ \}\} \preceq b_2\{\{d_1\}\}$. Note that because of the negation, it is necessary that $t' \preceq t$ instead of $t \preceq t'$; otherwise, transitivity of $\preceq$ would not be guaranteed (see the footnote in requirement 5).
   - $t_1 \not\preceq t_6$: in the only mapping $\pi = \{a_1 \mapsto a_1, \neg b_2\{\{d_1\}\} \mapsto \neg b_2\{\{d_1, e_2\}\}\}$ in which for all $t \in Succ^+(t_1)$ holds that $t \preceq \pi(t)$ (i.e. $a_1 \preceq a_2$), it does not hold that for all $\neg t \in Succ^-(t_1)$ with $\neg t' = \pi(\neg t)$ holds that $t' \preceq t$, because $b\{\{d_1, e_2\}\} \not\preceq b_2\{\{d_1\}\}$. Note that because of the negation, it is necessary that $t' \preceq t$ instead of $t \preceq t'$; otherwise, transitivity of $\preceq$ would not be guaranteed, because t_5 could match with a term that would not match with t_1, e.g. $f\{a_1, \neg b_2\{d_1\}, c_3\}$ (see again the footnote in requirement 5).

8. **ordered term specification: subterm negation (cf. requirements 4 and 5)**
   Let $t_1 = f[[a_1, \neg b_2]]$, let $t_2 = f[b_1, a_2, c_3]$, and let $t_3 = f[a_1, c_2, b_3]$. Position requirements are again trivial.
   - $t_1 \preceq t_2$: there exists an index monotonic mapping $\pi = \{(a_1, a_2)\}$ that is total on $Succ^+(t_1) = a_1$ and for all $s \in Succ^+(t_1)$ holds that $s \preceq \pi(s)$, and π cannot be completed to a mapping π' such that there exists a $t \in Succ^-(t_1)$ of the form $\neg t'$ with $t' \preceq \pi'(t)$, such that π' is index monotonic; the only feasible completion $\pi = \{a_1 \mapsto a_2, \neg b_2 \mapsto b_1\}$ is not index monotonic.

- $t_1 \not\preceq t_3$: there exists an index monotonic mapping $\pi = \{a_1 \mapsto a_1\}$ that is total on $Succ^+(t_1) = a_1$ and for all $s \in Succ^+(t_1)$ holds that $s \preceq \pi(s)$, but it can be completed to an index monotonic mapping $\pi' = \{a_1 \mapsto a_1, \neg b_2 \mapsto b_3\}$ such that there exists a $t \in Succ^-(t_1)$ of the form $\neg t'$ (i.e. $\neg b_2$) with $t' \preceq \pi'(t)$

9. **optional subterms (cf. requirement 6)**
   Let $t_1 = f\{\{\text{opt } a_1, b_2\}\}$, let $t_2 = f\{b_1, c_2\}$, let $t_3 = f\{a_1, b_2, c_3\}$, and let $t_4 = f\{\{a_1, \text{opt } b_2\}\}$
   - $t_1 \preceq t_1$: there exists an injective mapping $\pi = \{\text{opt } a_1 \mapsto \text{opt } a_1, b_2 \mapsto b_2\}$ such that $s \preceq \pi(s)$ that cannot be completed to a mapping π' with these properties, as π is already total; the other mapping $\tau = \{b_2 \mapsto b_2\}$ fulfils the same properties, but can be completed to π.
   - $t_1 \preceq t_2$: there exists a partial injective mapping $\pi = \{b_2 \mapsto b_1\}$ such that $s \preceq \pi(s)$ for all $s \in Succ^!(t_1)$ that cannot be completed to a π' with these properties, as a_1 does not simulate in any subterm of t_2.
   - $t_1 \preceq t_3$: there exists an injective mapping $\pi = \{\text{opt } a_1 \mapsto a_1, b_2 \mapsto b_2\}$ such that $s \preceq \pi(s)$ that cannot be completed to a mapping π' with these properties, as π is already total; the other mapping $\tau = \{b_2 \mapsto b_2\}$ fulfils the same properties, but can be completed to π.
   - $t_1 \not\preceq t_4$: the mapping $\pi = \{\text{opt } a_1 \mapsto a_1, b_2 \mapsto \text{opt } b_2\}$, which is the only mapping respecting the other properties, does not fulfil the requirement that for all $s \in Succ^!(t_1)$ holds that $\pi(s) \in Succ^!(t_2)$. This restriction is important, because t_4 does not guarantee that there exists a subterm labelled b.

4.4.4 Simulation Order and Simulation Equivalence

Ground query term simulation has been designed carefully to be transitive and reflexive, because it is desirable that ground query term simulation is an ordering over the set $\mathcal{T}^g$ of ground query terms. In particular, this property is used in the definition of *answers* below.

Theorem 4.9
$\preceq$ is reflexive and transitive.

Proof. cf. Appendix B.1 □

With this result, the following corollary follows trivially:

Corollary and Definition 4.10
$\preceq$ defines a preorder[8] on the set of all ground query terms called the *simulation order*.

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

Definition 4.11 (Simulation Equivalence)
Two ground query terms t_1 and t_2 are said to be *simulation equivalent*, denoted $t_1 \cong t_2$, if $t_1 \preceq t_2$ and $t_2 \preceq t_1$.

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

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

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

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

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

Corollary 4.12
$\preceq$ is a partial ordering on $\mathcal{T}^g/_{\cong}$.

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

4.5 Queries

A *query* is a connection of zero or more query terms using the n-ary connectives *and* and *or*, the query negation *not*, and the conditional constructs *case* and *if*. A query may furthermore be associated with resources against which the query terms are evaluated. Detached query terms (i.e. query terms not contained in one of the aforementioned constructs) and detached query terms associated with a resource are called *query atoms*, otherwise the query is a *compound query*.

4.5.1 Resource Declarations

Queries may be associated with input resource declarations expressed in terms of a URI or IRI, in which case all query terms that are part of the query are evaluated against the XML documents or semistructured databases located at the given URI/IRI. Resource declarations allow Xcerpt programs to consider any Web site as input for a query program. An input resource declarations has the following form:

```
in {
  resource [ <uri>, <format> ],
  <query>
}
```

`<uri>` is the URI or IRI used to locate the resource on the Web. A URI/IRI may refer to any Web resource, but the current prototype (cf. Chapter A) currently only supports resources accessible via the network protocols `http` (*Hypertext Transfer Protocol*) and `file` (i.e. files located on the local disk). `<format>` optionally specifies the format of the resource and may be used by the runtime system to choose the correct parser. Feasible input formats for Xcerpt are resources that describe semistructured data in various formats (e.g. Xcerpt, XML, HTML, LISP, RDF, OEM or BibTeX).

Example 4.20
Assume that the XML document bib.xml containing the data of bookstore A is accessible via the URI `http://www.xcerpt.org/bib.xml`. A query retrieving all book titles in this document is expressed as follows:

```
in {
  resource [ "http://www.xcerpt.org/bib.xml", "xml" ],
  bib {{
    book {{
      var Title → title {{ }}
    }}
  }}
}
```

Resource declarations may be nested, in which case the innermost declaration is relevant for the query; all outer declarations are shadowed.

4.5.2 Conjunctions and Disjunctions of Queries

Queries can be connected with the n-ary boolean connectives `and` and `or`. An expression of the form `and{`Q_1`, ..., `Q_n`}` is an *and* connected query, an expression of the form `or{`Q_1`, ..., `Q_n`}` is an *or* connected

query. Intuitively, `or` merely *merges* the resulting sets of substitutions resulting from the queries $Q_1, \ldots, Q_n$ (like `union` in relational database systems), whereas `and` creates the cross product of the substitution sets and thus *joins* the individual substitutions (if none of the Q_i contains a negation).

Curly braces in boolean connectives leave the evaluation order open to the runtime system. The evaluation engine may then apply heuristics to determine an optimal order of evaluation. For instance, the evaluation engine might prefer queries that do not involve network I/O. Square brackets enforce a specific evaluation order.

Example 4.21 (Boolean Connectives in Queries)

```
and {
  in {
    resource [ "file:bib.xml" ],
    bib [[
      book [[
        title [ var T  ],
        price [ var Pa ]
      ]]
    ]]
  },
  in {
    resource [ "file:reviews.xml" ],
    reviews [[
      entry [[
        title [ var T ],
        price [ var Pb ]
      ]]
    ]]
  }
}
```

Two queries to the XML documents `bib.xml` and `reviews.xml` connected with the boolean connective `and`. Note that the two query terms share the variable `T`. Since bindings of a variable in a substitution need to be consistent, this expression joins the prices of books with the same title in the two book stores.

```
or {
  in {
    resource [ "file:bib.xml" ],
    bib [[
      book [[
        title [ var T  ],
        price [ var P ]
      ]]
    ]]
  },
  in {
    resource [ "file:reviews.xml" ],
    reviews [[
      entry [[
        title [ var T ],
        price [ var P ]
      ]]
    ]]
  }
}
```

Two queries to the XML documents `bib.xml` and `reviews.xml` connected with the boolean connective `or`. Although the two query terms share the variable `T` and `P`, this query does not evaluate a join as `or` represents *alternatives*; results of the two queries are thus merely merged (i.e. the union of the two sets of substitutions is formed), and the result is a combined list of book titles and prices from the two book stores. Books found in both bookstores are listed twice.

In the following, query specifications are often conveniently denoted by infix or postfix $\wedge$ instead of `and` and infix or postfix $\vee$ instead of `or`, as in $Q_1 \wedge \cdots \wedge Q_n$ instead of `and{Q1,...,Qn}`.

Note that it is possible to specify empty conjunctions and disjunctions. As is common in logic, the empty conjunction represents *truth* and the empty disjunction represents *falsity*. Convenient abbreviations are thus `True` for the empty conjunction and `False` for the empty disjunction.

4.5.3 Query Negation: not

Besides the subterm negation introduced in Section 4.3.4 above (`without` construct), Xcerpt also supports *query negation*, denoted by expressions of the form `not` Q. The query negation used in Xcerpt is *negation as (finite or infinite) failure* like in logic programming, i.e. a negated query `not` Q succeeds if the query Q fails. Like in negated subterms, variables occurring in a negated query do not yield bindings, i.e. they have to appear elsewhere in the query outside the scope of a negation construct (cf. *range restrictedness*, Section 6.2).

Example 4.22
Recall the two XML documents `bib.xml` and `reviews.xml` representing the data of two book stores introduced in Section 2.4.2. The following query uses query negation to query for such books that appear in the first document but not in the second:

```
and {
  in {
    resource [ "file:bib.xml", "xml" ],
    bib {{
      book {{
        title { var Title }
      }}
    }}
  },
  not in {
    resource [ "file:reviews.xml", "xml" ],
    bib {{
      review {{
        entry { var Title }
      }}
    }}
  },
}
```

From a theoretical viewpoint, classical negation would be advantageous as it would ensure a precise declarative semantics, which does not exist for negation as failure in some cases. However, classical negation is not feasible in many practical applications. Consider for example a train time table. Using classical negation, the time table would have to contain entries not only for the train connections that exist, but also for all train connections that do *not* exist (i.e. infinitely many). With negation as failure, it is sufficient that the query for a train fails (i.e. an entry for a non-existent train does not exist) to fulfil the query. Because of the well-known problems with the declarative semantics of negation as failure, negation in Xcerpt requires so-called *stratification* (cf. Section 6.4).

In contrast to the subterm negation `without` introduced in Section 4.3.4, query negation is a *universal* negation. If the query is negated, there must not exist a term with which it matches, i.e. all terms are required to `not` match with the pattern. Note that (subterm and query) negation is not covered in the formal semantics described in Chapter 7.

4.5.4 Conditions

Using patterns to restrict admissible variable bindings in a query (either by variable position in the pattern or by pattern restrictions) is limited to structural properties like the term-subterm or sibling relationships and does not allow to express conditions that go beyond pattern matching (like "the value of variable V has to be larger than 50"). To express such *semantic conditions*, Xcerpt uses so-called *condition boxes* (reminiscent of the condition boxes used in the language *QBE* [127]). A condition box is attached to a query $\mathfrak{Q}$ and has the form

$\mathfrak{Q}$ `where { Conditions }`

Conditions apply only to the variables occurring in $\mathfrak{Q}$; other variables besides those occurring in $\mathfrak{Q}$ are not allowed to occur in the condition box. Conditions are comparison operators (e.g. $>$, $<$, $\geq$, $\leq$, $=$, or $\neq$) and need to have at least one of the variables occurring in the query as parameter. It is furthermore possible to use arithmetic expressions in conditions (cf. Section 4.6.3 below), but aggregation constructs

are not allowed, as the conditions apply to each different substitution separately (like the `WHERE` part in the language SQL [6])[9].

Example 4.23
The following query uses a condition box to select all students in the `students.xml` document (cf. Section 2.4.1) that have a score higher than 10 in the first exercise:

```
in {
  resource  [ "file:students.xml" ],
  students {{
    var S → student {{
      desc exercise {
        number { 1 },
        score  { var Score }
      }
    }}
  }}
} where { var Score > 10 }
```

Condition boxes may be attached to *any* kind of query, including `and` and `or` connected queries. The following example illustrates this property:

Example 4.24
Recall the book store databases introduced in Section 2.4.2. The following query selects books that are cheaper in bookstore A than in bookstore B (cf. also Example 4.21):

```
and {
  in {
    resource [ "file:bib.xml" ],
    bib [[
      book [[
        title [ var T  ],
        price [ var Pa ]
      ]]
    ]]
  },
  in {
    resource [ "file:reviews.xml" ],
    reviews [[
      entry [[
        title [ var T ],
        price [ var Pb ]
      ]]
    ]]
  }
} where { var Pa < var Pb }
```

When evaluating conditions, a basic type system would be desirable to distinguish e.g. the comparison operator $<$ on numbers from the comparison operator $<$ on strings. As typing is not investigated in this thesis, it is assumed that different operator symbols are used for different types and type casting is performed implicitly by the runtime system.

[9] Note that SQL supports conditions over aggregated values in a `HAVING` clause. Such conditions can be expressed in Xcerpt via rule chaining

4.6 Construct Terms: Patterns for Constructing Data

Construct terms serve to reassemble variable bindings, which are determined by query terms, so as to form new data terms. Whereas query terms are patterns for the data (and thus may contain partial term specifications), construct terms are patterns for the result (and thus may only contain total term specifications). Construct terms may furthermore contain variables (but no → restrictions), and so-called *grouping constructs* used for grouping different substitutions. Like in data terms, both constructs `[ ]` and `{ }` may occur in construct terms for expressing ordered and unordered sequences of subterms. The constructs `[[ ]]` and `{{ }}` are not allowed, as they express partial term specifications which do not make sense when constructing data items.

4.6.1 Variables

In construct terms, variables serve as place holders for the subterms they are bound to. Construct terms may contain label variables, namespace variables, and subterm variables without restriction. Allowing variable restrictions in construct terms is not desirable, as a construct term is merely a specification of how the variables should be reassembled and is not intended to constrain the set of possible variable bindings. Obviously, however, label variables may take only values that are admissible as term labels and namespace variables may only be bound to URIs or IRIs.

Example 4.25
In the bookstore example from Section 2.4.2, assume that there is the following set of answer substitutions for the variables `Title` and `Author`:

σ_1	`Title`	`title { "Vikinga Blot" }`
	`Author`	`author { last { "Ingelman-Sundberg" }, first { "Catharina" }}`
σ_2	`Title`	`title { "Boken Om Vikingarna" }`
	`Author`	`author { last { "Ingelman-Sundberg" }, first { "Catharina" }}`
σ_3	`Title`	`title { "Folket i Birka på Vikingarnas Tid" }`
	`Author`	`author { last { "Wahl" }, first { "Mats" }}`
σ_4	`Title`	`title { "Folket i Birka på Vikingarnas Tid" }`
	`Author`	`author { last { "Nordqvist" }, first { "Sven" }}`
σ_5	`Title`	`title { "Folket i Birka på Vikingarnas Tid" }`
	`Author`	`author { last { "Ambrosiani" }, first { "Björn" }}`

The following construct term collects a single title/author pair for these substitutions (one for each substitution):

```
results {
  result { var Title, var Author }
}
```

The result of applying the substitutions above to this construct term are the following five data terms:

```
results {
  result {
    title { "Vikinga Blot" },
    author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
  }
}

results {
  result {
```

```
    title { "Boken Om Vikingarna" },
    author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
  }
}

results {
  result {
    title { "Folket i Birka på Vikingarnas Tid" },
    author { last { "Wahl" }, first { "Mats" } }
  }
}

results {
  result {
    title { "Folket i Birka på Vikingarnas Tid" },
    author { last { "Nordqvist" }, first { "Sven" } }
  }
}

results {
  result {
    title { "Folket i Birka på Vikingarnas Tid" },
    author { last { "Ambrosiani" }, first { "Björn" } }
  }
}
```

4.6.2 Grouping and Sorting: all and some

It is often desirable to collect all bindings for a variable in a single answer term. The *grouping constructs* `all` and `some` serve this purpose:

- `all` groups *all possible instances* of the enclosed subterms resulting from different variable bindings as children of the enclosing term. At least one instance has to exist, and the number of instances always needs to be finite (otherwise the program does not terminate).
- `some` groups non-deterministically *some of the possible instances* of the enclosed subterms resulting from variable bindings as children of the enclosing term. Some is quantified by a number which restricts the (maximum) number of alternatives to use. At least one instance has to exist.

The requirement that there has to exist at least one instance in both grouping constructs may seem unintuitive. However, a construct term can only be evaluated if the rule it is part of "fires", i.e. the query part succeeds and thus yields at least one substitution for the variables occurring in the query. If this behaviour is not desired, the grouping constructs can be combined with `optional` (see below).

Example 4.26 (Grouping Constructs)
Consider again the substitutions of Example 4.25. The following construct term creates a list of `result` subterms (one for each title/author combination from the substitutions) below a `results` term using the `all`-construct to collect all instances:

```
results {
  all result { var TITLE, var AUTHOR }
}
```

The result of applying the substitutions to this construct term might be the following data term (compare with the set of data terms from Example 4.25):

```
results {
  result {
    title { "Vikinga Blot" },
    author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
  },
  result {
    title { "Boken Om Vikingarna" },
    author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
  },
  result {
    title { "Folket i Birka på Vikingarnas Tid" },
    author { last { "Wahl" }, first { "Mats" } }
  },
  result {
    title { "Folket i Birka på Vikingarnas Tid" },
    author { last { "Nordqvist" }, first { "Sven" } }
  },
  result {
    title { "Folket i Birka på Vikingarnas Tid" },
    author { last { "Ambrosiani" }, first { "Björn" } }
  }
}
```

Formally, `all t` or `some n t` denote the grouping of all or some instances of `t` obtained from all possible bindings of the variables that are free in the term `t`. Subterms of `t` that again have the form `all t'` or `some n' t'` are recursively evaluated in the same manner (see below). A variable is *free* in a (sub)term `t`, if it (1) occurs in `t`, and (2) is not in the scope of another, nested grouping construct. E.g. in the term

```
results {
  all result { var TITLE, var AUTHOR }
}
```

both variables `TITLE` and `AUTHOR` are not free, since they are in the scope of an `all` construct. In the term

```
results {
  result { all var TITLE, var AUTHOR }
}
```

the variable `AUTHOR` is free, whereas the variable `TITLE` is not free. A variable is said to be *free for a grouping construct*, if it is free in the term enclosed by the grouping construct. E.g. in the term `all t`, all variables that are free in `t` are free for the outermost `all`. All free variables in a construct term need to have the same binding in each of the substitutions that are used for grouping.

Example 4.27
Consider a slightly modified variant of the previous construct term. Note that only the variable `AUTHOR` is in the scope of the `all` construct, while the variable `TITLE` is free.

```
result { var TITLE, all var AUTHOR }
```

The result of applying the set of answer substitutions of Example 4.25 to this construct term is the following set of data terms:

```
result {
  title { "Vikinga Blot" },
  author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
}

result {
  title { "Boken Om Vikingarna" },
  author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
}

result {
  title { "Folket i Birka på Vikingarnas Tid" },
  author { last { "Wahl" }, first { "Mats" },
  author { last { "Nordqvist" }, first { "Sven" },
  author { last { "Ambrosiani" }, first { "Björn" }
}
```

Note that each of the three resulting data terms uses only one binding for the variable TITLE of the construct term, but groups possibly several bindings of the variable AUTHOR. In each instance (i.e. data term), the grouping construct groups together substitutions that have the same binding for TITLE. As there exists only one substitution for each of the titles "Vikinga Blot" and "Boken Om Vikingarna", the grouping construct only groups a single substitution in the first two data terms. In the third data term, three substitutions are grouped (each having the same binding for TITLE, but a different binding for AUTHOR).

The grouping constructs all and some are similar to the so-called *collection constructs* {.} and [.] in XMAS [72] and to the grouping construct {.} in XML-RL [70].

Nesting of Grouping Constructs

Grouping constructs may be nested to perform more complex restructuring tasks. Recall that a term of the form all t collects all instances of t with different bindings for the free variables in t. If t contains nested grouping constructs, *each* instance of t is further grouped according to the nested grouping constructs. For example, the construct term

```
results {
  all result {
     all var TITLE,
     var AUTHOR
  }
}
```

creates for each binding of the variable AUTHOR (i.e. the variable that is free for the outer all) an instance of the subterm result. In each instance, the inner all collects all instances of the variable TITLE (that are part of substitutions with the same binding for AUTHOR). Thus, the construct term creates a list of book titles for each author, and groups the result subterms below a results term. Likewise, the construct term

```
results {
  all result {
     var TITLE,
     all var AUTHOR
  }
}
```

lists for each book title all authors. Intuitively, nested grouping constructs are similar to nested iteration constructs in imperative languages (like for or while loops), where the inner loop performs a complete run for each iteration of the outer loop. Note, however, that nested grouping constructs do not compute the

"cross-product", but instead have to respect the different answer substitutions: in the example above, every result elements contains a book title, and *only* the authors of that book, whereas the cross-product would list for each result also the authors of other books. If it is desirable to compute the cross-product, it is necessary to appropriately modify the query/query term such that it selects titles and authors independently.

Explicit Grouping: group by

In many cases, it is desirable to group by variables whose values should not appear in the result, wherefore they are not part of the subterm that is enclosed by a grouping construct. For example, a construct term might group resulting instances based on the position of a row in an HTML table while not including this position (i.e. the integer number) in the result. While this result could be achieved by using several rules (one for creating the result and one for filtering out superfluous parts), this solution is very cumbersome. For this reason, the grouping constructs `all` and `some` may be accompanied by a `group by` clause containing the (additional) variables by which the instances are grouped. Such clauses have the form

```
all <subterm> group by { <variables> }
```

or

```
some <n> <subterm> group by { <variables> }
```

where `<n>` is the maximum number of instances for `some`, `<subterm>` is the subterm of which instances are created, and `<variables>` is a comma-separated list of variables. All these variables are considered to be part of the free variables of the subterm enclosed by the grouping construct and thus used for grouping, regardless of whether they appear in `<subterm>` or not..

Example 4.28 (Explicit Grouping)
Consider an HTML table, the cells containing arbitrary values. The following query term retrieves all cell values, together with row and column number:

```
desc table {{
  position var Row tr {{
    position var Col td { var Value }
  }}
}}
```

Now assume that the table should be "transposed", i.e. rows and columns are exchanged. The following construct term creates such a transposed table. Since the positions are necessary for grouping but should not be included in the resulting data term, it uses `group by` for this purpose:

```
table [
  all tr [
    all td [ var Value ] group by { var Row }
  ] group by { var Col }
]
```

The construct term is evaluated as follows: For each different binding of `Col` (`Col` is the only free variable in the scope of the outer `all`), an instance of `tr [ ... ]` is created. Within each instance, the inner `all` creates an instance of `td [ ... ]` for each different binding of `Row` (within the set of substitutions having the same binding for `Col`).

Sorting: order by

The grouping constructs `all` and `some` create sequences of subterms in arbitrary order (although they should try to return results in the same order in which the corresponding subterms appear in the original

sources, if possible). In order to sort the resulting sequence according to the bindings for certain variables, the grouping constructs `all` and `some` may be augmented by a *sorting specification*. Sorting specifications are very similar to explicit grouping and have the form

```
all <subterm> order by (<comparison>) [ <variables> ]
```

or

```
some <n> <subterm> order by (<comparison>)[ <variables> ]
```

where `<n>` is the maximum number of instances for `some`, `<subterm>` is the subterm of which instances are created, and `<variables>` is a comma-separated list of variables. `<comparison>` is the name of the comparison function to be used in sorting. Comparison functions take as arguments two lists of terms (representing two different substitutions for the variables in `<variables>`) and return a value indicating whether the first list is less than, equal to, or greater than the second list. The current prototype runtime system (cf. Appendix A) supports the two exemplary comparison functions `lexical` and `numeric` (both in ascending order); further comparison functions may be programmed natively in the implementation language of the prototype (i.e. Haskell).

The list of variables influences the grouping in two ways: (1) instances are grouped as if the variables occurred in a `group by` clause (i.e. are considered part of the variables free for the grouping construct) and (2) the instances are sorted on the bindings of the variables in the list using the specified comparison function. In the two exemplary functions, sorting is performed primarily with respect to the first variable in the list and more specific for each of the following variables. For instance, a variable list `[var Last,var First]` would specify to sort primarily by the last names, and within instances with the same last name sort by the first name.

Example 4.29
Sort the list of books by the book titles in ascending lexical order:

```
results {
  all result { all var Author, var Title } order by (lexical) [ var Title ]
}
```

Example 4.30
Consider the following query term (evaluated against the XML document representing the data of bookstore A in Section 2.4.2):

```
bib {{
  book {{
    var Title → title {{ }}
    var Author → author {{ var First → first {{ }}, var Last → last {{ }} }}
  }}
}}
```

The following construct term creates a list of authors for each book title. Authors are sorted by last name and then by first name. Note that grouping is performed on the variable `Author`, as well as the variables `Last` and `First`.

```
results {
  all result {
    all var Author order by (lexical) [var Last, var First],
    var Title
  }
}
```

Comparison with GROUP BY and Aggregations in SQL

Xcerpt's grouping constructs are very similar to `GROUP BY` clauses in SQL [6], which allow to group results with the same bindings on the specified variables into a combined representation. In SQL, `GROUP BY` is usually used in conjunction with an aggregation function over some of the variables not used for grouping. However, grouping in Xcerpt differs from grouping in SQL in several aspects:

- grouping is part of the *construction* instead of the *query*
- grouping without aggregation functions is necessary, as Xcerpt, unlike SQL, allows complex tree structures instead of flat tuples.
- grouping constructs have a *scope*; therefore, it is in most cases not necessary to explicitly specify the variables used for grouping. Instead, all free variables in the scope (i.e. enclosed subterm) are implicitly used.
- grouping constructs can be *nested*; a nested grouping construct is very similar to an aggregation function that creates a term sequence.

In relational databases, nesting of grouping constructs would create results that are in non-first normal form, i.e. tuples that are not flat, which is usually not permitted. In Xcerpt, nesting is possible (and desirable) because the data is tree-structured in the first place.

Example 4.31
Consider a relation `Scores(Student,ExerciseNr,Score)` used for storing exercise results of students. To keep the example simple, it is assumed that the first attribute in a tuple (`Student`) holds the student name. The following table represents the data from Section 2.4.1):

Scores	Student	ExerciseNr	Score
	Donald Duck	1	15
	Donald Duck	2	7
	Mickey Mouse	1	3
	Mickey Mouse	3	14
	Goofy	2	13

To sum up the totals for each student in SQL, one usually groups on the attribute `Student` and aggregates (for each student) over the attribute `Score`. The attribute `ExerciseNr` is ignored:

```
SELECT Student, sum (Score) FROM Scores GROUP BY Student
```

In Xcerpt, the same result would be created with the following construct term using nested grouping constructs and an aggregation function (aggregations in Xcerpt are introduced in Section 4.6.3 below). Note that although `group by` is used in this construct term, it could be omitted because the variable `Student` already appears inside the outer `all` and thus is used implicitly for grouping:

```
totals {
  all score {
     name        { var Student },
     total-score { sum (all var Score) }
  } group by { var Student }
}
```

4.6.3 Functions and Aggregations

In addition to arranging data in a new structure, it is often desirable to perform some sort of computation to create new content. For example, a bookstore might want to present books with the value added tax added to all prices, or calculate totals for the items contained in a customer's virtual shopping cart. For this reason, construct terms in Xcerpt may contain *functions* (i.e. computations with a fixed number of arguments) and *aggregations* (i.e. computations with a variable number of arguments). Both functions and aggregations take the form

`<fname> ( <arguments> )`

where `<fname>` is the function or aggregation name and `<arguments>` is a comma-separated list of arguments (variables or other non-grouping subterms). In the case of aggregations, `<arguments>` may also contain the grouping constructs `all` and `some`.

Example 4.32 (Shopping Cart: Adding the VAT and Computing Totals)
Consider the XML document representing the data of bookstore A (in Section 2.4.2). The following construct term might be used to create an HTML presentation of books in a shopping cart were prices are shown both without and with value added tax (in this case: 16%). The last row computes totals for all prices.

```
table [
  th [ td [ "Title" ], td [ "Price Net" ], td [ "VAT" ], td [ "Total" ] ],
  all tr [
    td [ var Title ],
    td [ var Price ],
    td [ mult ( var Price, 0.16 ) ],
    td [ mult ( var Price, 1.16 ) ]
  ],
  tr [
    td [ "Totals" ],
    td [ sum ( all var Price ) ],
    td [ sum ( all mult ( var Price, 0.16) ) ],
    td [ sum ( all mult ( var Price, 1.16) ) ],
  ]
]
```

Since a type system is not in the scope of this thesis, the current implementation assumes implicit type casting in functions and aggregations. For example, the function `mult` (for multiplication) implicitly assumes that all parameters are numbers. In order to provide a comprehensive set of functions and aggregations, a type system would however be beneficial.

The current implementation supports a number of exemplary functions and aggregations, which are summarised in Table 4.1. For some frequently used functions, this table also gives an abbreviated, infix notation that may be used instead of the more verbose general form. Beyond these, a wide range of functions are conceivable. The document *XQuery 1.0 and XPath 2.0 Functions and Operators* [114] gives an overview over functions that are desirable in Web query languages.

4.6.4 Optional Subterms: optional

Recall from Section 4.3.1 that the construct `optional` in query terms allows to express that certain subterms of a query term need only be matched if a corresponding subterm exists in the data term against which the query term is evaluated. In case the optional subterm contains variables, it might happen that some of the substitutions resulting from the evaluation of the query do not contain bindings for these variables (as the corresponding subterm did not participate in the matching). As a consequence, construct terms containing such variables need to make provisions for such cases. This is expressed by marking subterms

Name	Abbreviated	Default	Description
		Functions	
`add(n,m)`	`n + m`	—	adds the two numeric arguments `n` and `m`
`sub(n,m)`	`n - m`	—	subtracts the two numeric arguments `n` and `m`
`mult(n,m)`	`n * m`	—	multiplies the two numeric arguments `n` and `m`
`div(n,m)`	`n / m`	—	divides the two numeric arguments `n` and `m`
`concat(n,m)`	`n ++ m`	—	concatenates the two string arguments `n` and `m`
`glb(n,m)`	—	—	calculates the greatest lower bound of two terms `n` and `m` wrt. simulation order
`lub(n,m)`	—	—	calculates the least upper bound of two terms `n` and `m` wrt. simulation order
		Aggregations	
`count(...)`	—	0	count the number of arguments
`sum(...)`	—	0	compute the sum of all (numeric) arguments
`avg(...)`	—	NaN	compute the average of all (numeric) arguments
`min(...)`	—	$+inf$	compute the minimum of all (numeric) arguments
`max(...)`	—	$-inf$	compute the maximum of all (numeric) arguments
`join(...)`	—	`""`	join all string arguments to a single string
`first(...)`	—	exception	return the first argument
`reverse(...)`	—	empty	return the arguments in reverse order

Table 4.1: Exemplary functions and aggregations available in construct terms. All functions and aggregations perform an implicit type casting to the type given (e.g. “numeric” or “string”). The default value for aggregation functions is used in case the argument list is empty; `exception` indicates a runtime error and `empty` the empty list. For normal functions, default values are not applicable.

containing variables that are possibly unbound as *optional*. Like in query terms, such subterms take the form

```
optional <subterm>
```

but it is also possible to add a default value to be used if no instance can be created as in

```
optional <subterm> with default <default>
```

where `<subterm>` is the subterm containing the optional variables. In case at least one of the variables in `<subterm>` is not bound (i.e. no ground instance can be created), the first form simply omits the optional subterm, whereas the second form substitutes the subterm `<default>` for `<subterm>`. `<default>` may be any construct term.

Example 4.33
Consider the XML document representing the student database of Section 2.4.1. The following query term retrieves the student name and optionally his inscription number (contained in the subterm labelled `matrnr`) from this document. Note the use of `optional` to indicate optional selections.

```
students {{
  student {{
    name { var Name },
    optional matrnr { var MatrNr }
  }}
}}
```

Assume that a teacher wants to create an HTML table listing all student names with inscription numbers, leaving columns empty for each substitution that does not contain a binding for the variable `MatrNr`. The corresponding construct term would look as follows:

```
table [
  all tr [
    td [ var Name ],
    td [ optional var MatrNr ]
  ]
]
```

By using a default specification, it us also possible to insert the string `"unknown"` instead of simply leaving the columns empty for those substitutions that do not contain a value for `MatrNr`, as in the following construct term:

```
table [
  all tr [
    td [ var Name ],
    td [ optional var MatrNr with default "unknown" ]
  ]
]
```

The `optional` does not necessarily prefix the variable immediately, but may instead enclose a whole subterm containing optional variables; the following construct term does not generate a second column if there is no inscription number available, instead of leaving the second column empty:

```
table [
  all tr [
    td [ var Name ],
    optional td [ var MatrNr ]
  ]
]
```

Grouping Constructs and Optional Subterms

As mentioned above, the grouping constructs `all` and `some` require the existence of at least one instance of the enclosed subterms, because a query only succeeds if there exists at least one binding for its variables. With `optional`, this restriction can be lifted: query terms with optional subterms might match while not yielding any bindings for the variables occurring in the grouping construct. With `optional` in the construct term, it is thus possible to express possibly empty groupings:

Example 4.34 (Grouping and Optional)
The following query term selects student names and scores of submitted exercises. The subterm labelled `exercises` is marked `optional` in order to also select students without any exercise submissions.

```
students {{
  student {{
    name { var Name },
    optional exercise {{ score { var Score } }}
  }}
}}
```

Given the set of answer substitutions of this query term, the following construct term computes the sum of all exercises for each student. If no exercise has been submitted, the sum is 0:

```
scores {
  all student {
    name { var name },
    total { sum ( optional all var Score with default 0 ) }
  }
}
```

In case no exercise has been submitted (i.e. there exists no binding for the variable `Score` for a student), there exists no instance for `all var Score` and the default value of 0 is used as the only argument to the aggregation function `sum(...)`. Otherwise, a sequence of scores is created and summed up using `sum(...)`. Since the default value of `sum(...)` happens to be 0 if the number of arguments is zero (cf. Table 4.1), the `with default` clause may also be omitted in this case. Alternatively, the construct term could be written as

```
scores {
  all student {
    name { var name },
    total { optional sum ( all var Score ) with default 0 }
  }
}
```

In this case, the aggregation function `sum(...)` is not evaluated if there is no instance for `all var Score`; the value 0 is substituted without any further computation.

In practise, it is irrelevant whether the `optional` encloses the grouping construct (as in the examples above) or vice versa; both approaches are reasonable.

4.7 Construct-Query Rules (or Views)

An Xcerpt *query program* (or simply *program*) consists of one or more *construct-query rules*. Construct-query rules (short: *rules*) relate a construct term to a query (i.e. an *and* or *or* connected set of query terms). Xcerpt rules are *if-then* rules, i.e. *if* a query succeeds, *then* a result is created. The syntax of rules in Xcerpt loosely resembles SQL and is similar to the syntax used in XML-QL and XMAS (cf. Section 3.3.4). Rules have the form

```
CONSTRUCT
  <construct term>
FROM
  <query>
END
```

where *<query>* is a query as defined in Section 4.5 and *<construct term>* is a single construct term. Rules are *range restricted*: all variables occurring in *<construct term>* must also occur and yield bindings in *<query>* (cf. Section 6.2). Also, all rules in a program are understood as if they were variable disjoint, i.e. the scope of a variable is limited to a rule (*"standardisation apart"*, cf. page 137). If *<query>* can be evaluated successfully, a rule is said to be *applicable*.

Intuitively, a rule specifies how to transform the source data (which possibly is located at several Web sites) into a different representation. The query (or "query part") specifies how to select data items from different source documents, the construct term (or "construct part") specifies how to reassemble these data items in a new combined structure, yielding new data terms. A rule can thus be seen as a *view* upon source

data, very much like the views in query languages for relational database systems. However, in contrast to views in such languages, an Xcerpt rule is not applicable if no data items exist, because the query part cannot be evaluated.

Example 4.35
The following rule creates a unified view upon the book databases of the two book stores introduced in Section 2.4.2, summarising the prices for books that appear in both databases. The query part consists of an *and*-connection of two query terms evaluated against the two XML documents representing the book databases. The construct part reassembles the book titles and prices in both book stores in a `books-with-prices` subterm.

```
CONSTRUCT
  books-with-prices [
    all book-with-prices [
      title [ var T ], price-a [ var Pa ], price-b [ var Pb ]
    ]
  ]
FROM
  and {
    in {
      resource [ "file:bib.xml" ],
      bib [[
        book [[
          title [ var T  ], price [ var Pa ]
        ]]
      ]]
    },
    in {
      resource [ "file:reviews.xml" ],
      reviews [[
        entry [[
          title [ var T  ], price [ var Pb ]
        ]]
      ]]
    }
  }
END
```

More formally, a rule (together with the queried resources) is a specification of a set of data terms (for more details, see Chapter 7). These are called *results* of the rule induced by the answer substitutions of the query. Whether or not this set is materialised depends on the concrete implementation and the needs of the application. Note, however, that in general a materialisation is not possible, because the number of induced data terms might be infinite due to (recursive) rule chaining (see below).

4.7.1 Rule Chaining

As in logic programming languages like Prolog or Datalog, Xcerpt rules can query the results (instances) of other rules, a process usually referred to as *rule chaining*. Recursive rule chaining is possible, in which case a rule queries the results of a previous application of itself. Rule chaining distinguishes Xcerpt from most of the languages introduced in Section 3.3.4: although the languages UnQL, XML-QL, and XMAS all are rule-based, neither of them supports rule chaining. Rule Chaining serves several purposes:

- It allows to break down complex queries in smaller components that are easier to grasp and more declarative.

- It allows to structure queries in logical components (“separation of concerns”). For example, a query might be composed of several rules that query different resources and create a unified intermediate format (“mediators”), and a rule that queries data in this intermediate format and creates an XML document suitable for presentation in a browser. Further rules might be used to create different presentation formats for mobile devices or paper editions suitable for printing.
- It allows to build complex queries that require recursion. This includes, in particular, reasoning with Semantic Web data: a frequently needed operation is e.g. the computation of the transitive closure of a relation.

Several application scenarios for rule chaining can be found in Chapter 5. In Xcerpt, any query that is not associated with an external resource is considered to refer to the results of other rules within the program.

Example 4.36
Recall the rule used in Example 4.35 above, which creates a unified representation of books in two book stores. The following Xcerpt rule further queries this unified representation to create an HTML document suitable for presentation in a browser (cf. also the more detailed description of this example in Section 5.1.5):

```
CONSTRUCT
   table [
     tr [ td [  "Title"  ], td [  "Price at A"  ], td [  "Price at B"  ] ],
    all tr [ td [ var Title], td [ var PriceA], td [ var PriceB] ]
   ]
FROM
   books-with-prices [[
     book-with-prices [[
       title   [[ var Title ]],
       price-a [[ var PriceA]],
       price-b [[ var PriceB]]
     ]]
   ]]
END
```

Likewise, the following (very similar) rule queries the same unified representation to create a WML document suitable for a mobile device (e.g. cellular phone):

```
CONSTRUCT
   wml [
    all card [
        "Title: ", var Title,
        "Price A: ", var PriceA,
        "Price B: ",  var PriceB
     ]
   ]
FROM
   books-with-prices [[
     book-with-prices [[
       title   [[ var Title  ]],
       price-a [[ var PriceA ]],
       price-b [[ var PriceB ]]
     ]]
   ]]
END
```

Operationally, rule chaining can be seen as very similar to procedure or function calls (or perhaps "routines") in other programming or query languages. Rule languages in general allow to evaluate rules in two directions, so-called "forward chaining" and "backward chaining".

- Forward chaining is *rule driven*. Rules are evaluated iteratively against the current set of data terms until saturation is achieved (the so-called *fixpoint*). Forward Chaining is useful for instance for materialising views and for view maintenance, and is widely used in deductive databases []. If a query program contains recursive rules, forward chaining could result in an infinite fixpoint, i.e. the evaluation does not terminate. Also, if forward chaining is used to answer a query, most of the derived data is usually irrelevant to the query – forward chaining is not goal driven.

- Backward chaining is *goal driven*. Beginning with a query (composed of one or several query terms), program rules are selected if they are relevant for "proving" a query term. The query term in question is then replaced by the query part of the selected rule. Backward chaining is useful when the expected result is small in comparison with the number of possible results of the program. On the other hand, naïve backward chaining may not terminate even in cases where the fixpoint is guaranteed to terminate. Backward chaining is mainly used in expert or knowledge base systems and in logic programming languages like Prolog [71].

In a Web environment, both forward and backward chaining are desirable. A forward chaining approach is e.g. useful when creating a static Web site (consisting of several Web pages) from an input document containing the content and an Xcerpt program used as a "stylesheet" for adding layout and structure suitable for presentation in a Web browser, i.e. "materialising the HTML view" on the input data. On the other hand, backward chaining is necessary when querying large collections of documents, in particular the Web itself, where it is in practise not possible to begin with the complete set of data terms. In fact, if the considered set of data terms is the complete Web, its contents might even be unknown at the beginning.

Whereas *pattern matching* (as introduced in Section 4.4) is sufficient for a forward chaining evaluation, a backward chaining evaluation requires *unification*, as query terms need to be "matched" with construct terms and variables in both terms need to be bound. Xcerpt uses a non-standard unification algorithm called *simulation unification*, which is introduced in Chapter 8. Chapter 8 also describes a backward chaining algorithm for Xcerpt programs, and [25] compares different approaches to backward chaining in Xcerpt. A forward chaining algorithm is not investigated in this thesis.

4.7.2 Goals

Xcerpt programs may contain a particular form of rules called *goals*. The first instance of the construct term of a goal is considered to be a *result* of a program. Goals serve as the starting point of a backward chaining evaluation, but are otherwise very similar to the normal rules introduced in this Section. They have the form

```
GOAL
  out {
    resource [ <resource specification> ],
    <construct term>
  }
FROM
  <query>
END
```

where <construct term> and <query> are defined as for normal rules. A goal is always associated with an *output resource* (which uses the same syntax as the input resources introduced in Section 4.5) specifying the resource to which the result is written in case the goal is evaluated successfully. If no output resource is given, it is implicitly assumed that the result shall be written to standard output, e.g. the current console. In this case, goals have the form

```
GOAL
  <construct term>
FROM
  <query>
END
```

The instances of goals cannot be queried by the queries of other rules. Goals thus do not participate in rule chaining, except for being the starting point of the (backward chaining) evaluation. Every program needs to contain at least one goal.

Example 4.37
Consider the two rules of Example 4.36. The following two goals may be used to write their results to the files `prices.html` (in HTML format) and `prices.wml` (in WML format):

```
GOAL
  out {
    resource [ "file:prices.html", "html" ],
    html [
      head [ title [ "Price Comparison" ] ],
      body [ var Content ]
    ]
  }
FROM
  var Content → table {{ }}
END

GOAL
  out {
    resource [ "file:prices.wml", "wml" ],
    var Content
  }
FROM
  var Content → wml {{ }}
END
```

CHAPTER

FIVE

Xcerpt Use Cases

5.1 Restructuring Data

5.1.1 List of Authors vs. List of Titles

The following two examples are taken from the use case *XMP* of the *XML Query Use Cases* [34] (queries Q3 and Q4). Consider the document representing the bibliography database of bookstore A in Section 2.4.2 (Figure 2.5). In the first example, the task is to list "for each book in the bibliography the title and authors, grouped inside a `result` element"; the second example lists "for each author in the bibliography the author's name and the titles of all books by that author, grouped inside a `result` element". The following two Xcerpt rules create these results using nested `all` constructs.

```
CONSTRUCT
  results [
    all result [
      var Title,
      all var Author
    ]
  ]
FROM
  in {
    resource [ "file:bib.xml" ],
    bib [[
      book [[
        var Title → title {{ }},
        var Author → author {{ }}
      ]]
    ]]
  }
END
```

```
CONSTRUCT
  results [
    all result [
      all var Title
      var Author,
    ]
  ]
FROM
  in {
    resource [ "file:bib.xml" ],
    bib [[
      book [[
        var Title → title {{ }},
        var Author → author {{ }}
      ]]
    ]]
  }
END
```

Note that the two Xcerpt rules are mostly identical, except for the position of the inner `all` construct. In both cases, the query part consists of a single query term associated with the resource `file:bib.xml`, i.e. an XML document named `bib.xml` and located in the local file system. This query term binds the variables `Title` and `Author` to corresponding pairs of `title`/`author` elements in `bib.xml`. These bindings are used in the construct term to construct the result:

- In the first example (on the left), the primary grouping is performed on the book titles, i.e. one instance of a `result` element is created for each different binding of `Title`. Nested grouping is

then performed on the variable `Author`, simply listing all different bindings (for a given instance of `Title`).

- In the second example (on the right), the primary grouping is instead performed on the book authors, i.e. one instance of a `result` element is created for each different binding of `Author`. The nested grouping then lists all bindings of `Title` for each such instance.

Interestingly, these two examples differ considerably for the language XQuery (whereas in Xcerpt, the only difference is the position of the nested `all` construct). The following two XQuery queries are taken from the *XML Query Use Cases* and yield similar results to the two Xcerpt queries above[1]:

```
<results>
{
  for $b in doc("file:bib.xml")/bib/book
  return
    <result>
      { $b/title }
      { $b/author  }
    </result>
}
</results>
```

```
<results>
{
  let $a := doc("file:bib.xml")//author
  for $last in distinct-values($a/last),
      $first in distinct-values($a[last=$last]/first)
  order by $last, $first
  return
    <result>
      <author>
        <last>{ $last }</last>
        <first>{ $first }</first>
      </author>
      {
        for $b in doc("file:bib.xml")/bib/book
        where some $ba in $b/author
              satisfies ($ba/last = $last
                     and $ba/first=$first)
        return $b/title
      }
    </result>
}
</results>
```

It is easy to observe that the left query is much simpler than the right query (which requires the use of nested subqueries), although the queried data is identical. A reason for this might be that while the result of the first query is similar in structure to the queried data, the second query requires considerable restructuring. Arguably, separation of querying and construction in Xcerpt better conveys the structure of both, the result and the queried data.

5.1.2 Resolving ID/IDREF references

While Xcerpt provides its own reference mechanism (see Section 4.2), it is also straightforward to use and dereference ID/IDREF references using a variable that occurs in a query term both at the position of the ID and at the position of the IDREF. Suppose there exists an XML document representing a large text (e.g. this PhD thesis, see Section 2.4.3). References to the bibliography might be represented using ID/IDREF. The following query selects all authors cited in a section entitled “Xcerpt Terms” by dereferencing ID/IDREF references in `cite` elements to the respective `entry` elements in the bibliography and retrieving the authors contained in them:

```
CONSTRUCT
  authors {
    all var Author
  }
FROM
  in {
    resource [ "file:report.xml" ],
```

[1] the ordering of results may be different as ordering is not required in the task description

```
    report {{
      desc section {{
        title { "Xcerpt Terms" },
        desc cite {
          attributes { ref { var Ref } }
        }
      }},
      desc bibliography {{
        entry {{
          attributes { id { var Ref } },
          var Author → author {{ }}
        }}
      }}
    }}
  }
END
```

Note the use of the special `attributes` subterm to represent XML attributes in Xcerpt. In the first subterm of `report`, the variable `Ref` is bound successively to all identifiers of citations referred to the sections with title "Xcerpt Terms"; the same variable is used in the second subterm to select the corresponding bibliography entries, for which all authors are successively bound to the `Author` variable.

In combination with the subterm negation `without`, it is possible to verify whether an XML document contains references to non-existing identifiers. The following Xcerpt rule illustrates this on the PhD thesis example (compare also with the query above). It queries the text for all citations (retrieving them in the variable `Ref`) and checks whether the bibliography does not contain a corresponding entry. Note that the second occurrence of the variable `Ref` is part of a negated subterm.

```
CONSTRUCT
  unresolved_citations {
    all var Citation
  }
FROM
  in {
    resource [ "file:report.xml" ],
    report {{
      desc var Citation → cite {
        attributes { ref { var Ref } }
      },
      desc bibliography {{
        without entry {{
          attributes {{ id { var Ref } }}
        }}
      }}
    }}
  }
END
```

5.1.3 Completing an HTML table

Xcerpt's grouping constructs `all` and `some` can be used to perform powerful computations. Consider an HTML table containing numeric values (e.g. a spreadsheet represented in an HTML document), like the following (very simple) table:

```
<html>
  <head><title>A simple table</title></head>
  <body>
    <table>
      <tr>
        <td>1</td><td>2</td>
      </tr>
      <tr>
        <td>3</td><td>4</td>
      </tr>
    </table>
  </body>
</html>
```

A typical task could be to query this table and create a new table with the totals for each row and column added. Due to Xcerpt's grouping constructs, this query can be expressed using a single rule:

```
CONSTRUCT
  table [
    all tr [
      all td [ var Value ],
      td [ sum (all var Value) ]
    ] group by { var Row },
    tr [
      all td [
        sum (all var Value)
      ] group by { var Col },
      td [ sum (all var Value) ]
    ]
  ]
FROM
  in {
    resource [ "http://www.example.com/table.html", "html" ],
    html {{
      desc table {{
        position var Row tr {{
          position var Col td {{ var Value }}
        }}
      }}
    }}
  }
END
```

Note the use of the construct `group by` (cf. Section 4.6.2) and the use of the aggregation function `sum` (cf. Section 4.6.3). The query is evaluated as follows (compare with Example 4.28 on page 96): the query term selects all values of cells in the table, together with the respective row and column number (using the construct `position`). The construct term creates a table by adding one row (`tr` element) for each row in the original table (by grouping on the variable `Row`), and an additional row for the totals at the end of the table. In each row, all table cells of the respective row in the original table are inserted, and a new cell is added summing up the values of all cells (using the aggregation function `sum`). In the final row, a new cell is created for each column (by grouping on the variable `Col`, and each of these cells contains the totals of the whole column (using the aggregation function `sum` over all values that have the same column

number). Finally, the last cell of the last row is created by simply aggregating over all possible values without considering `Col` or `Row`.

An equivalent result can be achieved using the following two rules. Whereas the first rule adds a column containing the totals of each *row*, the second column adds a row containing the totals of each *column*. Both rules interact via rule chaining, i.e. the result of one rule application is queried by the other. As a consequence, it is not necessary to calculate the total value of all cells separately. Interestingly, the order in which the rules are applied is not relevant, both evaluation orders yield the same result (they are *confluent*). Note that the example is not complete: one of the rules needs to specify the resource from which the source document is to be retrieved.

```
CONSTRUCT
  table [
    all tr [
      all td [ var Value ],
      td [ sum (all var Value) ]
    ] group by { var Row },
  ]
FROM
  desc table {{
    position var Row tr {{
      position var Col td {{ var Value }}
    }}
  }}
END


CONSTRUCT
  table [
    all tr [
      all td [ var Value ],
    ] group by { var Row },
    tr [
      all td [
        sum (all var Value)
      ] group by { var Col },
    ]
  ]
FROM
  desc table {{
    position var Row tr {{
      position var Col td {{ var Value }}
    }}
  }}
END
```

5.1.4 List of Students

Consider the XML document representing the student database of Section 2.4.1. The following query term retrieves student information (i.e. name, optionally student id, and all exercises, if available) from this document. Note the use of `optional` to indicate optional selections.

```
students {{
  student {{
```

```
    name { var Name },
    optional matrnr { var MatrNr },
    optional exercise {{
      number { var Excercise },
      optional score { var Score }
    }}
  }}
}}
```

Assume that a teacher wants to create a Web site listing this information in an HTML table. He would probably use a construct term like the following:

```
table [
  all tr [
    td [ var Name ],
    td [ optional var MatrNr with default "unknown" ],
    td [
      optional ul [
        all li [
          "Exercise ", var Excercise,
          ", Score ", optional var Score with default "not yet available"
        ]
      ]
    ]
  ]
]
```

The result is constructed as follows: for each binding of `Name`, a table row is created containing the name in the first column. If there exists a binding for the variable `MatrNr` as well, the second column contains this binding; otherwise, it contains the value `"unknown"`. The third column creates an unordered HTML list, if at least one binding for `Excercise` exists (i.e. there exists an instance for the `ul [ ... ]` subterm). For each submitted exercise (binding of the variable `Excercise`), this list contains an entry. If a binding for `Score` is also available, it is included in the result; if not, the string `"not yet available"` is issued.

A sample result for the XML document of Section 2.4.1 thus looks as follows:

```
table [
  tr [
    td [ "Donald Duck" ],
    td [ "123456789" ],
    td [
      ul [
        li [ "Excercise ", "1", ", Score ", "15" ],
        li [ "Excercise ", "2", ", Score ", "3" ],
        li [ "Excercise ", "3", ", Score ", "not yet available" ]
      ]
    ]
  ],
  tr [
    td [ "Mickey Mouse" ],
    td [ "987654321" ],
    td [
      ul [
```

```
        li [ "Excercise ", "1", ", Score ", "3" ],
        li [ "Excercise ", "3", ", Score ", "14" ]
      ]
    ]
  ],
  tr [
    td [ "Goofy" ],
    td [ "unknown" ],
    td [
      ul [
        li [ "Excercise ", "2", ", Score ", "13" ],
        li [ "Excercise ", "3", ", Score ", "not yet available" ]
      ]
    ]
  ]
]
```

5.1.5 Separation of Concerns

Xcerpt rule chaining provides programmers with a means to structure complex query programs. A common way to structure a program is "separation of concerns", e.g. separating program logic or querying from presentation. Consider for example a complex query program that queries two online bookstores (cf. Section 2.4.2) and provides a summary over the prices for books in both book stores (*XML Query Use Cases*, *XMP-Q5*, [34]). An Xcerpt rule creating an HTML representation could look as follows (cf. also Section 4.7.1):

```
CONSTRUCT
  html [
    head [ title [ "Price Overview" ] ],
    body [
      table [
        tr [ td [ "Title" ], td [ "Price at A" ], td [ "Price at B" ] ],
        all tr [ td [ var T ], td [ var Pa ], td [ var Pb ] ]
      ]
    ]
  ]
FROM
   and {
     in {
       resource [ "file:bib.xml" ],
       bib [[
         book [[
           title [ var T  ],
           price [ var Pa ]
         ]]
       ]]
     },
     in {
       resource [ "file:reviews.xml" ],
       reviews [[
         entry [[
           title [ var T ],
           price [ var Pb ]
```

```
        ]]
      ]]
    }
  }
END
```

The query part of this rule consists of two query terms evaluated against the two data terms representing the bookstore databases. Note that both contain the variable `T` for binding the book title (thus both query terms need to match with books of the same title), whereas they differ in the variables used for binding the price.

Now assume that besides the HTML representation, it is also desirable to provide a representation suitable for mobile devices, e.g. in the format WML (*wireless markup language*[2]). This would require an additional Xcerpt rule with the same query part but different construct term. Assuming that the query part is complex, this approach is error prone and results in programs that are difficult to maintain as it contains many redundancies, and it is also more difficult to grasp the meaning of query programs. Using rule chaining, it is, however, possible to reuse the "complex" query part by separating it from the presentation and creating an intermediate representation for the data (in the example below: for each book, a `book-with-prices` term containing `title`, `price-a` and `price-b` subterms for the title, the price in the first bookstore and the price in the second bookstore). This "simpler" representation can then be queried by the two rules that create HTML and WML representations:

```
GOAL
  out {
    resource [ "file:prices.html" ,  "html" ],
    html [
      head [ title [ "Price Overview" ] ],
      body [
        table [
          tr [ td [  "Title"  ], td [  "Price at A"  ], td [  "Price at B"  ] ],
          all tr [ td [ var Title ], td [ var PriceA ], td [ var PriceB ] ]
        ]
       ]
     ]
   }
FROM
  books-with-prices [[
    book-with-prices [[
      title   [[ var Title  ]],
      price-a [[ var PriceA ]],
      price-b [[ var PriceB ]]
    ]]
  ]]
END

GOAL
  out {
    resource [ "file:prices.wml" ,  "xml" ],
    wml [
      all card [
        "Title: " , var Title ,
        "Price A: " , var PriceA,
        "Price B: " , var PriceB
```

[2] `http://www.wapforum.org/DTD/wml_1.1.xml`

```
      ]
    ]
  }
FROM
  books-with-prices [[
    book-with-prices [[
      title   [[ var Title  ]],
      price-a [[ var PriceA ]],
      price-b [[ var PriceB ]]
    ]]
  ]]
END

CONSTRUCT
  books-with-prices [
    all book-with-prices [
      title [ var T ],
      price-a [ var Pa ],
      price-b [ var Pb ]
    ]
  ]
FROM
  and {
    in {
      resource [ "file:bib.xml" ],
      bib [[
        book [[
          title [ var T  ],
          price [ var Pa ]
        ]]
      ]]
    },
    in {
      resource [ "file:reviews.xml" ],
      reviews [[
        entry [[
          title [ var T ],
          price [ var Pb ]
        ]]
      ]]
    }
  }
END
```

5.2 Querying the Web

Queries in the examples above considered mostly static content stored at one place and didn't take into account the dynamic and distributed nature of data on the Web. This Section illustrates on two scenarios how Xcerpt can be used to write queries to such Web data. The first scenario implements a Web service that generates a dynamic personal portal page, integrating news and weather information from the Web. The second scenario describes a (simple) Web crawler that can be used to traverse Web pages by following hyperlinks. For both approaches, backward chaining is preferable over forward chaining: in the first case, the queried data changes very frequently, which would require to update the portal even if noone is currently

viewing it; in the second case, the queried data is in the worst case the complete Web, which is obviously too large for a forward chaining evaluation.

5.2.1 Personal Portal Page: News and Weather[3]

The task of this scenario is to create a simple *personal portal page* Web service that integrates information from various Web sources, like news or weather. To this aim, this Section first describes rules for retrieving news and weather information from dynamically updated Web pages and then combines this information in an integrated portal page. The news and weather services are used as exemplary scenarios; other services are conceivable that can be queried and integrated in the same manner.

A salient aspect of this use case is that the queries are evaluated against other Web services; the queried data is highly dynamic, which requires to dynamically evaluate the queries when a user visits the personal portal page in a browser. Furthermore, the use of several rules for querying the various resources illustrates *separation of concerns* and provides a modular program design.

Querying Headlines of a News Ticker

Many media companies (like newspapers, magazines or television broadcasters) provide a so-called *news ticker* (or *news feed*) on their Web pages, which is constantly updated with the latest news and contains highly dynamic data. A common format used for representing news tickers is an XML application called *RDF site summary* (sometimes also called *rich site summary*) or *RSS*.[4] A typical RSS document could look as follows (the following excerpt is a news feed from the Swedish daily *Dagens Nyheter*):

```
<rss version="0.91">
  <channel>
    <title>Dagens Nyheter</title>
    <link>http://www.dn.se/</link>
    <description>
      De viktigaste nyheterna från Sveriges största morgontidning.
    </description>
    <language>sv-se</language>
    <image>
      <title>Dagens Nyheter</title>
      <url> http://www.dn.se/content/2/c4/13/99/logoDagensNyheter.gif</url>
      <link>http://www.dn.se/</link>
      <width>144</width>
      <height>18</height>
    </image>
    <item>
      <title>Dödligt gift i amerikanska senaten.</title>
      <link>http://www.dn.se/DNet/jsp/polopoly.jsp?d=145&amp;a=229799</link>
      <description>
        Det dödliga giftet ricin har påträffats i ett postrum i den amerikanske
        senaten i Washington. Ingen person har skadats.
      </description>
    </item>
    <item>
      <title>Ingen fågelinfluensa i Tyskland</title>
      <link>http://www.dn.se/DNet/jsp/polopoly.jsp?d=145&amp;a=229844</link>
      <description>
        Tester från Hamburg visar att de två kvinnor som misstänktes vara
        smittade av fågelinfluensan, som härjar i Asien, inte bar på smittan.
      </description>
    </item>
    <item>
      <title>Så påverkas du av partiernas familjepolitik</title>
      <link>http://www.dn.se/DNet/jsp/polopoly.jsp?d=145&amp;a=229781</link>
      <description>
        Med sin nya familjepolitik närmar sig kristdemokraterna folkpartiet
        och centern. Läs DN:s genomgång av de olika partiernas familjepolitiska
        förslag.
      </description>
    </item>
  </channel>
</rss>
```

[3] This use-case is available at `http://demo.xcerpt.org/cgi-bin/portal.xcerpt`

[4] `http://purl.org/rss/1.0/`

Querying this RSS document with Xcerpt is straightforward. The following Xcerpt rule provides an HTML view summarising the information contained in it. Note that the result of this rule is a `div` element that may be used in other rules to build a more complex page; it is not a complete HTML document itself.

```
CONSTRUCT
  div [
    attributes { id { "news" } },
    h1 [ "Channel:", var Channel ],
    all div [
      div [ attributes { class { "headline" } }, var Title ],
      div [ attributes { class { "abstract" } }, var Description ],
      div [ a [ attributes { href { var Link } }, "More ..." ] ]
    ]
  ]
FROM
  in {
    resource [ "http://www.dn.se/DNet/jsp/polypoly.jsp?d=1399" ],
    rss [[
      channel [[
        title [[ var Channel ]],
        item [[
          title [[ var Title ]],
          link [[ var Link ]],
          description [[ var Description ]]
        ]]
      ]]
    ]]
  }
END
```

With additional styling information (e.g. given in CSS), a result of this rule might look as in Figure 5.1 (snapshot taken on 09/06/2004). Since the data is highly dynamic, backward chaining is preferable for evaluation, i.e. the data is queried when the result is requested.

Querying Weather Information

Similarly, many weather services provide their information online in form of XML “feeds” that can be queried by Xcerpt. The following is a snapshot taken from `http://www.weatherroom.com`, which provides a wealth of information for display on the personal portal:

```
<WeatherFeed xmlns="http://www.weatherroom.com">
  <Current>
    <Location> Munich / Riem, Germany </Location>
    <RecordedAt> Munich / Riem, Germany </RecordedAt>
    <Updated> 950 AM GMT+1 WED JUN 9 2004 </Updated>
    <Conditions> Fair </Conditions>

    <Image>http://www.weatherroom.com/images/fcicons/fair.gif</Image>
    <Visibility>Mi</Visibility>
    <Temp>25°C</Temp>
    <Humidity>51%</Humidity>
    <Wind>W 10 MPH</Wind>
    <Barometer>30.21 in.</Barometer>

    <Dewpoint>14°C</Dewpoint>
    <HeatIndex>26°C</HeatIndex>
    <WindChill>25°C</WindChill>
```

Figure 5.1: Exemplary result of transforming an RSS feed to HTML with Xcerpt

```
    <Sunrise>4:13 AM GMT+1</Sunrise>
    <Sunset>8:11 PM GMT+1</Sunset>
    <MoonPhase>Last Quarter Moon</MoonPhase>
  </Current>

  <Forecast>
    <Date>+1 WED JUN 09 2004</Date>
    <Time>0500 AM GMT+1 WED</Time>
    <Afternoon> <Conditions>Partly Cloudy</Conditions> </Afternoon>
    <Evening>   <Conditions>Partly Cloudy</Conditions>  </Evening>
    <Overnight> <Conditions>Fair</Conditions>  </Overnight>
    <Morning>   <Conditions>Fair</Conditions>  </Morning>
  </Forecast>

  <Copyright>This feed is copyright 2004 weatherroom.com.</Copyright>
</WeatherFeed>
```

Similar in style to the “news ticker” rule, the following Xcerpt rule creates an HTML `div` element containing the current conditions (e.g. “Fair” or “Partly Cloudy”), the current temperature, and the current wind conditions from the “weather feed”. Obviously, other information could be retrieved as well (like the weather forecast).

```
CONSTRUCT
  div [
    attributes { id { "weather" } },
    h1 { "Weather for ", var Loc },
    div [
      table [
        tr [ td [ "Conditions" ], td [ var Conditions ] ],
        tr [ td [ "Temperature" ], td [ var Temp ] ],
        all tr [ td [ "Wind" ], td [ var Wind ] ]
      ]
    ],
    div [
      img { attributes { src { var Image } }
    ]
```

```
  ]
FROM
  in {
    resource [ "http://www.weatherroom.com/xml/ext/EDDM", "xml" ],
    WeatherFeed {{
      Current {{
        Location { var Loc },
        Temp { var Temp },
        Conditions { var Conditions },
        Image { var Image },
        Wind { var Wind }
      }}
    }}
  }
END
```

Creating a Combined Representation: The Personal Portal Page

Using rule chaining, the information from the news and weather query can be integrated easily into a common HTML page by using a single goal. The query of the goal contains two query terms used for querying the results of the two rules described above into the variables `News` and `Weather`. The construct term (head) of the goal provides the appropriate HTML framework. The element labelled `style` contains additional styling information e.g. expressed using *Cascading Style Sheets* (CSS, [115]), which is not given in the example.

```
GOAL
  html [
    head [
      title [ "Personal Portal" ]
      style [ ... ]
    ],
    body [
     h1 [ "Personal Portal" ],
     var News,
     var Weather
    ]
  ]
FROM
  and {
    var News → div {{ attributes {{ id { "news" } }} }},
    var Weather → div {{ attributes {{ id { "weather" } }} }}
  }
END
```

Combined with the two other rules, this goal can be used to implement a dynamic Web service that is evaluated every time a user points his browser to the portal page and requests “fresh” data from the dynamic resources. Figure 5.2 shows a sample evaluation of this program (on 09/06/2004).

5.2.2 Web Crawler

A Web crawler (sometimes also called “spider” in analogy to “Web”) is a program that visits Web pages and recursively follows hyperlinks to other Web pages. Web crawlers are e.g. used by search engines to index existing Web pages. Other applications include searching for a certain piece of information in a

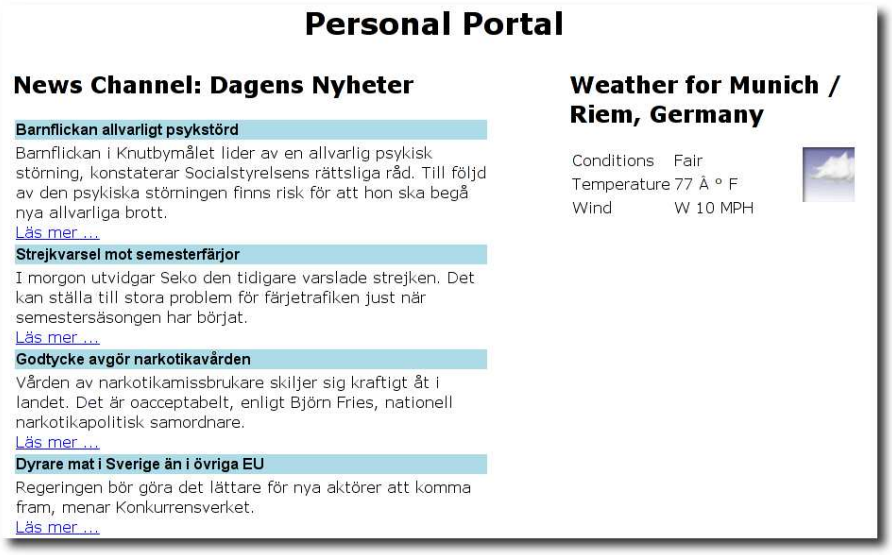

Figure 5.2: A sample evaluation of the “portal” program.

larger collection of Web pages, e.g. a complete Web site or even several Web sites. A rule-based language with recursion, like Xcerpt, is a natural choice for implementing a Web crawler. In the following, several examples based on a basic Web crawler are illustrated. For obvious reasons, only a backward chaining approach is feasible for evaluating such a crawler.

Note that the crawler, as it is implemented here, does not function on the current prototype (cf. Appendix A), as the latter does not support variables in resource specifications (for technical reasons) and provides no means to memoise sites that have already been visited (which, in case of cyclic hyperlinks, anticipates termination).

A Basic Web Crawler

The following Xcerpt program consisting of two rules illustrates the scheme for the traversal of hyperlinks on a very basic crawler, which only retrieves the URIs contained as hyperlinks in the pages it visits, grouped inside a `crawler` subterm. The first rule simply queries all hyperlinks (`a` subterms) on the page `http://www.xcerpt.org`, and serves as the base case for the evaluation. The second rule implements the recursive case: it first recursively calls the crawler for the URIs of all hyperlinks in visited pages and then queries these URIs again for hyperlinks, grouping them (and the URIs retrieved from the recursive call) inside the `crawler` subterm. Note that this crawler is not *grouping stratifiable* (cf. Section 6.4.1) and thus currently not covered by the formal semantics given in Chapter 7.

```
CONSTRUCT
  crawler {
    all link { var Link }
  }
FROM
  in {
    resource [ "http://www.xcerpt.org" ],
    desc a {{
      attributes {{ href { var Link } }}
    }}
  }
```

```
END

CONSTRUCT
  crawler {
    all link { var Link }
    all link { var RecLink }
  }
FROM
  and [
    crawler {{
      link { var RecLink }
    }},
    in {
      resource [ var RecLink ],
      desc a {{
        attributes {{ href { var Link } }}
      }}
    }
  ]
END
```

This crawler is very basic in several aspects, as it does not retrieve the content of visited Web pages and thus does not allow to search for anything beyond the URIs of visited pages. Also, it provides no means to check for cyclic structures of hyperlinks and thus might not terminate in such cases.

A Content Aware Crawler

The basic crawler can be extended to a more sophisticated crawler that also retrieves the content of Web pages (starting at `http://www.xcerpt.org`) in a straightforward manner as follows. Note that instead of the links contained in a page, the crawler now simply retrieves the complete content of pages, which is searched for `a` subterms in the recursive call (in the second rule). The result is a list of Web pages each wrapped inside a `page` subterm.

```
CONSTRUCT
  crawler {
    page {
      from { "http://www.xcerpt.org" },
      content { var Content }
    }
  }
FROM
  in {
    resource [ "http://www.xcerpt.org" ],
    var Content → html {{ }}
  }
END

CONSTRUCT
  crawler {
    all var RecPage,
    all page {
      from { var Link },
      content { var Content }
```

```
    }
  }
FROM
  and {
    crawler {{
      var RecPage → page {{
        content {{
          desc a {{ attributes {{ href { var Link } }} }}
        }}
      }}
    }},
    in {
      resource [ var Link ],
      var Content → html {{ }}
    }
  }
END
```

Searching a Web of Pages

The content-aware crawler may be used to implement various retrieval tasks. For example, the following goal retrieves the URIs of all Web pages that contain an `h1` element containing the word "XML" and that are reachable from the site `http://www.xcerpt.org` by chaining with the rules of the content-aware crawler:

```
GOAL
  results {
    all var URI
  }
FROM
  crawler {{
    page {{
      from { var URI },
      content {{
        desc h1 {{ /.*XML.*/ }}
      }}
    }}
  }}
END
```

Representing a Web of Pages as Nested Xcerpt Terms

The content-aware crawler simply creates a list of Web pages and ignores the "hyperlink graph" that connects these pages. It might, however, be desirable to represent this graph as a nested Xcerpt term, e.g. to easily search over the hyperlink structure. The following additional rules convert the result of the content-aware crawler into such a nested tree structure (a graph structure would be conceivable, but is more complicated). The first rule retrieves all "leaves" (i.e. such subterms that do not contain a hyperlink). The second rule recursively retrieves (nested) pages (beginning with the "leaves") and queries the content-aware crawler for all pages that refer to these pages. A nested structure is constructed in the head of this rule.

```
CONSTRUCT
  pages {
```

```
    var Page
  }
FROM
  crawler {{
    var Page → page {{
      content {{
        without desc a {{ }}
      }}
    }}
  }}
END

CONSTRUCT
  pages {
    page {
      from { var From },
      content { var Content },
      pages {
        all var Page
      }
    }
  }
FROM
  and {
    pages {{
      var Page → page {{
        from { var Link }
      }}
    }},
    crawler {{
      page {{
        from {{ var From }}
        content {{
          var Content → desc a {{ attributes {{ href { var Link } }} }}
        }}
      }}
    }}
  }
END
```

Using the nested structure constructed by these rules, it is possible to express queries to the hyperlink structure of the crawled Web pages. For example, the following goal retrieves pages that are reachable by pages containing the word “XML”:

```
GOAL
  result {
    all var URI
  }
FROM
  pages {
    desc page {{
      content {{ desc /.*XML.*/ }},
      pages {{
```

```
        desc page {{
           from { var URI }
        }}
      }}
    }
  }
END
```

5.3 Semantic Web Reasoning

The Semantic Web aims at enriching Web data with meta-data (and even meta-meta-data), allowing retrieval of data while respecting available "semantic" information. A query language for such data needs to be able to query both standard (XML) data and meta-data, and furthermore needs to provide reasoning capabilities that go beyond simple retrieval (cf. "Reasoning Capabilities", Section 1.3.8). In the current state of the Semantic Web, meta-data is usually expressed in some kind of ontology language (e.g. OWL, [118]), which essentially allows to describe a hierarchy or network of concepts and properties of these concepts. The two most common reasoning tasks are to determine subconcepts/superconcepts (e.g. to infer that "fiction book" is a subconcept of "book"), and to test whether a given object (i.e. Web resource) is an instance of a particular concept (e.g. to infer that a certain book about the "Viking Age" is a "history book").

This section illustrates the use of Xcerpt for querying XML data together with Semantic Web data. To this aim, a small example called the *Clique of Friends* (Section 5.3.1) is first used to illustrate some basic reasoning (mainly the *transitive closure* of a relation) for the Semantic Web. This example does not use any particular Semantic Web language itself. Building on these concepts, a more complex scenario is introduced (Section 5.3.2), which illustrates querying a collection of books in the presence of a simple book ontology defined in the language OWL [118].

5.3.1 Clique of Friends

Consider a collection of address books where each address book has an owner and a set of entries, some of which are marked as "friend" to indicate that the person associated with this entry is considered a friend by the owner of the address book. In XML, this collection of address books can be represented in a straightforward manner as follows:

```
<address-books>
  <address-book>
    <owner>Donald Duck</owner>
    <entry>
      <name>Daisy Duck</name>
      <friend/>
    </entry>
    <entry>
      <name>Scrooge McDuck</name>
    </entry>
  </address-book>

  <address-book>
    <owner>Daisy Duck</owner>
    <entry>
      <name>Gladstone Duck</name>
      <friend/>
    </entry>
```

```
    <entry>
      <name>Ratchet Gearloose</name>
      <friend/>
    </entry>
  </address-book>
</address-books>
```

In this example, the collection contains two address books, the first owned by “Donald Duck” and the second by “Daisy Duck”. Donald’s address book has two entries, one for “Scrooge”, the other for “Daisy”, and only “Daisy” is marked as “friend”. Daisy’s address book again has two entries, both marked as “friend”.

The *clique-of-friends* of Donald is the set of all persons that are either direct friends of Donald (i.e. in the example above only “Daisy”) or friends of a friend (i.e. “Gladstone” and “Ratchet”), or friends of friends of friends (none in the example above), and so on. As a first step towards this clique of friends, the following Xcerpt rule defines the relation `friend-of` as a view over the address book collection.[5]

```
CONSTRUCT
  friend-of [ var X, var Y ]
FROM
  in {
    resource [ "file:address-books.xml" ],
    address-books {{
      address-book {{
        owner { var X },
        entry {{
          name { var Y },
          friend {}
        }}
      }}
    }}
  }
END
```

Note that it would be easy to define the relation `friend-of` as reflexive by simply using curly braces instead of square brackets in the construct term.

Defining the transitive closure of the relation requires a recursive rule, but is pretty straightforward otherwise: a person `Y` is a `friend-of-friend` of some person `X`, if `Y` is either a `friend-of X` or there exists a person `Z` that is a `friend-of X`, such that `Y` is a `friend-of-friend` of `Z`:

```
CONSTRUCT
  friend-of-friend [ var X, var Y ]
FROM
  or {
    friend-of [ var X, var Y ],
    and {
      friend-of [ var X, var Z ],
      friend-of-friend [ var Z, var Y ]
    }
  }
END
```

[5] Note that, in contrast to other logic programming approaches, interpreting the term labelled `friend-of` as a relation is only valid with respect to the application at hand; in general, `friend-of` is just a term.

Finally, the `clique-of-friends` is simply the collection of all `friend-of-friend` relationships. It is constructed in a goal as follows:

```
GOAL
  clique-of-friends {
    all var FOF
  }
FROM
  var FOF → friend-of-friend {{ }}
END
```

Although this example does not make use of Semantic Web data, the relationship to reasoning is obvious. The book ontology below uses very similar rules for defining a relation `subclass-of` that may be used to collect sub- or superconcepts.

5.3.2 Ontology Reasoning: The Book Ontology

Consider a book store (like the two book stores of Section 2.4.2) that provides an online catalogue containing the books it offers. Searching a book usually requires searching the book titles and maybe abstracts of the content. If a customer wants to search by topic rather than by title (e.g. “history books”), this kind of search usually misses many of the relevant entries and yields a large number of false positives. For example, the book entitled “Folket i Birka” (Swedish: “The People of Birka”, a historical novel for children illustrating the life of people in a Viking Age town) is only found when searching for “Birka”, which already requires much knowledge over the domain of interest. A “semantic” query would be able to include the book “Folket i Birka” when searching for books about the “Viking Age” or “History Books for Children” without requiring to include more specific search parameters.

The Book Ontology

Using the Semantic Web, such semantic queries become feasible. The online book store might provide an ontology describing the relations between different categories of books, and the properties of these categories. The following example uses the Web Ontology Language *OWL* [118] for describing a simple part of this book ontology:

```
<rdf:RDF xmlns:owl  = "http://www.w3.org/2002/07/owl#"
         xmlns:rdf  = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"       >

  <owl:Class rdf:ID="Book"/>

  <owl:Class rdf:ID="Novel">
    <rdfs:label>Novel</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Book"/>
  </owl:Class>

  <owl:Class rdf:ID="History">
    <rdfs:label>History Book</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Book"/>
  </owl:Class>

  <owl:Class rdf:ID="Classic_History">
    <rdfs:label>Book about Classic History</rdfs:label>
    <rdfs:subClassOf rdf:resource="#History"/>
```

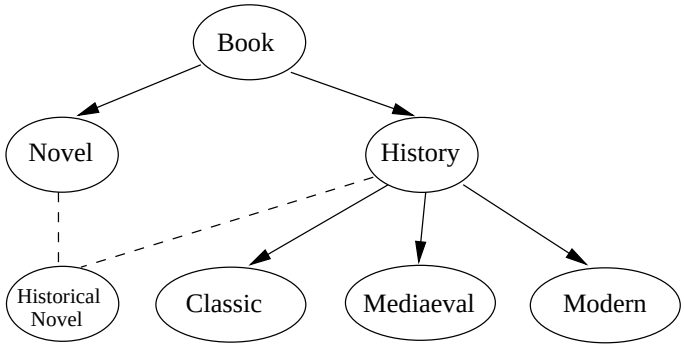

Figure 5.3: Part of a book ontology for an online book store. Solid lines indicate subconcepts, dotted lines intersection of concepts.

```
  </owl:Class>

  <owl:Class rdf:ID="Mediaeval_History">
    <rdfs:label>Book about Mediaeval History</rdfs:label>
    <rdfs:subClassOf rdf:resource="#History"/>
  </owl:Class>

  <owl:Class rdf:ID="Modern_History">
    <rdfs:label>Book about Modern History</rdfs:label>
    <rdfs:subClassOf rdf:resource="#History"/>
  </owl:Class>

  <owl:Class rdf:ID="Historical_Novel">
    <rdfs:label>Historical Novel</rdfs:label>
    <owl:intersectionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#Novel"/>
      <owl:Class rdf:about="#History"/>
    </owl:intersectionOf>
  </owl:Class>

</rdf:RDF>
```

This ontology describes the following hierarchy of concepts (cf. Figure 5.3):

- a *Novel* and a *History Book* is a *Book*
- books about *Classic History*, *Mediaeval History*, and *Modern History* are *History* books.
- *Historical Novel* is the intersection of *History Book* and *Novel* (both referenced by `rdf:about`). Note that intersection is stronger than simply being the subconcepts of two concepts.

Note that OWL ontologies may be serialised in XML in many different manners, e.g. using nested `owl:Class` definitions.

Subclass Checking with Xcerpt

Using the rules for transitive closure of the *Clique of Friends*, checking subclasses in this hierarchy of concepts is straightforward. The following Xcerpt program defines a relation `subclass-of` that relates concepts with all parent concepts based on the serialisation of the book ontology above. The first rule defines

a relation `immediate-subclass-of`, which provides a simplified view on the ontology data and relates a concept (variable *X*) with its immediate parent concepts (variable *Y*). The second rule defines `subclass-of` as the transitive closure over `immediate-subclass-of` (note the similarity with the `friend-of-friend` rule in the previous section). Note also the use of XML namespaces in this program.

```
ns-prefix owl  = "http://www.w3.org/2002/07/owl#"
ns-prefix rdf  = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
ns-prefix rdfs = "http://www.w3.org/2000/01/rdf-schema#"

CONSTRUCT
  immediate-subclass-of [ var X, var Y ]
FROM
  in {
    resource [ "file:books.owl" ],
    rdf:RDF {{
      var X → owl:Class {{
        rdfs:subClassOf {{
          attributes {{ rdf:resource { /#(var YRef →.*)/ } }}
        }}
      }},
      var Y → owl:Class {{
        attributes {{ rdf:ID { var YRef } }}
      }}
    }}
  }
END

CONSTRUCT
  subclass-of [ var X, var Y ]
FROM
  or {
    immediate-subclass-of [ var X, var Y ],
    and {
      immediate-subclass-of [ var X, var Z ],
      subclass-of [ var Z, var Y ]
    }
  }
END
```

Checking for all parent concepts or child concepts of a specific concept is now easy. For example, the following goal retrieves all child concepts of the concept with `rdfs:label` “History Book” by chaining with the rules above:

```
ns-prefix owl  = "http://www.w3.org/2002/07/owl#"
ns-prefix rdfs = "http://www.w3.org/2000/01/rdf-schema#"

GOAL
  subconcepts {
    all var Concept
  }
FROM
  subclass-of [
    var Concept,
```

```
    owl:Class {{
      rdfs:label { "History Book" }
    }}
  ]
END
```

Annotating Books with Meta-Data

To add "semantic" meta-data to the XML document used for representing the book store database, it is necessary to annotate the data as follows. Each book is given a unique `rdf:ID`, and relationships between books (identified by the value of `rdf:ID` and concepts in the ontology are established (using `rdf:type`). Changes to the original document are indicated by red colour:

```
<bib xmlns:owl  = "http://www.w3.org/2002/07/owl#"
     xmlns:rdf  = "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <book year="1995" rdf:ID="vikinga_blot">
    <title>Vikinga Blot</title>
    <authors>
      <author>
        <last>Ingelman-Sundberg</last>
        <first>Catharina</first>
      </author>
    </authors>
    <publisher>Richters</publisher>
    <price>5.95</price>
  </book>
  <book year="1998" rdf:ID="boken_om_vikingarna">
    <title>Boken Om Vikingarna</title>
    <authors>
      <author>
        <last>Ingelman-Sundberg</last>
        <first>Catharina</first>
      </author>
    </authors>
    <publisher>Prisma</publisher>
    <price>22.95</price>
  </book>
  <book year="1999" rdf:ID="folket_i_birka">
    <title>Folket i Birka på Vikingarnas Tid</title>
    <authors>
      <author>
        <last>Wahl</last>
        <first>Mats</first>
      </author>
      <author>
        <last>Nordqvist</last>
        <first>Sven</first>
      </author>
      <author>
        <last>Ambrosiani</last>
        <first>Björn</first>
      </author>
    </authors>
    <publisher>BonnierCarlsen</publisher>
    <price>39.95</price>
  </book>
  <book year="1997" rdf:ID="vikingar_i_österled">
    <title>Vikingar i Österled</title>
    <editor>
      <last>Larsson</last>
      <first>Mats</first>
      <affiliation>Lunds universitet</affiliation>
    </editor>
    <publisher>Atlantis</publisher>
    <price>49.95</price>
  </book>

  <owl:Thing rdf:about="#vikinga_blot">
    <rdf:type rdf:resource="#Mediaeval_History"/>
    <rdf:type rdf:resource="#Novel"/>
  </owl:Thing>

  <owl:Thing rdf:about="#boken_om_vikingarna">
```

```
    <rdf:type rdf:resource="#Mediaeval_History"/>
  </owl:Thing>

  <owl:Thing rdf:about="#folket_i_birka">
    <rdf:type rdf:resource="#Mediaeval_History"/>
    <rdf:type rdf:resource="#Novel"/>
  </owl:Thing>

  <owl:Thing rdf:about="#vikingar_i_österled">
    <rdf:type rdf:resource="#Mediaeval_History"/>
  </owl:Thing>

</bib>
```

Although the construct `owl:Thing` looks strange, it is required by the OWL specification when describing resources defined elsewhere. Note that a book is only associated with the most specific concepts it belongs to: the book "Folket i Birka" has the `rdf:ID folket_i_birka` and belongs to the concepts `Mediaeval_History` and `Novel`, but not explicitly to the concepts `History` or `Book`.

Ontology Reasoning: Querying by Topic

Using the book ontology, the Xcerpt rules for checking subconcepts, and the extended book document, it is now possible to perform "semantic" queries as described in the introduction. Consider a customer that is interested in "History Books", i.e. all books belonging to either the concept itself, or to any subconcept. In Xcerpt, a query for such books can be expressed by the following rule (namespace prefixes are omitted for brevity but are defined as above):

```
CONSTRUCT
  history_books {
    all var Book
  }
FROM
  and {
    in {
      resource [ "file:bib.xml" ],
      bib {{
        var Book → book {{
          attributes {{ rdf:ID { var ID } }}
        }},
        owl:Thing {{
          attributes {{ rdf:about { /#(var ID →.*)/ } }},
          rdf:type {{
            attributes {{ rdf:resource { /#(var Class →.*)/ } }},
          }}
        }}
      }
    },
    subclass-of [
      owl:Class {{
        attributes {{ rdf:ID { var Class } }}
      }},
      owl:Class {{
        rdfs:label { "History Book" }
      }}
    ]
  }
END
```

This rule queries the extended `bib.xml` document for all books, and retrieves the respective classes they belong to (by querying the `owl:Thing` subterms describing the `rdf:ID` of the books). By querying the results of the rule defining the relation `subclass-of`, it is then verified whether the book is indeed a "History Book".

The rule above can be generalised for arbitrary instance tests in a straightforward manner. The following Xcerpt rule defines a relation `belongs-to` that explicitly associates a book with all concepts ("classes") it belongs to. Note that both variables `Class` and `SuperClass` are grouped inside the `classes` subterm, as some books might yield more than one binding for the variable `Class`.

```
CONSTRUCT
  belongs-to [
    var Book
    classes {
      all var Class,
      all var SuperClass
    }
  ]
FROM
  and {
    in {
      resource [ "file:bib.xml" ],
      bib {{
        var Book → book {{
          attributes {{ rdf:ID { var ID } }}
        }},
        owl:Thing {{
          attributes {{ rdf:about { /#(var ID →.*)/ } }},
          rdf:type {{
            attributes {{ rdf:resource { /#(var Class →.*)/ } }},
          }}
        }}
      }
    },
    subclass-of [
      owl:Class {{
        attributes {{ rdf:ID { var Class } }}
      }},
      var SuperClass
    ]
  }
END
```

Note that this rule is not capable of inferring that a concrete book that is a "Novel" about "Medieaeval History" is also a "Historical Novel", as it lacks support for OWL's intersection construct.

Ontology Reasoning: Intersection

Recall that the ontology used in this Section also contains the concept "Historical Novel", which is defined as the intersection of "History Book" and "Novel". Whereas querying for "History Book" is rather straightforward, querying for "Historical Novel" requires more complex rules taking into account the intersection of concepts.

The following Xcerpt rule builds upon the generic `belongs-to` relation to also include concepts that contain intersections. For each book, it creates a `belongs-to-extended` term containing the book and all

concepts (contained in the subterm labelled `classes`), including those defined by the intersection of other concepts:

```
CONSTRUCT
  belongs-to-extended {
    var Book,
    classes { all var Class, var ISClass }
  }
FROM
  and {
    belongs-to [ var Book, classes {{ var Class }} ],
    in {
      resource [ "file:books.owl" ],
      rdf:RDF {{
        var ISClass → owl:Class {{
          owl:intersectionOf {{ }}
        }}
      }}
    },
    not {
      and {
        in {
          resource [ "file:books.owl" ],
          rdf:RDF {{
            var ISClass → owl:Class {{
              owl:intersectionOf {{
                owl:Class {{
                  attributes {{ rdf:about { var CRef } }}
                }}
              }}
            }},
            var SomeClass → owl:Class {{
              attributes {{ rdf:ID { var CRef } }}
            }}
          }}
        },
        belongs-to [ var Book, classes {{ without var SomeClass }} ],
      }
    }
  }
END
```

This rather complex rule is evaluated as follows. The first two query terms of the conjunction in the query part retrieve books with associated concepts (variables `Book` and `Class`), and possible candidate concepts defined by intersection (variable `ISClass`). The last part of this conjunction is a negated subquery that checks whether all of the concepts used in the definition of `ISClass` (bound to the variable `SomeClass` by dereferencing using the variable `CRef`) are associated with the book bound to `Book`, i.e. there does not exist a concept of `ISClass` bound to `SomeClass` that is not contained in the concepts associated with `Book`. Note that, due to range restrictedness, it is necessary to query both the ontology and the `belongs-to` relation twice; otherwise, the variables `Book` and `ISClass` would not yield bindings.

CHAPTER

SIX

Range Restrictedness, Standardisation Apart, and Stratification

This Chapter discusses syntactic restrictions to which Xcerpt programs in this thesis are considered to conform. These restrictions either simplify the formal semantics in Chapter 7, or avoid programming mistakes, or both. All of them are purely syntactical properties that they can be verified statically when programs are parsed.

Range Restrictedness[1] (Section 6.2) is a restriction on the kinds of admissible rules and goals. Rules/-Goals that are range restricted do not contain variables in the construct part that are not "justified" (i.e. bound by a non-negated query term) in the query part. Range restricted programs can be evaluated in both a backward chaining and a forward chaining fashion, whereas programs that are not range restricted often are difficult to treat in forward chaining algorithms, because results of a rule are not necessarily ground.

Stratification (Section 6.4) is a further restriction on programs that contain the grouping constructs `all` and `some` or the negation constructs `not` and `without`. In stratified programs, negation is only allowed if it does not affect recursive rule evaluations. Stratification avoids many of the problems that come with non-monotonic negation.

As this thesis does not intend to cover the wide areas of non-monotonic negation and different approaches to forward chaining, stratification and range restrictedness are suitable assumptions for the formalisation of Xcerpt as it is presented here. Other, less rigid, approaches are feasible and not excluded by Xcerpt *per se*.

6.1 Preliminaries

In subsequent chapters, the following notations are used to simplify the discussion over the semantics of Xcerpt programs:

- *Programs* are sets of rules (and goals), usually denoted by $P = \{R_1, \ldots, R_n\}$.
- *Rules* are denoted by $R = t^c \leftarrow \mathcal{Q}$, where t^c is the construct term of the rule and $\mathcal{Q}$ the query part; the set of rules of a program P is usually denoted by $\mathcal{R} \subseteq P$
- *Goals* are denoted by $G = t^c \leftarrow_g \mathcal{Q}$, where t^c is the construct term of the rule and $\mathcal{Q}$ the query part; the set of goals of a program P is usually denoted by $\mathcal{G} \subseteq P$
- *Queries* of the form `and`$\{Q_1, \ldots, Q_n\}$ are sometimes denoted by $Q_1 \wedge \cdots \wedge Q_n$ or by $\bigwedge_{1 \leq i \leq n} Q_i$; likewise, queries of the form `or`$\{Q_1, \ldots, Q_n\}$ are sometimes denoted by $Q_1 \vee \cdots \vee Q_n$ or by $\bigvee_{1 \leq i \leq n} Q_i$ and queries of the form `not Q` are sometimes denoted by $\neg Q$.
- *Resources* are considered to be internalised, i.e. it is assumed that any data term referred to by an input resource specification of the form `in{...}` is part of the program; this assumption simplifies the formal treatment below and can be implemented in a straightforward (but inefficient) manner.

[1] Many publications, e.g. [71] and [98], refer to *range restricted* programs as *allowed* or *safe* programs

- The set $\mathcal{T}$ denotes the set of all terms, $\mathcal{T}^q \subsetneq \mathcal{T}$ the set of all query terms, $\mathcal{T}^g \subsetneq \mathcal{T}^q$ the set of all ground query terms, and $\mathcal{T}^d \subsetneq \mathcal{T}^g$ the set of all data terms.

In most parts of the formalisation, rules and goals are not distinguished unless explicitly mentioned; this simplification is useful as rules and goals are very similar. Also, parts of the formalisation use a simplified term representation, where strings and regular expressions are simply treated as compound terms with empty content and total term specification. E.g. the string `"XML"` is represented as `"XML"{}`.

6.2 Range Restrictedness

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

The following sections give a formal, syntactic criterion for range restrictedness, which considers negated queries and optional subterms as described in Sections 4.3.4, 4.3.1 and 4.5.3, as well as disjunctions in rule bodies.

6.2.1 Polarity of Subterms

So as to determine whether a rule is range restricted, variable occurrences in query and construct terms are associated with the polarities *positive* (+) or *negative* (−), and the attributes *optional* (?) or *not optional* (!) for such variables that are contained within an optional subtree and thus are not bound in all valid matchings. Intuitively, a *negative* variable occurrence is a *defining* occurrence, whereas a *positive* variable occurrence is a *consuming* occurrence. Since most terms are considered to be not optional, the attribute ! is omitted in most examples.

The polarity of variable occurrences in a term can be determined by recursively attributing all subterms of a term.

Definition 6.1 (Polarity of Subterms)

1. Let t be a query term with polarity p and optionality o.
   - if t is of the form `without` t', then t' is of polarity + (regardless of p) and optionality o
   - if t is of the form `optional` t', then t' is of polarity p and optionality ?.
   - if t is of one of the forms $l\{\{t'_1,\ldots,t'_n\}\}$, $l\{t'_1,\ldots,t'_n\}$, $l[[t'_1,\ldots,t'_n]]$ or $l[t'_1,\ldots,t'_n]$ ($n \geq 0$), then $t'_1, \ldots, t'_n$ are of polarity p and optionality o.
   - if t is of the form `desc` t' then t' is of polarity p and optionality o.
   - if t is of the form *var* $X \rightarrow t'$ then t' is of polarity p and optionality o.
2. Let t be a construct or data term with polarity p and optionality o.
   - if t is of the form `optional` t', then t' is of polarity p and optionality ?.
   - if t is of one of the forms $f\{t'_1,\ldots,t'_n\}$ or $f[t'_1,\ldots,t'_n]$ ($n \geq 0$), then $t'_1, \ldots, t'_n$ are of polarity p and optionality o.
   - if t is of the forms `all` t' or `some` t', then t' is of polarity p and optionality o.
   - if t is of the form $op(t'_1,\ldots,t'_n)$, with op a function or aggregation identifier, then $t'_1, \ldots, t'_n$ are of polarity p and optionality o.

The root of a query term is usually of negative polarity (and thus define variable bindings), as query terms usually occur in rule bodies. The root of a construct or data term is usually of positive polarity.

Example 6.1 (Polarities within a Term)
The following figure gives the polarities for a query term (the polarity of the root node is thus -) querying a student database for such students that have not submitted exercise E. Some students might not have an student id ("matrnr"), indicated by the keyword `optional`.

```
-students {{
  -student {{
    -name { -var N },
    -optional -?matrnr { -?var MNr },
    -exercises {{
      -without +exercise {{ +nr { +var E } }}
     }}
   }}
 }}
```

Note that both variables N and MNr occur negatively (and thus define variable bindings), while variable E occurs positively (and thus does not define a variable binding). Furthermore, variable MNr is attributed as optional.

In a rule, the construct term in the head always has positive polarity and the query part has negative polarity and both are, by default, not optional. If negation constructs occur, the polarity changes according to Definition 6.1. Furthermore, if parts of a query are negated by `not`, the polarity of these parts is again positive:

Definition 6.2 (Polarity in Rules)

1. If $R = t^c \leftarrow Q$ is a rule or goal with t^c a construct term and Q a query part, then the polarity of t^c is $+$ and the polarity of Q is $-$.
2. Let Q be a query part with polarity p.
   - if Q is of the form *not* Q', then Q' is of polarity $+$ (regardless of p)
   - if Q is of the forms *and*$\{Q_1,\ldots,Q_n\}$, *and*$[Q_1,\ldots,Q_n]$, *or*$\{Q_1,\ldots,Q_n\}$ or *or*$[Q_1,\ldots,Q_n]$, then $Q_1, \ldots, Q_n$ are of polarity p
   - if Q is of the form *in*$\{R, Q'\}$, with R a resource specification and Q' a query, then Q' is of polarity p and R is of polarity $+$ (regardless of p)
   - if Q is of the form t (a query term), then t is of polarity p.

Note that the polarity of negated subterms and queries is *always* positive, regardless of the level of nesting. The rationale behind this is that, since negation in Xcerpt is *negation as failure* and not the negation of classic logic, additional negations do not completely revert previous negations. Variable occurrences that are in the scope of at least one negation construct are always consuming occurrences, since negation as failure requires to perform auxiliary computations.

Example 6.2 (Polarities in a Rule)
Consider the following example of a rule, which is intended to create a list of students which have not submitted exercise 2.

```
CONSTRUCT
+not_submitted {
  +all +student {
    +name { +var N },
    +matrnr {
      +optional +?var MNr
         with default +"N.N."
    }
  }
}
FROM
-students {{
  -student {{
    -name { -var N },
    -optional -?matrnr { -?var MNr },
    -exercises {{
      -without +exercise {{ +nr { +2 } }}
     }}
   }}
 }}
END
```

The variables N and MNr occur both positively and negatively. The variable MNr is in addition attributed as optional, as it is contained in a subterm that is optional.

6.2.2 Range Restrictedness

Range restrictedness requires that in a rule, for each consuming occurrence of a variable, there exists at least one defining occurrence. Furthermore, a variable for which all defining occurrences are optional also needs to be optional on all consuming occurrences. This restriction is straightforward to understand, as it just requires that "each variable in the head or in a negated query needs to be bound elsewhere".

This intuitive definition of range restrictedness is complicated by the possibility of disjunctions in the rule body, in which case a variable occurring positively in the rule head needs to occur negatively in *each* disjunct. Since disjunctions can also be nested, it is useful to define a *disjunctive rule normal form*:

Definition 6.3 (Disjunctive Rule Normal Form)

1. A rule or goal $t^c \leftarrow Q$ is in disjunctive rule normal form, iff the query part Q is in disjunctive normal form, where t^c is a construct term, possibly associated with an output resource.

2. A query part Q is in disjunctive normal form, iff it has the form $Q_1 \vee \cdots \vee Q_n$ ($n \geq 0$) such that each of the Q_i has the form $t_1^q \wedge \cdots \wedge t_m^q \wedge \neg t_{m+1}^q \cdots \wedge \neg t_{m+k}^q$ ($m \geq 0, k \geq 0$) with t_j^q being query terms, possibly associated with input resources.

Note that, although rules are disjunctive like in other logic programming languages, the disjunctions cannot be factored out by splitting a rule in two, as the rule head might contain the *all* construct which collects all alternative bindings and is thus obviously influenced by the disjunctions in the rule body.

Proposition 6.4
Every rule can be transformed into disjunctive rule normal form.

Proof Sketch. Transformation is straightforward and follows mostly the known transformation rules for formulas in first order languages. Resource specifications can be distributed to query terms by simply making explicit their scope using a recursive traversal over the formula.

□

Range restrictedness requires that each variable that occurs positively in one of the disjuncts occurs also negatively in the same disjuncts. Range restrictedness is formalised by the following definition:

Definition 6.5 (Range Restrictedness)
Let R be a rule or goal and let $R' = t^c \leftarrow Q_1 \vee \cdots \vee Q_n$ ($n \geq 0$) be the disjunctive rule normal form of R. R is said to be *range restricted*, iff

1. for each disjunct Q_i ($1 \leq i \leq n$) holds that each variable occurring with positive polarity in either t^c or Q_i also occurs at least once with negative polarity in Q_i.

2. each variable attributed as *optional* and with *negative polarity* in at least one of the Q_i ($1 \leq i \leq n$), and without another non-optional, negative occurrence in Q_i, is also attributed as optional in all positive occurrences in Q_i and t^c.

A program P is called *range restricted*, if all rules $R \in P$ are range restricted.

Example 6.3 (Range Restrictedness)
Consider the following rule, which is a slight modification of Example 6.2 and is intended to retrieve such students that have not submitted exercise E. The rule is not range restricted as the variable E occurs only with positive polarity, and the variable MNr is not attributed as optional in the rule head:

```
CONSTRUCT
+not_submitted {
  +all +student {
    +name { +var N },
    +matrnr {
      +var MNr
    }
  }
}
```

```
FROM
-students {{
  -student {{
    -name { -var N },
    -optional -?matrnr { -?var MNr },
    -exercises {{
      -without +exercise {{ +nr { +var E } }}
    }}
  }}
}}
END
```

An interesting example of a somewhat strange but nonetheless range restricted program is the following rule, in which two query terms mutually define one variable while consuming another:

Example 6.4

```
CONSTRUCT
  +f{ +var X, +var Y }
FROM
  -and {
    -g{ -var X, -without +k{ +var Y } },
    -h{ -var Y, -without +l{ +var X } },
  }
END
```

Note that the first query term has a negative (defining) occurrence of the variable X and a positive (consuming) occurrence of the variable Y, while the second term has a negative (defining) occurrence of the variable Y and a positive (consuming) occurrence of the variable X. Although this example looks strange, it is, by definition, range restricted.

It might be argued that such programs are not range restricted because the defining occurrences of the variables are in a mutual lock and that the query part of this rule thus cannot be evaluated. However, such programs do not have the problems that range restrictedness aims to solve: they can be evaluated in a forward chaining manner, and all variable bindings are finite and justified by a query.

6.3 Standardisation Apart (or Rectification)

In Xcerpt, rules are standardised apart (or *rectified*). Informally, this means that rules are considered to be variable disjoint and all variables occurring in a rule are restricted to a single rule instance, i.e. each recursive application of the rule uses "different" or "fresh" variables. In an implementation, standardisation apart can easily be achieved by simply renaming the variables for each instance using fresh, otherwise unused variable names. In the formalisation below, standardisation apart is realised by treating every rule as universally closed.

Standardisation apart is an important property for rule-based languages, as it ensures a certain amount of "locality": otherwise, in each rule it would be necessary to consider all variable occurrences in the complete program, which yields programs that are very difficult to maintain. Furthermore, recursion is only reasonably defined if each instance of a rule in a recursive rule chain uses different variables.

6.4 Stratification for Grouping Constructs and Negation

Stratification is a technique first proposed by Apt, Blair, and Walker [8] to define a class of logic programs where non-monotonic features like Xcerpt's grouping constructs or negation can be defined in a declarative manner. The principal idea of stratification is to disallow programs with a recursion over negated queries or grouping constructs and thereby precluding undesirable programs. While this requirement is very strict (e.g. the Web crawler of Section 5.2.2 is not stratifiable), its advantages are that it is straightforward to

understand and can be verified by purely syntactical means without considering terms that are not part of the program. Several refinements over stratification have been proposed, e.g. *local stratification* [87] that allow certain kinds of recursion, but these usually require more "knowledge" of the program or the queried resources.

Xcerpt programs in this thesis are considered to be stratifiable[2]. Furthermore, the notion of stratification is not only used for proper treatment of negation, it also extends to rules with grouping constructs, because a recursion over grouping constructs usually defines undesirable behaviour. In this section, this new, so-called *grouping stratification* is introduced first (Section 6.4.1). Subsequently, *negation stratification* is introduced in accordance with the definition given in [8] (Section 6.4.2), and both stratifications are combined to so-called *full stratification* of Xcerpt programs (Section 6.4.3).

6.4.1 Grouping Stratification

The grouping constructs `all` and `some` are powerful constructs that are justified by many practical applications (cf. Chapter 5). However, using them in recursive rules allows to define programs with no useful meaning. Consider for example the program

$f\{all\ var\ X\} \leftarrow f\{\{var\ X\}\}$
$f\{a\}$

The meaning of such programs is often unclear and unintended by the program author. Besides this issue, rules with grouping constructs usually require that all rules they depend on are completely evaluated, because `all` expresses to collect *all* possible results, and a too early evaluation of a grouping construct might yield terms that do not properly reflect the meaning of the grouping constructs. Consider the program

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

Obviously, the evaluation of the first rule depends on the evaluation of the second rule, and because the first rule expresses to collect all subterms of a g, it is necessary to defer the evaluation of the first rule until the second rule is evaluated, although the rule body of the first rule would already match with $g\{a\}$.

In this thesis, the solution to both issues is to disallow recursion of rules with grouping constructs, and to require that all rules on which a rule with grouping constructs depends can be evaluated first. This property can be verified syntactically by *stratifying* a program with so-called *grouping stratification*. If a program is not *grouping stratifiable*, it might (but not necessarily does) contain problematic rules.

Informally, the grouping stratification is rather straightforward: for a program, create a dependency graph between rules, i.e. a multi-graph where vertices represent rules and edges represent the possibility that one rule calls the other, and mark those edges where a rule with grouping construct in the rule head depends on any other rule. A program is called *stratifiable*, if it is possible to partition the dependency graph into disjunctive layers (so-called *strata*) such that the rules in each layer only have unmarked dependencies to other rules in the same layer or lower layers, and marked dependencies to rules in strictly lower layers. In this way, only such rules may contain grouping constructs where the results required for satisfying the query part can be completely fixed beforehand. Note that this definition of stratification differs slightly from the traditional definition (as given e.g. in [8]) in that it defines dependencies between rules rather than between terms. The rationale for this is twofold: first, the space that is partitioned is the set of rules, and not the set of terms; second, grouping constructs affect the rule as a whole and not individual terms.

In the following, $P = P_1 \uplus \ldots \uplus P_n$ denotes the *partitioning* of a set P into disjoint subsets $P_1, \ldots, P_n$. Furthermore, the following definition uses the notion of *simulation unification* defined in Chapter 8 to define dependencies between rules, because Xcerpt does not differentiate between term labels and predicates (cf. Section 7.2). Also, simulation unification allows to take into account variables or regular expressions occurring in term labels. Note that using unification still does not correspond to a *local stratification* as proposed by [87].

[2] Rather than calling a program *stratified* as in the original definition, we call it *stratifiable* as it is not necessary to compute the stratification during (backward chaining) evaluation.

Definition 6.6 (Grouping Stratification)
Given a program P consisting of rules/goals $\{R_1, \ldots, R_m\}$ ($m \geq 1$).

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

2. P is called *grouping stratifiable*, if there exists a partitioning ($n \geq 1$)

   $$P = P_1 \uplus \cdots \uplus P_n$$

   of P such that for every stratum P_i ($1 \leq i \leq n$) and every rule $R \in P_i$ holds:

   - if the rule head of R contains no grouping constructs and R depends on a rule R' then $R' \in \bigcup_{j \leq i} P_j$, i.e. R' is either in the same stratum as R or in a lower stratum than R
   - if the rule head of R contains grouping constructs and R depends on a rule R' then $R' \in \bigcup_{j < i} P_j$, i.e. R' is in a strictly lower stratum than R

The partition $P = P_1 \uplus \cdots \uplus P_n$ is called a *grouping stratification* of P, and the P_i are called *grouping strata* of P.

Example 6.5
Consider the two programs above.

1. First we have

   $$f\{all\ var\ X\} \leftarrow f\{\{var\ X\}\}$$
   $$f\{a\}$$

   This program is not grouping stratifiable, because the rule depends on itself and obviously cannot be in a lower stratum than itself.

2. The second example is

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

   This program is grouping stratifiable into two strata P_2 and P_1 as follows:

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

Note that the definition of dependency (item 1 in the definition above) only considers isolated unifications of a query term with a construct term and does not respect that a recursive chain of rules might be inconsistent and thus unproblematic. Consider for example the program

$$f\{all\ var\ X\} \leftarrow and\{g\{var\ X\}, k\{var\ X\}\}$$
$$g\{g\{var\ Y\}\} \leftarrow f\{\{var\ Y\}\}$$
$$g\{a\}$$
$$k\{a\}$$

Although the program is not grouping stratifiable (because of the grouping construct in the first rule, the second rule would be in a lower stratum than the first, but as it depends on the first rule, it also needs to be at least in the same or in a higher stratum), it has a valid answer $f\{a\}$, because the second rule only

"produces" terms of the form $g\{g\{\ldots\}\}$, whereas the query part of the first rule only successfully queries $g\{a\}$, because the query for $k\{var\ X\}$ otherwise fails.

A refined approach (not investigated in this thesis) could extend the definition above such that dependencies are not treated as isolated relations between two rules, but instead as chains of rules, where the substitution sets returned by the simulation unifications in a chain are verified for consistency (which in practice results in a partial evaluation of the program). Recursive chains with inconsistent substitution sets could then be admitted. Note that this refinement is similar to *local stratification* as proposed by [87].

Note furthermore that treatment of external resources, although in principle precluded in this and subsequent chapters, is easy: rules or data terms identified by external resources are always considered to be in a stratum lower than any other rule or data term of the program.

6.4.2 Negation Stratification

Negation as Failure (NaF, cf. [40]), like Xcerpt's `not`, is common in rule-based programming languages (e.g. Prolog and Datalog). In NaF, a negated query succeeds if the query itself fails finitely (i.e. can be proven to be not provable). NaF is desirable for a Web query language, because it is close to the intuitive understanding of negation: for instance, it is natural to assume that a train not listed in a train timetable does not exist, instead of requiring that every non-existent train is explicitly listed in the timetable.

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

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

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

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

In this thesis, Xcerpt programs are therefore assumed to be also *negation stratifiable*, a syntactic restriction that excludes such programs that involve problematic use of negation as in the example above. Negation stratification in Xcerpt programs is defined in the usual manner (as e.g. in [8]). In stratifiable programs, both recursion and negation are allowed, but a recursion "through negation" is disallowed.

Note that (negation) stratification is one of many approaches suggested for negation in logic programming and deductive databases. In contrast to most other approaches (like *well-founded semantics* or *stable model semantics*), stratification has the advantage that it is a property that is decidable, can be determined statically, and considers only the examined program, whereas most other approaches also require the data on which the program operates. This latter requirement is impractical for Web query languages, as it would require to consider the complete Web when compiling Xcerpt programs.[3]

As stratification is a rather rigid requirement and also excludes programs that are unproblematic and desirable, it might also be interesting to investigate different, more sophisticated approaches to negation, e.g. *local stratification* [87] or *paraconsistent interpretations* [26, 28] in future work. However, all of these approaches first need to be evaluated for their practical applicability.

Similar to grouping stratification, negation stratification divides a program (represented as a set of rules) into *strata*. The main idea is to disallow recursion over negated queries, but allow any other kind of recursion. A rule R is said to depend negatively on another rule R', if the rule body of R contains a negated query term (i.e. a query term contained in one or more `not` constructs) that simulation unifies with the head of R'. Likewise, a rule R is said to depend positively on another rule R', if the rule body of R contains a non-negated query term that simulation unifies with the head of R'. Note that double negation of a query

[3] In a sense, one could say that after 20 years of research on non-monotonic negation, we have to return to the beginnings due to the particularities of the Web.

term also yields a negative dependency, because `not` implements negation as failure and the query term thus does not yield variable bindings. If a rule R positively depends on a rule R', then R' may be in the same layer as R or in any lower layer. If a rule R depends negatively on a rule R', then R' must be in a strictly lower layer than R. Again, this definition differs from the traditional definition of stratification in that it considers dependencies between rules rather than between terms. The justification for this decision is similar to the reasons given in the previous section.

Again, the following definition uses the *simulation unification* described in Chapter 8. Recall also, that $P = P_1 \uplus \ldots \uplus P_n$ denotes the *partitioning* of a set P into disjoint sets P_i.

Definition 6.7 (Negation Stratification)
Given a program P consisting of rules/goals $\{R_1, \ldots, R_m\}$ ($m \geq 1$).

1. Let $R = t^c \leftarrow Q$ and $R' = t^{c'} \leftarrow Q'$ be rules.

   - R *depends positively* on R', if there exists a query term t^q where all variable occurrences have negative polarity in Q such that t^q simulation unifies in $t^{c'}$, i.e. simulation unification of t^q in t^c yields a non-empty substitution set Σ
   - R *depends negatively* on R', if there exists a query term t^q in Q such that at least one variable occurs positively in t^q, and t^q simulation unifies in $t^{c'}$, i.e. simulation unification of t^q in t^c yields a non-empty substitution set Σ

2. P is called *negation stratifiable*, if there exists a partitioning ($n \geq 1$)

   $$P = P_1 \uplus \cdots \uplus P_n$$

   of P such that for every stratum P_i ($1 \leq i \leq n$) and every rule $R \in P_i$ holds:

   - if R depends positively on a rule R' then $R' \in \bigcup_{j \leq i} P_j$, i.e. R' is either in the same stratum as R or in a lower stratum than R
   - if R depends negatively on a rule R' then $R' \in \bigcup_{j < i} P_j$, i.e. R' is in a strictly lower stratum than R

The partition $P = P_1 \uplus \cdots \uplus P_n$ is called a *negation stratification* of P, and the P_i are called *negation strata* of P.

Example 6.6 (Negation Stratification)

1. The following program (consisting of a single rule) cannot be stratified, as the rule negatively depends on itself:

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

2. The following program can be stratified into three strata from bottom to top:

P_3	$p\{\} \leftarrow and\{not\ q\{\}, s\{\}\}$
P_2	$q\{\} \leftarrow r\{\}$
	$r\{\} \leftarrow s\{\}$
P_1	$s\{\}$

6.4.3 Full Stratification: Combining Grouping Stratification and Negation Stratification

Grouping stratification and negation stratification can be combined in a straightforward manner. The grouping strata further divide the negation strata (or vice versa) such that the respective properties also hold. As before, this definition uses the simulation unification described in Chapter 8, and $P = P_1 \uplus \ldots \uplus P_n$ denotes a *partitioning* of a set P into disjoint sets P_i.

Definition 6.8 (Full Stratification)
Given a program P consisting of rules/goals $\{R_1, \ldots, R_m\}$ ($m \geq 1$).

1. Let $R = t^c \leftarrow Q$ and $R' = t^{c'} \leftarrow Q'$ be rules.

   - R *depends* on R' if there exists a (negated or non-negated) query term t^q in Q such that t^q simulation unifies in $t^{c'}$
   - R *depends positively* on R' if there exists a non-negated query term t^q in Q such that t^q simulation unifies in $t^{c'}$
   - R *depends negatively* on R' if there exists a negated query term *not* t^q in Q such that t^q simulation unifies in $t^{c'}$

2. P is called *fully stratifiable* (or simply *stratifiable*), if there exists a partitioning ($n \geq 1$)

   $$P = P_1 \uplus \cdots \uplus P_n$$

   of P such that for every stratum P_i ($1 \leq i \leq n$) and for every rule $R \in P_i$ holds:

   - if R depends negatively on a rule R', or the head of R contains grouping constructs and R depends positively or negatively on R', then $R' \in \bigcup_{j<i} P_j$, i.e. R' is in a strictly lower stratum than R
   - if the head of R contains no grouping constructs and R depends positively on a rule R' then $R' \in \bigcup_{j \leq i} P_j$, i.e. R' is in the same or in a lower stratum than R

The partition $P = P_1 \uplus \cdots \uplus P_n$ is called a *full stratification* of P, and the P_i are called *strata* of P.

CHAPTER

SEVEN

Declarative Semantics: A Model Theory for Xcerpt

This chapter introduces a model theory for grouping stratifiable Xcerpt programs without negation. Intuitively, the definition of interpretations and models is straightforward: an interpretation is a set of data terms and specifies what data terms exist; a model is then simply an interpretation that consists of the terms that are "produced" by the rules in a program.

The model theory for Xcerpt programs follows the classical Tarski-style semantics for first order logic rather closely, but needs to take into account the particularities of Xcerpt terms and programs. As a first step, Section 7.2 introduces *term formulas*, which depart from the formulas in first order logic in that they do not differentiate between relation symbols and term symbols, because Xcerpt only considers "data" and not "statements". Next, a notion of *substitution sets* is described in Section 7.3. Substitution sets take the role of substitutions in first order logic and logic programming and are required to properly convey the meaning of the grouping constructs `all` and `some`. Interpretations and satisfaction of term formulas are then defined in Section 7.4. This definition makes use of the *ground query term simulation* relation described in Section 4.4 to take into account query terms with incomplete term specifications (e.g. unordered or partial terms). In Section 7.5, a fixpoint semantics for stratifiable Xcerpt programs is suggested, first for programs without negation, and then for arbitrary Xcerpt programs. Finally, Section 7.6 contains some concluding remarks on the model theory and fixpoint semantics introduced here.

7.1 Preliminaries

Like in Chapter 6, the following notations are used throughout subsequent sections to simplify the discussion of the semantics of Xcerpt programs:

- *Programs* are sets of rules (and goals), usually denoted by $P = \{R_1, \ldots, R_n\}$.
- *Rules* are denoted by $R = t^c \leftarrow \mathcal{Q}$, where t^c is the construct term of the rule and $\mathcal{Q}$ the query part; the set of rules of a program P is usually denoted by $\mathcal{R} \subseteq P$
- *Goals* are denoted by $G = t^c \leftarrow_g \mathcal{Q}$, where t^c is the construct term of the rule and $\mathcal{Q}$ the query part; the set of goals of a program P is usually denoted by $\mathcal{G} \subseteq P$
- *Queries* of the form $\texttt{and}\{Q_1, \ldots, Q_n\}$ are sometimes denoted by $Q_1 \wedge \cdots \wedge Q_n$ or by $\bigwedge_{1 \leq i \leq n} Q_i$; likewise, queries of the form $\texttt{or}\{Q_1, \ldots, Q_n\}$ are sometimes denoted by $Q_1 \vee \cdots \vee Q_n$ or by $\bigvee_{1 \leq i \leq n} Q_i$ and queries of the form `not Q` are sometimes denoted by $\neg Q$.
- *Resources* are considered to be internalised, i.e. it is assumed that any data term referred to by an input resource specification of the form $\texttt{in}\{\ldots\}$ is part of the program; this assumption simplifies the formal treatment below and can be implemented in a straightforward (but inefficient) manner.
- The set $\mathcal{T}$ denotes the set of all terms, $\mathcal{T}^q \subsetneq \mathcal{T}$ the set of all query terms, $\mathcal{T}^g \subsetneq \mathcal{T}^q$ the set of all ground query terms, and $\mathcal{T}^d \subsetneq \mathcal{T}^g$ the set of all data terms.

In most parts of the formalisation, rules and goals are not distinguished unless explicitly mentioned; this simplification is useful as rules and goals are very similar. Also, parts of the formalisation use a simplified term representation, where strings and regular expressions are simply treated as compound terms with empty content and total term specification. E.g. the string `"XML"` is represented as `"XML"{}`.

The model theory described below requires programs to be grouping stratifiable (cf. Definition 6.8), as the meaning of such programs are in every case reasonable. Furthermore, negation (as failure) is not further investigated in this thesis, but standard logic programming approaches might be applicable (as e.g. described in [9, 98]).

7.2 Terms as Formulas

Classical logic distinguishes between

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

Statements represented by formulas have truth values, objects represented by terms have no truth value. In contrast, XML and Web data does not need this distinction, because it has no (formal) semantics and merely holds semistructured data. Therefore, Xcerpt terms (corresponding to Web data) are considered as being atomic formulas representing the statement that the respective terms "exist". A salient aspect of this representation is the possibility to specify integrity constraints for data terms. These are briefly introduced in the *Perspectives*, Section 9.6.

7.2.1 Term Formulas

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

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

Example 7.1
The following example shows a term formula built up from query terms, implications and quantifiers. It represents an integrity constraint that requires all books in the `bib.xml` document to have at least one author:

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

7.2.2 Xcerpt Programs as Formulas

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

In addition to the usual quantifiers $\forall$ and $\exists$, the grouping constructs `all` and `some` that may be part of a construct term may bind variables in a formula within a specific scope, usually the head and body of a rule. As these constructs are contained within the term structure, their scope is not immediately

apparent. It is thus useful to introduce new symbols $\ll \cdot \gg$ that are used to indicate the scope of *all* the grouping constructs contained in them. In practice, it is neither desirable nor useful to have scopes extending over different subformulas for the grouping constructs contained in a single construct term, thus a single scope for all grouping constructs suffices. The grouping constructs of a construct term always refer to the variables of a single rule and thus all have the same scope.

Example 7.2
Consider for example the program (in formula notation)

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

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

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

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

Example 7.3
Consider the following Xcerpt program (in the notation introduced in Section 6.1 and with internalised resources):

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

The formula representation of this program is as follows:

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

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

7.3 Substitutions and Substitution Sets

7.3.1 Preliminary Notions

A first intuitive notion of substitutions has already been given in Section 4.4. This notion was rather straightforward and similar to the usual definition of substitutions, since Section 4.4 only considered query and data terms. However, variable restrictions occurring in query terms have to be taken into account. As a variable might be restricted, not every substitution is applicable to every query term.

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

In order to define such groupings, it is therefore necessary to provide a construct that represents all possible alternatives and can be applied to a construct term. This is called a *substitution set* below. Substitution sets are used in Section 7.4 which defines satisfaction for Xcerpt term formulas and later in Section 8.2.1 to define the notion of a *simulation unifier*. In the following, substitutions are denoted by lowercase greek letters (like σ or π), while substitution sets are denoted by uppercase greek letters (like Σ or Π).

Substitutions

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

If a substitution is *applied* to a query term t^q, all occurrences of variables for which the substitution provides assignments are replaced by the respective assignments (see Section 7.3.2 below). The resulting term is called an *instance* of t^q and the substitution. Not every substitution can be applied to every query term: variable assignments in the substitution have to respect variable restrictions occurring in the pattern for a substitution to be applicable (see also 7.3.2). If a substitution σ respects the variable restrictions in a query term t^q, it is said to be *a substitution for* t^q. For example, the substitution $\{X \mapsto f\{a\}\}$ is a substitution for *var* $X \rightarrow f\{\{\}\}$, but not for *var* $X \rightarrow g\{\{\}\}$. Note that a substitution cannot be applied to a construct term, because construct terms may contain grouping constructs that group several instances of subterms together. Instead, substitution sets are used for this purpose (see below).

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

A substitution σ_1 is a *subset* of a substitution σ_2 (i.e. $\sigma_1 \subseteq \sigma_2$), if $\sigma_1(X) \cong \sigma_2(X)$ for every variable name X with $\sigma_1(X) \neq X$ (i.e. σ_1 does not map X to itself), where $\cong$ denotes simulation equivalence (cf. Section 4.4.4). Correspondingly, two substitutions σ_1 and σ_2 are considered to be *equal* (i.e. $\sigma_1 = \sigma_2$), if $\sigma_1 \subseteq \sigma_2$ and $\sigma_2 \subseteq \sigma_1$. For example, $\{X \mapsto f\{a,b\}\}$ and $\{X \mapsto f\{b,a\}\}$ are equal. This definition is reasonable because the data terms resulting from applying two such substitutions are treated equally in the model theory described below.

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

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

Substitution Sets

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

Substitution sets can be *applied* to a query *or* construct term (cf. Sections 7.3.2 and 7.3.3). The result of this application is in general a set of terms called the *instances* of the substitution set and the term. A substitution set Σ is only applicable to a query term t^q, if all substitutions in Σ are applicable to t^q. In this case, Σ is called *a substitution set for* t^q. Since construct terms do not contain variable restrictions, every substitution set except for the empty set is a substitution set for a construct term. There exists no query or construct term t such that the empty substitution set $\{\}$ is a substitution set for t.

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

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

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

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

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

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

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

Maximal Substitution Sets

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

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

Definition 7.1 (Maximal Substitution Set)
Let Q be a quantifier free query term formula with set of variables V, let P be a property, and let Σ be a set of substitutions such that P holds for Σ. Σ is called a *maximal substitution set wrt. P and Q*, if there exists no substitution set Φ such that P holds for Φ and $\Sigma_{|V}$ is a proper subset of $\Phi_{|V}$ (i.e. $\Sigma_{|V} \subsetneq \Phi_{|V}$).

7.3.2 Application to Query Terms

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

Definition 7.2 (Substitutions: Application to Query Terms)
Let t^q be a query term.

1. The application of a *substitution* σ to t^q, written $\sigma(t^q)$ is recursively defined as follows:

   - $\sigma(var\ X) = t'$ if $(X \mapsto t') \in \sigma$
   - $\sigma(var\ X \rightarrow s) = t'$ if $(X \mapsto t') \in \sigma$ and $\sigma(s) \preceq t'$
   - $\sigma(f\{t_1, \ldots, t_n\}) = \sigma(f)\{\sigma(t_1), \ldots, \sigma(t_n)\}$
   - $\sigma(f[t_1, \ldots, t_n]) = \sigma(f)[\sigma(t_1), \ldots, \sigma(t_n)]$
   - $\sigma(f\{\{t_1, \ldots, t_n\}\}) = \sigma(f)\{\{\sigma(t_1), \ldots, \sigma(t_n)\}\}$
   - $\sigma(f[[t_1, \ldots, t_n]]) = \sigma(f)[[\sigma(t_1), \ldots, \sigma(t_n)]]$
   - $\sigma(without\ t) = without\ \sigma(t)$
   - $\sigma(optional\ t) = optional\ \sigma(t)$

   for some $n \geq 0$.

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

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

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

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

7.3.3 Application to Construct Terms

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

Definition 7.3 (Grouping of a Substitution Set)
Given a substitution set Σ and a set of variables $V = \{X_1, \ldots, X_n\}$ such that all $\sigma \in \Sigma$ have bindings for all $X_i, 1 \leq i \leq n$.

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

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

Example 7.4
Given the substitution set $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ with

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

The grouping of Σ on $V = \{X_1\}$ is

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

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

Example 7.5
Given a term $t = f\{X_1, g\{all\ X_2\}\}$, i.e. $FV(t) = \{X_1\}$. Consider again

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

from Example 7.4. The result of applying Σ to t is

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

The following definition specifies how a substitution set is applied to a construct term t^c. The definition is divided into two parts: In the first part, it is assumed that all substitutions in the substitution set Σ contain the same assignments for the free variables of t^c (variables occurring within the scope of grouping constructs are unrestricted). As the quotient $\Sigma/_{\simeq_{FV(t^c)}}$ in this case obviously only contains a single equivalence class, the application of this restricted Σ to t^c yields only a single term, which simplifies the recursive definition. In the second part of Definition 7.4, this restriction is lifted.

Since the construction of data terms requires to construct new lists of subterms, the following definition(s) use the notion of *term sequences* introduced in Section 4.4.2. Recall that a sequence is a binary relation between a set of integers and a set of terms, and usually denoted by $S = \langle x_1, \ldots, x_n \rangle$ for some n and terms x_i. Recall furthermore the definitions of *subsequences* and *concatenation* (Definition 4.5 on page 81).

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

Definition 7.4 (Substitutions: Application to Construct Terms)

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

   - $[\![\sigma]\!](var\ V) = \langle \sigma(V) \rangle$[2]
   - $[\![\sigma]\!](f\{t_1,\ldots,t_n\}) = \langle [\![\sigma]\!](f)\{[\![\sigma]\!](t_1) \circ \cdots \circ [\![\sigma]\!](t_n)\} \rangle$ for some $n \geq 0$
   - $[\![\sigma]\!](f[t_1,\ldots,t_n]) = \langle [\![\sigma]\!](f)[[\![\sigma]\!](t_1) \circ \cdots \circ [\![\sigma]\!](t_n)] \rangle$ for some $n \geq 0$
   - $[\![\sigma]\!](all\ t) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!],\ldots,[\![\tau_k]\!]\} = [\![\sigma]\!]/_{\simeq_{FV(t)}}$
   - $[\![\sigma]\!](all\ t\ group\ by\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!],\ldots,[\![\tau_k]\!]\} = [\![\sigma]\!]/_{\simeq_{FV(t)\cup V}}$
   - $[\![\sigma]\!](all\ t\ order\ by\ f\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\langle [\![\tau_1]\!],\ldots,[\![\tau_k]\!] \rangle = sort(f(V), [\![\sigma]\!]/_{\simeq_{FV(t)\cup V}})$
   - $[\![\sigma]\!](some\ k\ t) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!],\ldots,[\![\tau_k]\!]\} \subseteq [\![\sigma]\!]/_{\simeq_{FV(t)}}$
   - $[\![\sigma]\!](some\ k\ t\ group\ by\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!],\ldots,[\![\tau_k]\!]\} \subseteq [\![\sigma]\!]/_{\simeq_{FV(t)\cup V}}$
   - $[\![\sigma]\!](some\ k\ t\ order\ by\ f\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\langle [\![\tau_1]\!],\ldots,[\![\tau_k]\!] \rangle \sqsubseteq sort(f(V), [\![\sigma]\!]/_{\simeq_{FV(t)\cup V}})$
   - $[\![\sigma]\!](optional\ t) = \begin{cases} [\![\sigma]\!](t) & \text{if the ground instance } [\![\sigma]\!](t) \text{ exists} \\ \langle\rangle & \text{otherwise} \end{cases}$
   - $[\![\sigma]\!](optional\ t\ with\ default\ t') = \begin{cases} [\![\sigma]\!](t) & \text{if the ground instance } [\![\sigma]\!](t) \text{ exists} \\ [\![\sigma]\!](t') & \text{otherwise} \end{cases}$

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

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

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

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

[1] As the substitution set is grouped on V, all substitutions in $[\![\sigma]\!]$ (respectively $[\![\tau]\!]$) provide identical bindings for variables in V.
[2] Note that σ is the representative of the equivalence class $[\![\sigma]\!]$

Example 7.6
Consider the substitution set

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

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

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

The ground instances of t_1 and t_2 by Σ are thus

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

7.3.4 Application to Query Term Formulas

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

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

Definition 7.5 (Substitutions: Application to Query Term Formulas)
Let F be a quantifier-free term formula where all atoms are query terms (a *query term formula*).

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

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

7.4 Interpretations and Entailment

The definition of satisfaction of Xcerpt term formulas, and in particular of Xcerpt programs, is similar to the approach taken in classical first order logic, but differs in several important aspects: term formulas do not differentiate between relations and terms, and the incompleteness of query terms and the grouping constructs in construct terms have to be taken into account. Section 7.4.1 gives an intuitive meaning of interpretations for Xcerpt term formulas. Satisfaction is then defined in Section 7.4.2 in terms of the *simulation relation* introduced earlier in Section 4.4. Based on this definition of satisfaction, entailment between formulas can be defined in the classical manner.

7.4.1 Interpretations

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

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

Definition 7.6 (Interpretation)
An *interpretation* M is a tuple $M = (I, \Sigma)$ where I is a set of data terms and $\Sigma \neq \emptyset$ is a grounding substitution set.

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

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

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

7.4.2 Satisfaction and Models

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

Unlike in traditional logic programs, rules in Xcerpt are not treated as (classical) implications ($\Rightarrow$ below), because the grouping constructs `all` and `some` require that the query part of a rule is not only satisfied, but that it is also satisfied in the maximal manner, i.e. the substitution set yielding the ground instance of the construct term must include all possible substitutions for which the query part is satisfied. Otherwise, interpretations would include answer terms for a rule that differ from the intuitive understanding of the constructs `all` and `some` (see Example 7.8 below). The definition of satisfaction for Xcerpt rules uses the notion of maximal substitution sets defined above in Definition 7.1.

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

Definition 7.7 (Satisfaction, Model)

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

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

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

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

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

The standardisation apart of Xcerpt rules (cf. Section 6.3) allows to replace Σ by Σ' in the model definition for $\forall^* \ll t \leftarrow Q \gg$. Otherwise, the substitutions in Σ and Σ' would have to be merged to $\Sigma \circ \Sigma'$.

Example 7.7 (Satisfaction of Term Formulas)
Let $M = (I,\Sigma)$ be an interpretation with

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

The following statements hold for M:

1. $M \models f[a,b]$, because for each $t \in \Sigma(f[a,b]) = \{f[a,b]\}$ exists a $t' \in I$ with $t \preceq t'$
2. $M \not\models f[a,d]$, because for $t = f[a,d] \in \Sigma(f[a,d]) = \{f[a,d]\}$ does not exist a $t' \in I$ with $t \preceq t'$.
3. $M \models f\{a,b\}$, because for each $t \in \Sigma(f\{a,b\}) = \{f\{a,b\}\}$ exists a $t' \in I$ with $t \preceq t'$
4. $M \models f\{\{var\ X, var\ Y\}\}$, because
   - $\sigma_1 = \{X \mapsto a,\ Y \mapsto b\}$ and $\sigma_1(f\{\{var\ X, var\ Y\}\}) \preceq f[a,b]$, and
   - $\sigma_2 = \{X \mapsto a,\ Y \mapsto c\}$ and $\sigma_2(f\{\{var\ X, var\ Y\}\}) \preceq f[a,c]$
5. $M \models \exists Z.f\{\{var\ Z\}\}$, because $M' = (I,\Sigma')$ with
   $\Sigma' = \{\{X \mapsto a,\ Y \mapsto b,\ Z \mapsto a\}, \{X \mapsto a,\ Y \mapsto c,\ Z \mapsto a\}\}$
   is a model for $f\{\{var\ Z\}\}$
6. $M \not\models \forall Z.f\{\{var\ Z\}\}$, because there exists a term $f[a,b]$ as substitution for Z such that $M \not\models f\{\{f[a,b]\}\}$
7. $M \models \forall Z.var\ Z$, because for all $t \in I$ holds that $M' = (I,\Sigma')$ with $\Sigma' = \{\{X \mapsto a,\ Y \mapsto b,\ Z \mapsto t\}, \{X \mapsto a,\ Y \mapsto c,\ Z \mapsto t\}\}$
   is a model for $var\ Z$[3]

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

Example 7.8 (Satisfaction of Xcerpt Programs)
Let P be the following Xcerpt program (in compact notation):

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

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

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

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

$$f\{all\ var\ X\} \leftarrow g\{var\ X\}$$
$$g\{g\{var\ Y\}\} \leftarrow g\{var\ Y\}$$
$$g\{a\}$$

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

7.5 Fixpoint Semantics

A classical approach to describing the semantics of logic programs is the so-called *fixpoint semantics*, first proposed by Van Emden and Kowalski [103]. In the fixpoint semantics, a model is constructed by iteratively trying to apply program rules (using an operator called T_P) to a set of data terms and adding their results until a fixpoint is reached, i.e. no new data terms can be added. This smallest fixpoint is then a model of the program (assuming that programs do not contain negation).

Example 7.9
Consider again the program

$$f\{all\ var\ X\} \leftarrow g\{\{var\ X\}\}$$
$$g\{a\}$$

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

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

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

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

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

The following section proposes a fixpoint semantics for Xcerpt programs with grouping constructs but without negation, and shows that the fixpoint of the program is also a model of a program. Since the fixpoint semantics is the most precise characterisation of Xcerpt programs available, it is also used as the reference for the verification of the backward chaining algorithm. Programs with negation are not considered in this thesis, but their treatment should be very similar to the treatment of negation in other logic programming

languages. Since Xcerpt programs are negation stratifiable, a similar approach to the approach taken by Apt, Blair, and Walker [8] appears promising.

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

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

Definition 7.8 (Fixpoint Operator T_P, Fixpoint Interpretation)
Let P be an Xcerpt program.

1. The fixpoint operator T_P is defined as follows:

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

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

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

Example 7.10
Consider the following Xcerpt program (cf. Example 6.5):

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

Applying the fixpoint operator T_P yields the following results:

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

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

Therefore, we refine the notion of fixpoint interpretations to fixpoint interpretations for stratifiable programs. Constructing fixpoints for Xcerpt programs containing grouping constructs is based on the grouping stratification of such programs and simply applies the fixpoint operator stratum by stratum, beginning with the lowest stratum and ending with the highest. The following definition follows closely a definition by Apt, Blair, and Walker [8]:

Definition 7.9 (Fixpoint Interpretation for Stratifiable Programs)
Let P be a program with grouping stratification $P = P_1 \uplus \cdots \uplus P_n$ ($n \geq 1$). The fixpoint interpretation M_P is defined by

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

with $M_P = M_n$.

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

Example 7.11
Consider the following Xcerpt program stratifiable into two strata P_1 and P_2 (cf. Example 6.5):

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

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

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

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

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

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

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

Theorem 7.10
Let P be a grouping stratified program without negation. Then the fixpoint M_P of P is a model of P.

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

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

□

7.6 Remarks

The model theory and fixpoint semantics described above provide a rather straightforward declarative semantics for Xcerpt programs. However, this semantics is unsatisfactory in that it only covers a limited set of Xcerpt programs (namely those that are grouping stratifiable), does not cover negation (as failure), and does not provide a theory of minimal model as is usually done in traditional logic programming. Solutions to the restrictions imposed by stratification and to negation might be found in other approaches that have been investigated in logic programming. Minimal models have been investigated extensively in the course of this thesis, but a satisfactory definition has not yet been found. Under satisfactory we understand a characterisation of Xcerpt programs that exactly covers the semantics given by fixpoint interpretations, but which is not just fixpoint interpretations wrapped in different clothes. Furthermore, this characterisation still needs to be easy to understand. Characterisations that do not adhere to these properties would not add anything to the semantics of Xcerpt programs.

CHAPTER

EIGHT

Operational Semantics: Backward Chaining and Simulation Unification

This chapter describes an algorithm for the evaluation of Xcerpt programs using a backward chaining strategy. The algorithm is defined in terms of a simple constraint solver (described in Section 8.1). Constraint solving is a method that allows a rather efficient evaluation by excluding irrelevant parts of the solution space as early as possible, and has been applied to many practical problems (cf. [51]). Constraint solving is advantageous because

- it uses declarative simplification rules that are easy to understand,
- it allows to reduce the search space by detecting inconsistencies early,
- it tries to avoid complex computations (like creating answer terms) as long as possible, and
- it allows to easily add user-defined theories specified in terms of additional simplification rules to the evaluation engine.

This constraint solver differs from traditional constraint solvers in that it needs to treat disjunctions between constraint formulas and negation, but the approach taken here is rather straightforward.

The evaluation algorithm is defined in two parts: first, an algorithm called *simulation unification* is introduced. Simulation unification is a novel kind of (non-standard) unification that allows to treat the particularities of Xcerpt terms properly and is based on the notions of ground query term simulation and answers of Chapter 4. It has first been proposed in [24] and is further refined here. Based on simulation unification, a *backward chaining* algorithm is then described that eventually determines answer terms as defined in Chapter 7. Salient aspects of this backward chaining algorithm are the treatment of the grouping constructs `all` and `some`, and the unusually high level of branching in the proof trees that result from incomplete term specifications. While evaluation rules for programs with negation and optional subterms are given, these are not verified against the declarative semantics, as the fixpoint theory described in Chapter 7 currently does not cover negation.

This chapter is structured as follows: Section 8.1 introduces the constraint solver and data structures used in this chapter, and defines the meaning of a constraint store in form of *solution sets*. Section 8.2 describes the simplification rules that constitute simulation unification algorithm and shows the correctness of this algorithm against an abstract formalisation of most general simulation unifiers. Finally, Section 8.3 describes the rules for a backward chaining evaluation. A soundness and weak completeness result for this algorithm is also given.

8.1 A Simple Constraint Solver

The evaluation of Xcerpt programs is described in terms of a constraint solver that applies so-called *simplification rules* to a constraint store consisting of conjunctions and disjunctions of constraints. The purpose of the constraint solver is to determine variable bindings for variables occurring in query and construct

terms, which ultimately yield substitutions that can be used to create the answer terms of a program. A simplification rule in this thesis has the following form:

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

where $C_1, \ldots, C_n$ ($n \geq 1$) are atomic constraints (the condition) and D is either an atomic constraint, or a conjunction or disjunction of constraints (the consequence). If a simplification rule is applied, then the conjunction $C_1 \wedge \cdots \wedge C_n$ in the constraint store is replaced by the constraint D. Note that these simplification rules are similar to the simplification rules in the language *Constraint Handling Rules* [50], albeit with a different notation.

The constraint solver is non-deterministic to a high degree in that the order in which simplification rules are applied is not significant. This approach might be advantageous, as it gives much freedom to the evaluation engine to e.g. perform optimisations (cf. Section 9.5.4).

This constraint solver differs from common approaches in that the result of a rule may contain disjunctions, whereas usually only conjunctions are admitted. Such constraint solvers have been studied in constraint programming research, e.g. in [126]. The approach taken in this thesis is rather simplistic, as it after each application of a simplification rule creates the disjunctive normal form (DNF) of the constraint store. Simplification rules are independently applied to the different conjuncts of the DNF. This approach is rather inefficient in implementations, and various optimisations can be considered. A straightforward optimisation would be to not create the DNF after *each* simplification step, but instead only if it is "necessary", because no other simplification rules apply. However, such optimisations are not further investigated in this thesis, as the focus is on Web query languages and not on constraint programming.

Furthermore, the constraint solver needs to be able to treat negation. As both negation constructs `not` and `without` describe negation as failure, the negation behaves differently to classic negation in some cases (cf. Example 8.4). The treatment of negation is described in the formula simplification rules in Section 8.1.3, and in the consistency verification rules 3, 4, and 5 in Section 8.1.4 below.

8.1.1 Data Structures and Functions

Constraints

The main data structure of the evaluation algorithm is the *constraint store* which may contain several types of constraints, including other (sub-)constraint stores. For the purpose of this thesis, constraints are defined by the following grammar (defined in a variant of *Extended Backus-Naur Form*):

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

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

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

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

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

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

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

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

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

Functions

substitutions(CS): The ultimate step of the algorithm, after no more rules are applicable or necessary, is always to generate a set of substitutions from the constraint store. In this step, *CS* is put in DNF, all constraints of the form $X \preceq_u t$ (where X is a variable and t is a construct term[1]) are replaced by $X = t$ and for each conjunct of *CS* a separate substitution is generated from these replacements. Note that

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

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

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

apply(Σ,t): Applying a set of substitutions Σ to a term is implemented recursively over the term structure. The implementation of this function can be derived from Definitions 7.2 and 7.4 in a straightforward manner.

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

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

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

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

8.1.2 Solution Set of a Constraint Store

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

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

Definition 8.1 (Solution Set of a Constraint Store)
Let *CS* be a completely solved constraint store, i.e. consisting only of simulation constraints where one side is a variable, conjunctions, disjunctions, and the boolean constraints *True* and *False*. The solution set $\Omega(CS)$ is a grounding substitution set recursively defined as follows:

- $\Omega(True)$ is the set of all all-grounding substitutions (cf. Section 7.3)
- $\Omega(False) = \{\}$, i.e. the empty set
- $\Omega(X \preceq_u t)$ is the set of all all-grounding substitutions σ such that $\sigma(X) \cong \sigma(t)$
- $\Omega(t \preceq_u X)$ is the set of all all-grounding substitutions σ such that $\sigma(t) \preceq \sigma(X)$
- $\Omega(C_1 \wedge C_2) = \Omega(C_1) \cap \Omega(C_2)$
- $\Omega(C_1 \vee C_2) = \Omega(C_1) \cup \Omega(C_2)$
- $\Omega(\neg C) = \Omega(True) \setminus \Omega(C)$

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

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

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

Corollary 8.2
Let $CS = C_1 \vee \cdots \vee C_n$ be a constraint store in disjunctive normal form, and V the set of variables occurring in CS. If in every conjunct C_i, each variable $X \in V$ occurs in a simulation constraint of the form $X \preceq_u t$ where t is a data term, then $substitutions(CS) = \Omega(CS)_{|V}$.

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

8.1.3 Constraint Simplification

The usual simplification rules for formulas apply, for example:

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

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

8.1.4 Consistency Verification Rules

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

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

Rule 1: Consistency

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

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

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

Example 8.1 (Consistency Rule)

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

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

Rule 2: Transitivity

The *transitivity* rule replaces variable occurrences of a variable X in the upper bounds of a variable by the upper bound of X. This rule is justified by the simulation pre-order defined in Corollary 4.10 and is needed to ultimately create ground terms as bindings for all variables. In the following, the notation $t[t'/X]$ denotes "replace all occurrences of X in t by t'".

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

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

Example 8.2 (Transitivity Rule)

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

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

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

8.1.5 Constraint Negation

Negated constraints represent exclusion of certain variable bindings, and may result from the evaluation of the constructs `without` (subterm negation), `optional` (optional subterms), and `not` (query negation). For example, the constraint $\neg(X \preceq f\{a,b\})$ disallows bindings for X that are simulation equivalent with $f\{a,b\}$. Note that, although these constructs implement negation as failure, constraint negation is the ordinary negation of classical logic. The usual transformation rules apply, namely $\neg(C \wedge D) = \neg C \vee \neg D$, $\neg(C \vee D) = \neg C \wedge \neg D$, $\neg True = False$, and $\neg False = True$. Note, however, that $\neg\neg C \neq C$, because C is not allowed to define variable bindings (cf. *range restrictedness*, Section 6.2).

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

Rule 3: Consistency with Negation

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

$$\frac{X \preceq_u t_1 \quad \neg(X \preceq_u t_2)}{X \preceq_u t_1 \wedge \neg(t_1 \preceq_u t_2 \wedge t_2 \preceq_u t_1)}$$

Example 8.3 (Consistency Rule with Negation)
Consider the constraint store

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

Applying the consistency rule with negation yields

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

the DNF of which is

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

and after further decomposition steps

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

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

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

Rule 4: Transitivity with Negation

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

$$\frac{\neg(t_1 \preceq_u t_1') \text{ such that } t_1' \text{ contains the variable } X \quad X \preceq_u t_2}{\neg(t_1 \preceq_u t_1') \wedge X \preceq_u t_2 \wedge \neg(t_1 \preceq_u t_1'[t_2/X])}$$

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

$$\frac{t_1 \preceq_u t_1' \text{ such that } t_1' \text{ contains the variable } X \quad \neg(X \preceq_u t_2)}{t_1 \preceq_u t_1' \wedge \neg(X \preceq_u t_2) \wedge \neg(t_1 \preceq_u t_1'[t_2/X])}$$

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

Rule 5: Negation as Failure

The last rule is necessary for cases where a variable only appears in a negated simulation constraint, but nowhere else in a non-negated simulation constraint of the constraint store. Due to the range restrictedness of Xcerpt rules, such constraints can never be produced directly in the treatment of `not` or `without` (range restrictedness enforces that each variable occurring in a negated part also appears elsewhere in a non-negated part). They may, however, be the consequence of applications of rules 3 and 4, and might be produced when decomposing a query term containing the construct `optional` (see Section 8.2.2 below).

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

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

Constraints of the form $\neg\mathit{True}$ and $\neg\mathit{False}$ are treated by the formula simplification described above. Example 8.9 shows a case where this consistency rule is needed. An interesting application of this rule involves double negation:

Example 8.4 (Negation as Failure Rule)
Consider the simulation constraint $\neg\neg(X \preceq_u t)$ such that X does not occur elsewhere in a non-negated simulation constraint. Applying Rule 5 to this constraint yields $\neg\mathit{False} = \mathit{True}$ (and not $X \preceq_u t$ as one might expect). The rationale for this is that the negation used is negation as failure and not classical negation, and variables within a simulation constraint that are negated twice do not define variable bindings (see also the definition of *range restrictedness* in Chapter 6).

8.1.6 Program Evaluation

Program evaluation starts at the program goals, and tries to determine answer terms by evaluating the query parts for each goal in a backward chaining fashion. Given a program P, the general scheme of program evaluation is as follows (the backward chaining algorithm itself is described in Section 8.3 below):

Algorithm 8.1

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

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

Algorithm 8.2

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

Note that "rule" in the algorithm above denotes a simplification rule of the constraint solver and not an Xcerpt rule. Rules from all three parts may be interleaved and the decision on the selection of rule applications is deliberately left open (i.e. the algorithm described here is non-deterministic), as long as the selection is "fair" (i.e. each possible rule is applied within finitely many steps). This non-determinism allows for interesting considerations about selection strategies that have not been investigated much in logic programming (cf. Section 9.5.4).

8.2 Simulation Unification

Simulation Unification, as previously described in [24], is an algorithm that, given two terms t_1 and t_2, determines variable substitutions such that the ground instances of t_1 and t_2 simulate. Like standard unification (cf. [91]), simulation unification is *symmetric* in the sense that it can determine (partial) bindings for variables in both terms. Unlike standard unification, it is however *asymmetric* in the sense that it does not make the two terms equal, but instead ensures a ground query term simulation, which is directed and asymmetric. The outcome of Simulation Unification is a set of substitutions called *simulation unifier*.

Simulation Unification consists mainly of decomposition rules that operate recursively and in parallel on the two unified terms (Section 8.2.2). When all terms are completely decomposed, the result is a constraint store containing conjunctions and disjunctions of simulation constraints where the left or the right side is a variable. These yield variable bindings by replacing simulation constraints of the form $X \preceq_u t$ by $X = t$. The consistency verification rules described above ensure that all simulation constraints are consistent and can be interleaved at any point.

8.2.1 Simulation Unifiers

In Classical Logic, a unifier is a substitution for two terms t_1 and t_2 that, applied to t_1 and t_2, makes the two terms identical. The *simulation unifiers* introduced here follow this basic scheme, with two extensions: instead of equality, simulation unifiers are based on the (asymmetric) simulation relation of Section 4.4 and instead of a single substitution, substitution sets as introduced in Section 7.3 are considered. Both extensions are necessary, as they recognise the special Xcerpt constructs *all* and *some* and incomplete term specifications.

Informally, a *simulation unifier* for a query term t^q and a construct term t^c is a set of substitutions Σ, such that each ground instance $t^{q'}$ of t^q in Σ simulates into a ground instance $t^{c'}$ of t^c in Σ. This restriction is too weak for fully describing the semantics of the evaluation algorithm. For example, consider a substitution set $\Sigma = \{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto b, Y \mapsto a\}$, a query term $t^q = f\{var\ X\}$ and a construct term $t^c = f\{var\ Y\}$. With the informal description above, Σ would be a simulation unifier of t^q in t^c, but this is not reasonable. We therefore also require that the substitution $\sigma \in \Sigma$ that yields $t^{q'}$ also is "used" by $t^{c'}$. This can be expressed by grouping the substitutions according to the free variables in t^c (cf. Definition 7.3 on page 148).

Definition 8.3 (Simulation Unifier)
Let t^q be a query term, let t^c be a construct term with the set of free variables $FV(t^c)$, and let Σ be an all-grounding substitution set. Σ is called a *simulation unifier* of t^q in t^c, if for each $[\![\sigma]\!] \in \Sigma/_{\simeq_{FV(t^c)}}$ holds that

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

Recall from Section 7.3 that all substitutions in an all-grounding substitution set assign data terms to each variable. Intuitively, it is sufficient to only consider grounding substitutions for t^q and t^c. However, all-grounding substitution sets simplify the formalisation of most general simulation unifiers below.

Example 8.5 (Simulation Unifiers)

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

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

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

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

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

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

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

Definition 8.4 (Most General Simulation Unifier)
Let t^q be a query term and let t^c be a construct term without grouping constructs such that there exists at least one simulation unifier of t^q in t^c. The *most general simulation unifier* (mgsu) of t^q in t^c is defined as the union of all simulation unifiers of t^q in t^c.

In Section 8.2.4, we shall see that the simulation unification algorithm described here computes the most general simulation unifier. Note that the most general simulation unifier is indeed always a simulation unifier if t^c does not contain grouping constructs. This is easy to see because the union of two simulation unifiers simply adds ground instances of t^q and t^c where for every ground instance $t^{q'}$ of t^q there exists a ground instance $t^{c'}$ of t^c such that $t^{q'} \preceq t^{c'}$. This does in general not hold for construct terms with grouping, but as grouping is not treated inside the unification algorithm, the definition above suffices for the purpose of formalising the results of this algorithm.

8.2.2 Decomposition Rules

Decomposition rules take a single simulation constraint and try to recursively decompose the two terms in parallel until no further rules are applicable. Each decomposition step yields one or more subsequent constraints, often even a large disjunction containing alternatives. This reflects the many different alternative ground query term simulations that need to be considered in case of partial term specifications. This section begins with several notations that mostly are similar to those used in Section 4.4.

All decomposition rules are first given without examples, because the examples tend to be very extensive, and mutually depend on other decomposition rules. Section 8.2.3 illustrates important aspects of simulation unification on several more extensive examples.

Preliminaries

In the following, let l (with or without indices) denote a label, and let t^1 denote query terms and t^2 construct terms (both with or without indices). Furthermore, let $\perp$ be a special term (not occurring as subterm in any actual term) with the property that for all $t \neq \perp$ holds that $t \preceq_u \perp = False$, i.e. no term unifies with $\perp$. In the following sections, it is furthermore assumed that t^2 contains neither grouping constructs, functions, aggregations, nor optional subterms. In practice, this restriction is insignificant, because construct terms containing one of these constructs are always made ground before computing the simulation unification (see *Dependency Constraint* below).

Definition 8.5
Given two terms $t^1 = l\{t_1^1,\ldots,t_n^1\}$ and $t^2 = l\{t_1^2,\ldots,t_m^2\}$, the following sets of functions $\Pi_X : \langle t_1^1,\ldots,t_n^1\rangle \rightarrow \langle t_1^2,\ldots,t_m^2\rangle$ are defined (cf. Definition 4.6):

- $SubT^+ \subseteq \langle t_1^1,\ldots,t_n^1\rangle$ is the sequence of all non-negated subterms of t^1 and $SubT^- \subseteq \langle t_1^1,\ldots,t_n^1\rangle$ is the sequence of all negated subterms of t^1
- $SubT^! \subseteq \langle t_1^1,\ldots,t_n^1\rangle$ is the sequence of all non-optional subterms of t^1 and $SubT^? \subseteq \langle t_1^1,\ldots,t_n^1\rangle$ is the sequence of all optional subterms of t^1
- Π is the set of partial, index injective functions π from $\langle t_1^1,\ldots,t_n^1\rangle$ to $\langle t_1^2,\ldots,t_m^2\rangle$ that are total on $SubT^+$ and on $SubT^!$, each completed by $t \mapsto \bot$ for all t on which π is not defined
- Π_{mon} is the set Π restricted to all index monotonic functions
- Π_{bij} is the set Π restricted to all index bijective functions
- Π_{pp} is the set of all position *preserving* functions
- Π_{pr} is the set of all position *respecting* functions
- $\Pi_{m-pr} = \Pi_{mon} \cap \Pi_{pr}$, $\Pi_{b-pr} = \Pi_{bij} \cap \Pi_{pr}$, $\Pi_{b-pp} = \Pi_{bij} \cap \Pi_{pp}$, and $\Pi_{m-b} = \Pi_{bij} \cap \Pi_{mon}$

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

Example 8.6
Consider the terms $t^1 = f[[a, \text{without } b]]$ and $t^2 = f[a,b,c]$. The set of index monotonic mappings of the set of subterms of t^1 into the set of subterms of t^2 (Π_{mon}) is as follows (`without` b abbreviated as $\neg b$):

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

Note that all these mappings can be generated in a rather straightforward manner by creating a table with the terms $t_1^1 \cdots t_n^1$ arranged top-down and the terms $t_1^2 \cdots t_m^2$ arranged left-right and then determining paths from the first line to the n^{th} line that fulfil certain properties. This technique is called the *memoisation matrix* and described for the prototype in Appendix A.7.2.

Root Elimination

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\rightarrow$ Elimination

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

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

Descendant Elimination

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

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

Decomposition with `without`

The declarative specification of `without` in the ground query term simulation of Section 4.4 requires that a partial function (of the set of non-negated subterms into the set of subterms of the second term) is not completable to a (partial or total) function such that one of the negated subterm is mapped to a subterm in which it simulates. Since the term on the right hand side of a simulation constraint is always a data or construct term, it is sufficient to consider the case where the right term does not contain negated subterms (case 4 in Definition 4.8). For a simulation constraint $t^1 \preceq_u t^2$, the decomposition rules for the case without negated subterms is intuitively described as follows:

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

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

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

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

Decomposition Rule ***without***:

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

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

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

Note that decomposition with `without` is currently not covered in the completeness and correctness proofs of Section 8.2.4.

Decomposition with `optional` in the query term

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

- for all required subterms (i.e. not of the form `optional` t), the treatment is as before (since all negated subterms are required, they must be treated here as well, but this is omitted in the rules below to enhance readability)

- for all optional subterms, a certain number is "enabled" by adding appropriate simulation constraints, and all others are "disabled" by adding appropriate negated simulation constraints

In the following, these requirements are expressed as follows: given a partial mapping $\pi \in \Pi$ (by definition π must be total on the set of non-optional subterms, but may be partial on the set of optional subterms), it is first verified whether π yields a simulation by unifying all terms on which π is defined with their mapping (in the same manner as before). In the second part of the formula, it is then necessary to ensure that π is also the *maximal* mapping with this property, i.e. π is not completable to a mapping π' such that this would also yield a simulation. This is ensured by adding a negated disjunction testing for all mappings that are identical with π on the subterms for which π is defined, but differ on the other subterms, whether there exists an additional subterm that would unify with the subterm it is mapped to in π'. If yes, π is not maximal and completable to π'. If no, π is maximal.

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

Decomposition Rule ***optional***:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

whereas the evaluation of the second set of terms would yield

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

Note that decomposition with `optional` is currently not covered in the completeness and correctness proofs of Section 8.2.4.

Example 8.9 on page 176 illustrates the decomposition of a term containing two optional subterms. Note that more efficient evaluation techniques for the decomposition rules above are conceivable. For example, if one of the unification steps in the part for which π is defined already fails, it is not necessary to consider all different alternative mappings that are equal on the subterms on which π is defined.

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

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

Although a complete decomposition is preferable, it is (fortunately) not necessary for evaluating Xcerpt programs, as grouping constructs always depend on the bindings of the variables in the query part of a rule. Rules containing grouping constructs are treated by the *dependency constraint* (cf. Section 8.3.1), which performs an auxiliary computation for solving the query part of a rule and then substitutes the results in the rule head. Thus, in this case it is sufficient to treat the unification of a query term with a data term, which does not contain grouping constructs (and obviously also no variables).

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

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

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

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

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

Term References: Memoing of Previous Computations

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

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

Memoing. Dereferencing alone is not sufficient for treating references, because the simulation unification would not terminate in case both terms contain cyclic references. The technique used by the algorithm to avoid this problem is *memoing* (also known as *tabling*). In general, memoing is used to avoid redundant computations by storing the result of all previous computations in memory (e.g. in a table). If a computation has already been performed previously, it is not necessary to repeat it as the result can simply be retrieved from memory. This technique is among others used in certain implementations of Prolog [122, 37].

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

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

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

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

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

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

It might be somewhat surprising that the constraint is reduced to $True/False$ instead of inserting the result of a previous computation. The rationale behind this is that the result of the previous computation is already part of the current conjunct in the disjunctive normal form. *True* and *False* are the neutral elements of conjunction and disjunction, and thus terminate the unification while keeping results of previous computations. Examples 8.10 and 8.11 illustrate the simulation unification with references.

8.2.3 Examples

Since most examples for the decomposition rules are rather extensive, they are all grouped in this Section to improve readability. As in the examples in Section 4.4, the construct `optional` is sometimes abbreviated by `opt`, the construct `position` is sometimes abbreviated by `pos`, and the construct `without` is sometimes abbreviated by ¬. The latter abbreviation is unproblematic, as ¬ can otherwise never occur within a term. Some of the more complicated examples also provide a "decomposition tree" which shows the application of decomposition steps in the different conjuncts of the DNF. In these trees, nodes represent conjuncts and edges represent decompositions. If applying a simplification rule to a conjunct yields a disjunction, its corresponding node has more than one alternative successors. Read from the root to the leaves, these trees allow to follow the sequences of decomposition steps that lead to substitutions. The consistent end states of the constraint store are often emphasised by a rectangular frame.

Example 8.7 (Decomposition)
This example consists of three decompositions of simple simulation constraints. Figures 8.1, 8.2, and 8.3 provide a graphical illustration of the decompositions.

1. Consider the simulation constraint (cf. Figure 8.1)

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

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

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

   No further simplification rules are applicable.

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

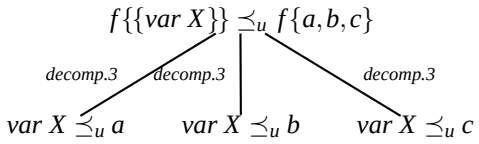

Figure 8.1: Derivation tree of $f\{\{var\ X\}\} \preceq_u f\{a,b,c\}$ (Example 8.7, part 1). Different paths denote different alternatives, nodes represent conjuncts, and edges represent applications of simplification rules.

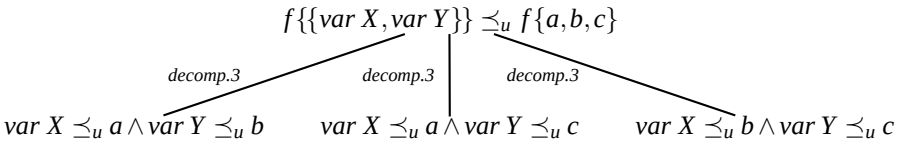

Figure 8.2: Derivation tree of $f[[var\ X, var\ Y]] \preceq_u f[a,b,c]$ (Example 8.7, part 2). Different paths denote different alternatives, nodes represent conjuncts, and edges represent applications of simplification rules.

2. Consider the simulation constraint (cf. Figure 8.2)

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

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

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

3. Consider the simulation constraint (cf. Figure 8.3)

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

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

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

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

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

   Simulation unification in all three conjuncts yields

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

   and formula simplification simplifies this constraint store to

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

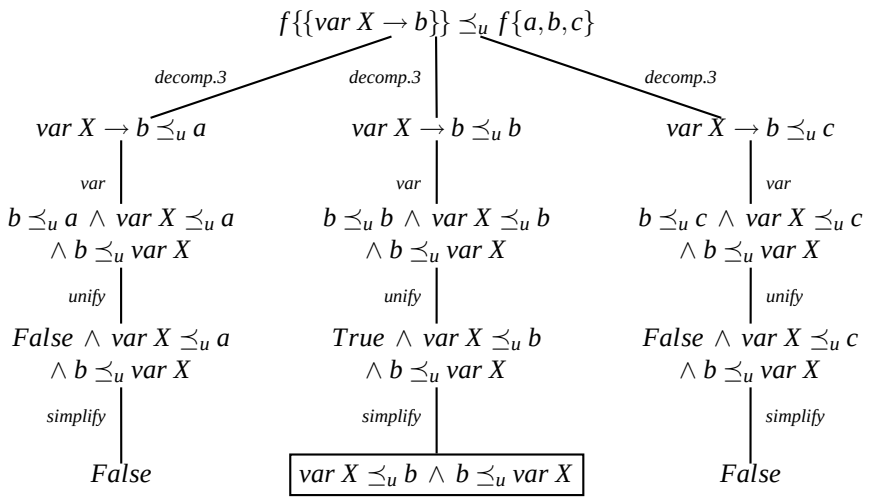

Figure 8.3: Derivation tree of $f\{\{var\ X \to b\}\} \preceq_u f\{a,b,c\}$ (Example 8.7, part 3). Different paths denote different alternatives, nodes represent conjuncts, and edges represent applications of simplification rules.

Example 8.8 (Simulation Unification with `without`)

1. Consider

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

   The set Π of partial mappings that are total on $SubT^+$ is as follows (partial mappings completed by mapping undefined values to $\bot$)[3]:

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

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

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

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

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

   and formula simplification results in

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

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

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

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

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

[3] note that $without\ b$ is abbreviated by $\neg b$

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

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

Evaluating the constraints contained in the negated subformulas yields:

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

and formula simplification results in

$$a \preceq_u c$$

which of course can be further decomposed to False.

Example 8.9 (Simulation Unification with `optional`)
Consider the constraint $C = f[[a, \text{opt } g\{\mathit{var}\, X\}, \text{opt } h\{\mathit{var}\, Y\}\,]] \preceq_u f[a, g\{b\}]$

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

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

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

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

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

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

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

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

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

Example 8.10 (Simulation Unification with References)
Consider the simulation constraint

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

In the following, the sequence of decomposition steps that result in a complete simulation unification of the simulation constraint is described. For each conjunct, the set $\mathcal{H}_i$ denotes the current memoing history of the conjunct. So as to better distinguish the path that lead to this history, the index is composed of the numbers of the branches followed in previous steps. For example, $\mathcal{H}_121$ is the history of the node that can be located by following the first branch on the top level, the second branch on the second level, and the first branch on the third level. Note that Figure 8.4 gives a graphical representation of the decomposition tree that might be easier to read. In this tree, the history of a node is easily determined by following the path from the root node to the current node, and thus not given explicitly. The first decomposition step yields

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

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

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

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

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

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

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

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

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

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

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

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

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

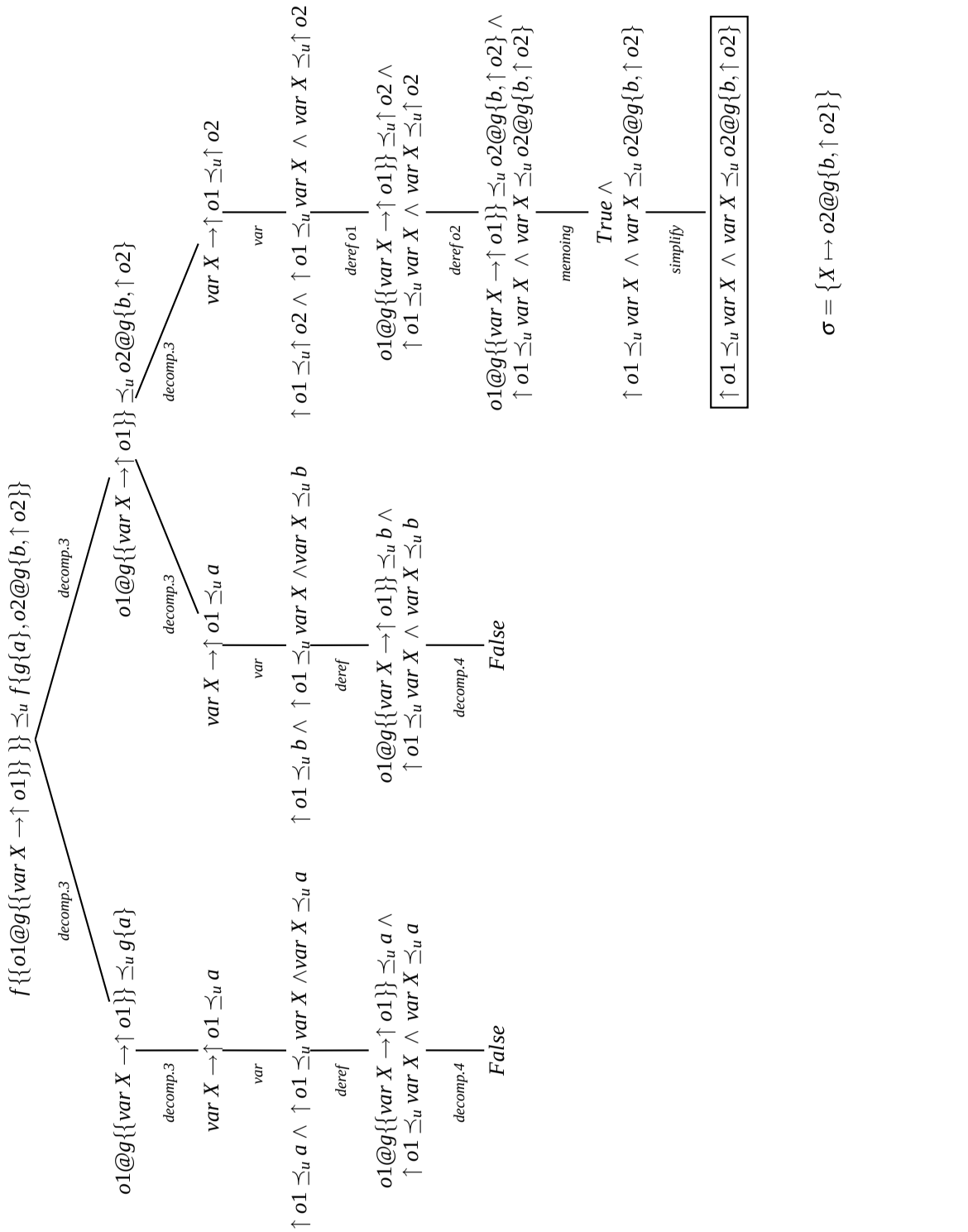

Figure 8.4: Derivation tree of $f\{\{o1@g\{\{var\ X \to\uparrow o1\}\}\}\} \preceq_u f\{g\{a\}, o2@g\{b,\uparrow o2\}\}$ (Example 8.10). The memoing history $\mathcal{H}$ of a node is represented by the path from the root to that node.

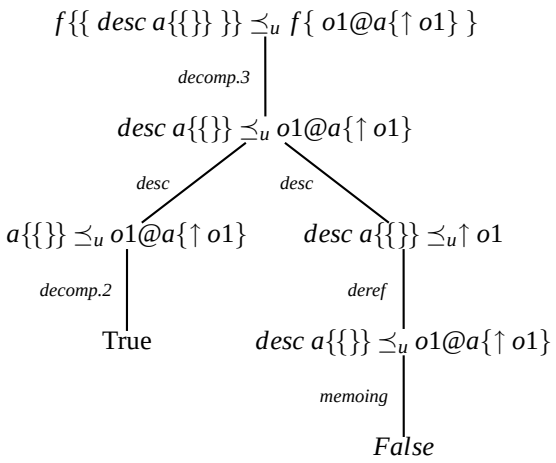

Figure 8.5: Derivation tree of $f\{\{\ desc\ a\{\{\}\}\ \}\} \preceq_u f\{\ o1@a\{\uparrow o1\}\ \}$ (Example 8.11). In this graph, the memoing history $\mathcal{H}$ of a node is represented by the path from the root to that node.

Example 8.11 (Simulation Unification with References and Descendant)
Consider the simulation constraint

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

The sequence of decomposition steps is as follows (cf. Figure 8.5 for a graphical illustration). The first decomposition step (*decomp.3*) yields

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

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

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

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

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

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

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

8.2.4 Soundness and Completeness

The following theorem shows soundness and completeness for the simulation unification algorithm applied to a simulation constraint of the form $t^q \preceq_u t^c$. t^q is assumed to not contain subterm negation or optional subterms. Also, as rules with grouping constructs are always evaluated in an auxiliary computation using the dependency constraint, it is assumed that t^c does not contain grouping constructs. Furthermore, t^c is assumed not to contain functions, aggregations or optional subterms.

Theorem 8.6 (Soundness and Completeness of Simulation Unification)
Let t^q be a query term without subterm negation and optional subterms and let t^c be a construct term without grouping constructs, functions/aggregations, and optional subterms. A substitution set Σ is a most general simulation unifier of t^q and t^c if and only if the simulation unification of $t^q \preceq_u t^c$ terminates with a constraint store CS such that $\Sigma = \Omega(CS)$.

Proof. cf. Appendix B.2 □

8.3 Backward Chaining

The backward chaining algorithm presented here is inspired by the SLD resolution calculus used in logic programming [71]. However, traditional approaches like SLD resolution do not account well for Xcerpt constructs like partial term specification or grouping constructs. Both kinds of constructs seriously influence the resolution calculus:

High Branching Rate. In traditional logic programming, there are two elements of nondeterminism that lead to branching in the proof tree: selection of the predicate to unfold in the evaluation of a rule body, and the selection of the program rule used for further chaining. Xcerpt's usage of partial patterns adds a third element: When using partial patterns, there is in general no single way to match two terms. Instead, all possible alternative matchings have to be considered, which leads to a significantly higher branching rate.

Grouping Constructs `all` *and* `some`. Unlike Prolog's *setof* and *bagof* predicates, the grouping constructs `all` and `some` are an integral part of the language. It is hence desirable to support such higher order constructs in the proof calculus itself rather than treating them as external predicates.

In the following, a backward chaining algorithm based on constraint solving is introduced. It makes use of the simple constraint solver of Section 8.1 and the simulation unification algorithm of Section 8.2. In this algorithm, it is assumed that Xcerpt programs are range restricted, stratified, and separated apart (cf. Chapter 6). Evaluation always begins with a single, folded query constraint, i.e. a single constraint of the form $\langle Q \rangle$ for some goal $t^c \leftarrow_g Q$, and terminates when the constraint store either fails or is sufficiently solved to produce the answer term for the goal.[4] "Sufficiently" currently means that the constraint store is solved completely, but it might be desirable to investigate optimisations based on the construct term t^c of the goal that solve only relevant parts of the constraint store.

Instead of using backtracking to evaluate rule chaining, the backward chaining algorithm for Xcerpt uses disjunctions in the constraint store to represent alternatives. In this manner, it is possible to use other selection strategies than depth-first search for the selection of paths to evaluate. This is desirable as the `all` construct requires to find all solutions to a query anyway.

Note that the algorithm does not necessarily terminate for any input, as programs may contain recursive rules that produce infinite chains. As it is desirable to have this expressive power in Xcerpt, it is the duty of programmers to ensure that programs terminate. Non-termination might also be desirable, e.g. to produce continuous streams of data (together with the `all` construct), but such applications have not yet been investigated in detail (cf. Section 9.5.2).

The following Sections first introduce the dependency constraint as a means to treating the grouping constructs `all` and `some`, functions, and aggregations by performing an auxiliary computation. Afterwards, simplification rules for unfolding folded queries are discussed, which also implement the main part of the algorithm Different approaches to backward chaining in Xcerpt have been considered in the course of this thesis [25, 94]. The approach presented here is a further refinement of the "all at once" approach presented in [25].

8.3.1 Dependency Constraint

The dependency constraint is of the form $(t_1 \preceq_u t_2 \mid D)$ for a simulation constraint $t_1 \preceq_u t_2$ and some constraint D (usually a folded query) and expresses a temporal and functional dependency between $t_1 \preceq_u t_2$ and D. A dependency constraint of the form above requires to completely evaluate the constraint D in an auxiliary computation (also considering other constraints with which the dependency constraint is in conjunction) before $t_1 \preceq_u t_2$, and applies the substitution resulting from the evaluation of D to t_2 (application to t_1 is not necessary as the terms t_1 and t_2 stem from different rules and are thus variable disjoint). If the evaluation of D fails, then the dependency constraint also fails without evaluating $t_1 \preceq_u t_2$. The following simplification rule formalises this treatment:

$$\frac{(\ t_1 \preceq_u t_2 \mid D\ )}{\bigvee_{t_2' \in \Sigma(t_2)} t_1 \preceq_u t_2'} \quad \Sigma = subst(solve(D))$$

[4] Recall that the result of a goal is always either failure or a single data term.

Note that if Σ is empty (i.e. there is no solution for D), the set $\Sigma(t_2)$ is empty and thus the result of the evaluation is the empty disjunction, which simplifies to *False*. In case the evaluation of D yields simply *True*, the resulting substitution set Σ is not empty, but contains the empty substitution (identity).

The dependency constraint is necessary because the (incomplete) simulation unification with a construct term containing the grouping constructs `all` or `some`, or functions and aggregations, usually does not sufficiently characterise the possible bindings of the variables in the two terms.

In order to detect inconsistencies early (and avoid unnecessary recursion), it is reasonable to perform a partial unification between the query term and the construct term and add that result to D in order to exclude such cases for which no answer can exist. Consider for instance the simulation constraint $f\{\{g\{var\ X\}\}\} \preceq_u f\{all\ h\{var\ Y\}\}$). A partial unification could determine that for all Y must hold that $g\{\cdot\} \preceq_u Y$, but not $g\{var\ X\} \preceq_u Y$ as this would possibly yield inconsistent restrictions for the variable X. The following refinement of the rule above uses the incomplete decomposition of `all` and `some` to add such information:

$$\frac{(\ t_1\ \preceq_u\ t_2\ \mid\ D\ )}{\bigvee_{t_2' \in \Sigma(t_2)}\ t_1\ \preceq_u\ t_2'} \quad \Sigma = subst(solve(D \wedge t^1 \preceq_u t^2))$$

8.3.2 Query Unfolding

The rules for query unfolding take a folded query constraint of the form $\langle Q \rangle$ and evaluate it by "unfolding" it. For and/or connected queries, this simply means to distribute the evaluation to the subqueries and connect the corresponding folded query constraints with the respective connectives. For query terms (i.e. atomic queries), this means either to query the terms at the associated resource, or to query the construct parts of program rules. In both cases, the algorithm reverts to simulation unification for determining the solution. In case a query term queries the construct parts of program rules, it is furthermore necessary to evaluate the respective query parts of the rules and to take care of grouping constructs that possibly occur in the construct part of rules. The following query unfolding rules are used:

And/Or-Connection The connectives *and* and *or* are simply mapped to their counterparts in the constraint store. The rules for *and* and *or* are therefore straightforward:

$$\frac{\langle \mathtt{and}\{Q_1,\ldots,Q_n\}\rangle_{\mathcal{R}}}{\langle Q_1\rangle_{\mathcal{R}} \wedge \cdots \wedge \langle Q_n\rangle_{\mathcal{R}}} \quad \frac{\langle \mathtt{or}\{Q_1,\ldots,Q_n\}\rangle_{\mathcal{R}}}{\langle Q_1\rangle_{\mathcal{R}} \vee \cdots \vee \langle Q_n\rangle_{\mathcal{R}}}$$

Note that the resource specification $\mathcal{R}$ is distributed recursively, and that in particular, $\mathcal{R}$ may be empty (i.e. $\mathcal{R} = \emptyset$).

Query Negation Xcerpt query negation is negation as failure (NaF), and evaluated in an auxiliary computation very much like the dependency constraint. The result of this auxiliary computation is a constraint formula C specifying which variable bindings are disallowed for the variables occurring in Q. It is thus first restricted to constraints containing variables that occur in Q and then added negated to the original constraint store. The consistency verification rules 3–5 of the constraint solver ensure that variables cannot be bound to values disallowed by C.

$$\frac{\langle \mathtt{not}\ Q\rangle_{\mathcal{R}}}{\neg C} \quad V = vars(Q), C = restrict(V, solve(\langle Q\rangle_{\mathcal{R}}))$$

Resource Specification In the case where the query is the specification of an input resource, this resource needs to be retrieved. The function $retrieve(RSpec)$ takes a resource specification of any form (e.g. an URI together with a format specification of "xml" such that it can be parsed correctly) and returns a set of data terms corresponding to this resource. Note that it is also possible that a resource contains more than one term, e.g. when the resource is another Xcerpt program.

$$\frac{\langle in\{RSpec, Q\}\rangle_{\mathcal{R}'}}{\langle Q\rangle_{\mathcal{R}}} \quad \mathcal{R} = retrieve(RSpec)$$

Note that the old resource specification $\mathcal{R}'$ is shadowed by the new resource specification $\mathcal{R} = retrieve(RSpec)$

Query Term Two simplification rules process query terms. The first rule considers query terms with associated resources. In this case, the query term is unfolded to a *disjunction* of simulation constraints, one constraint for each resource. The intuitive meaning is "query any of the given resources".

$$\frac{\langle t^q \rangle_{\{t_1,\ldots,t_n\}}}{t^q \preceq_u t_1 \vee \cdots \vee t^q \preceq_u t_n}$$

The second query term unfolding works on such query terms that have *no* resource associated. In such a case, the query term is evaluated against all rules in the program. For each rule containing grouping constructs, functions, or aggregations, a dependency constraint is added which evaluates the unification between the query term and the head of the rule only, if the body of the rule can be evaluated successfully and the result can be applied to the rule head. For each rule not containing a grouping construct, the folded query is replaced by a simulation constraint between the query term and the construct term of the rule together with the (folded) query part of the rule. Each rule evaluation is an alternative, hence the result is a disjunction of constraints.

In the following, let $\mathcal{P}_{grouping} \subseteq P$ be the set of program rules $t^c \leftarrow Q$ such that t^c contains grouping constructs, functions, aggregations, or optional subterms, and let $\mathcal{P}_{nongrouping} \subseteq P$ bet the set of program rules $t^c \leftarrow Q$ such that t^c does not contain grouping constructs, functions, aggregations, or optional subterms. Note that goals are not considered in either case, as they do not participate in chaining. Furthermore, $n \geq 0$ and $m \geq 0$.

$$\frac{\langle t^q \rangle_{\emptyset}}{\bigvee_{t^c \leftarrow Q \in \mathcal{P}_{grouping}} (t^q \preceq_u t^c \mid \langle Q \rangle_{\emptyset}) \vee \bigvee_{t^c \leftarrow Q \in \mathcal{P}_{nongrouping}} t^q \preceq_u t^c \wedge \langle Q \rangle_{\emptyset} \vee \bigvee_{t^d \in P} t^q \preceq_u t^d}$$

8.3.3 Examples

This Section contains several examples that show various aspects of the evaluation algorithm. Like for simulation unification in Section 8.2.3 above, the examples are also illustrated in derivation trees. Nodes represent conjuncts, edges represent applications of simplification rules, and different nodes on the same level are alternatives. Each non-*False* leaf node in these trees represents an alternative solution of the evaluation.

Example 8.12 (Chaining)
Consider the following Xcerpt program (represented in compact notation and with internalised resources):

$f\{var\ X\} \leftarrow and\{g\{\{var\ X\}\}, h\{var\ X\}\}$
$g\{a, b\}$
$h\{b\}$

Figure 8.6 shows the evaluation of the query $\langle f\{var\ R\}\rangle$. Note the use of consistency verification rules in some of the lower parts of the tree.

Example 8.13 (Chaining, Query Negation)
Consider the following Xcerpt program (represented in compact notation and with internalised resources):

$f\{var\ X\} \leftarrow and\{g\{\{var\ X\}\}, not\ h\{var\ X\}\}$
$g\{a, b\}$
$h\{b\}$

Figure 8.7 shows the evaluation of the query $\langle f\{var\ R\}\rangle$. Note the use of consistency verification rules in some of the lower parts of the tree and the auxiliary computation used for evaluating the negated part. This auxiliary computation is indicated by the dashed line and evaluates $\langle h\{var\ X\}\rangle$ like in Example 8.12 before to $var\ X \preceq_u b$.

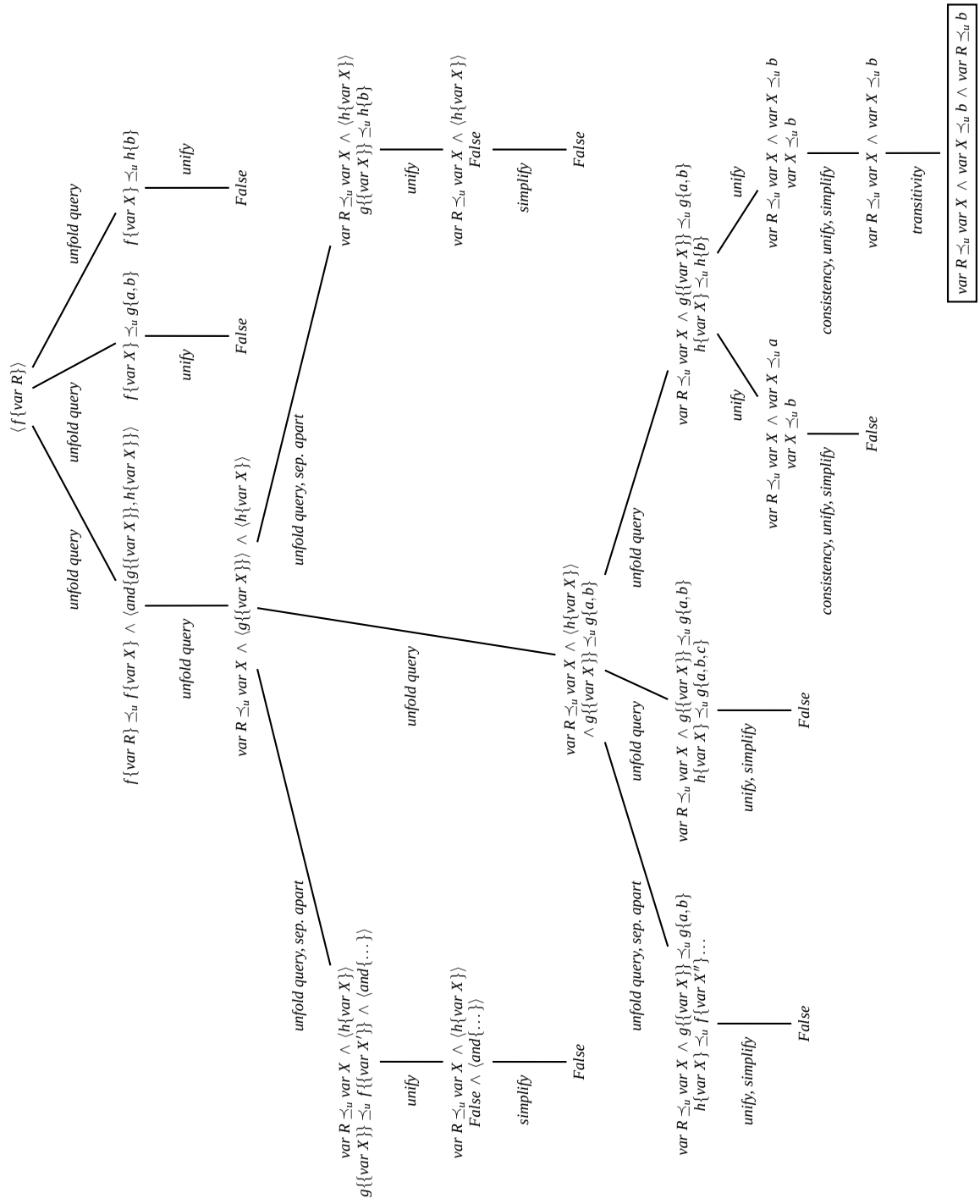

Figure 8.6: Derivation tree showing the evaluation of $\langle f\{var\ R\}\rangle$ with the program in Example 8.12 (chaining).

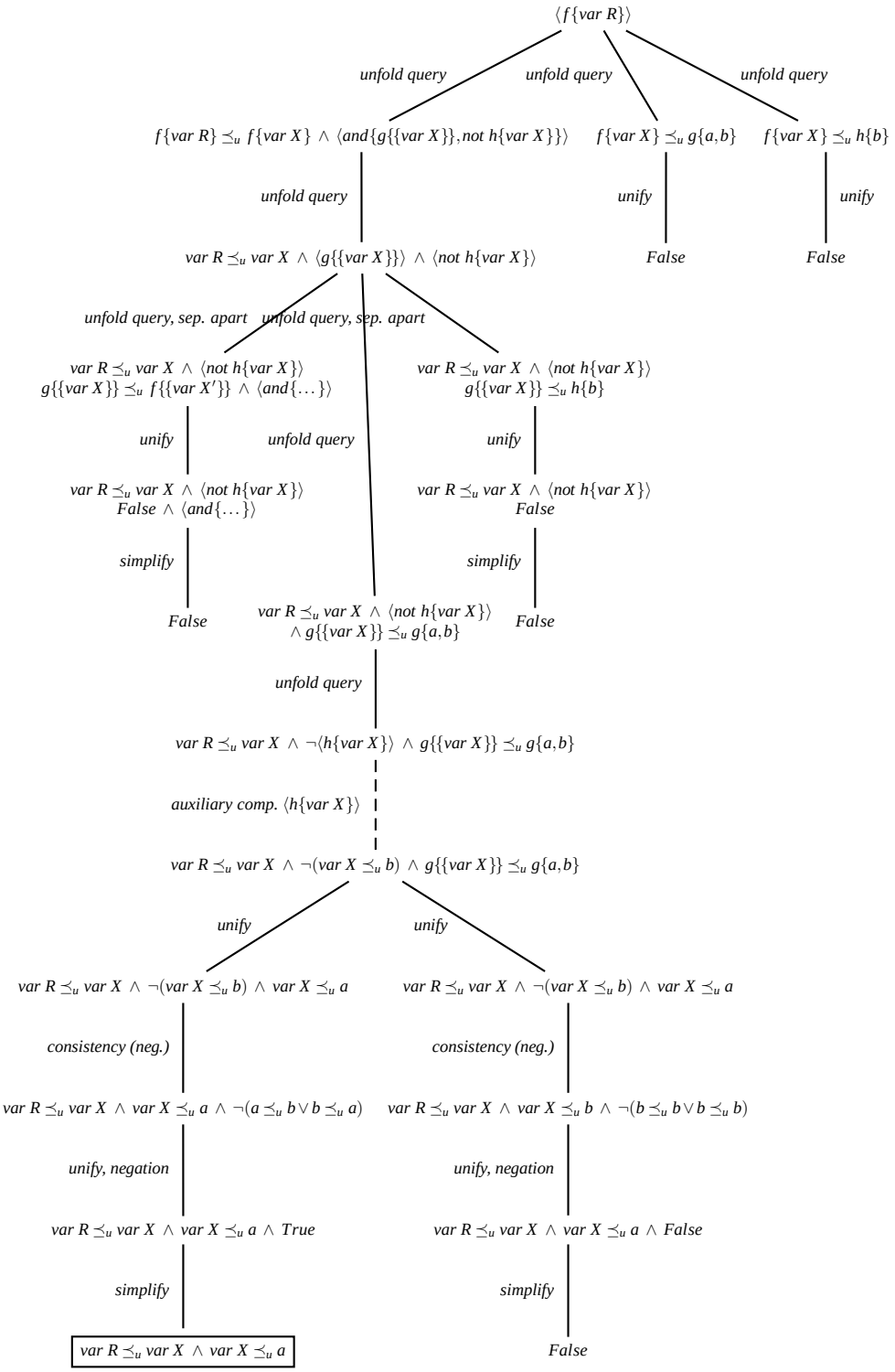

Figure 8.7: Derivation tree showing the evaluation of $\langle f\{var\ R\}\rangle$ with the program in Example 8.13 (chaining with negation).

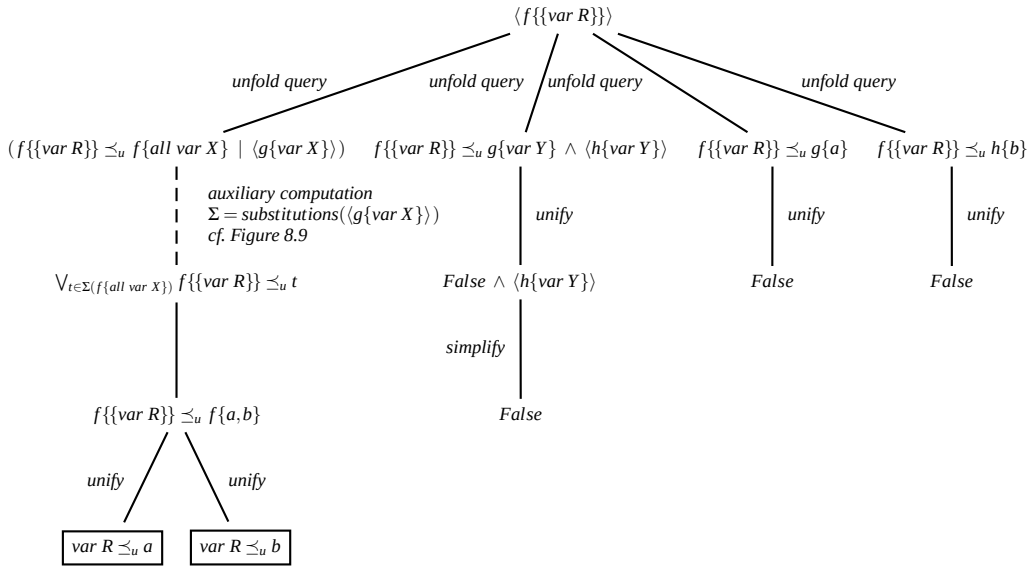

Figure 8.8: Derivation tree showing the evaluation of $\langle f\{var\ R\}\rangle$ with the program in Example 8.14 (chaining with grouping constructs).

Example 8.14 (Chaining, Grouping Constructs)
Consider the following grouping stratifiable Xcerpt program (cf. Examples 6.5 and 7.11):

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

The evaluation of a query $f\{\{var\ R\}\}$ in this program is shown in Figures 8.8 (main evaluation) and 8.9 (auxiliary computation of $g\{var\ X\}$ for the dependency constraint). Note that the evaluation of the dependency constraint in Figure 8.9 uses the incomplete unification with `all` to avoid unnecessary auxiliary computations.

8.3.4 Soundness and Completeness

In this section, it is shown that the backward chaining algorithm is sound with respect to the fixpoint semantics described in Section 7.5, and that it is complete in all cases where the algorithm terminates. This completeness result is weak, but appears to be inherent to backward chaining. As rules with grouping constructs in the rule head require the body to be maximally satisfied (cf. Chapter 7), the proofs for soundness and completeness are tightly interweaved. We therefore first show the following Lemma, which is at the core of both soundness and (weak) completeness. Recall that $\Omega(CS)$ denotes the solution set of a constraint store CS.

Lemma 8.7
Let P be a negation-free, grouping stratified Xcerpt program without goals, let M_P be the fixpoint of P, and let Q be a negation-free query (composed of one or more query terms). If the evaluation of $\langle Q\rangle$ terminates with a constraint store CS, then $\Sigma = \Omega(CS)$ is a maximal substitution set with $M_P \models \Sigma(Q)$.

Proof. cf. Appendix B.3 □

This Lemma contains almost all necessary "ingredients" for both soundness and completeness: it states that the solution set of the resulting constraint store is a maximal (i.e. "complete") substitution set for the satisfaction (i.e. "soundness") of the query part of a goal.

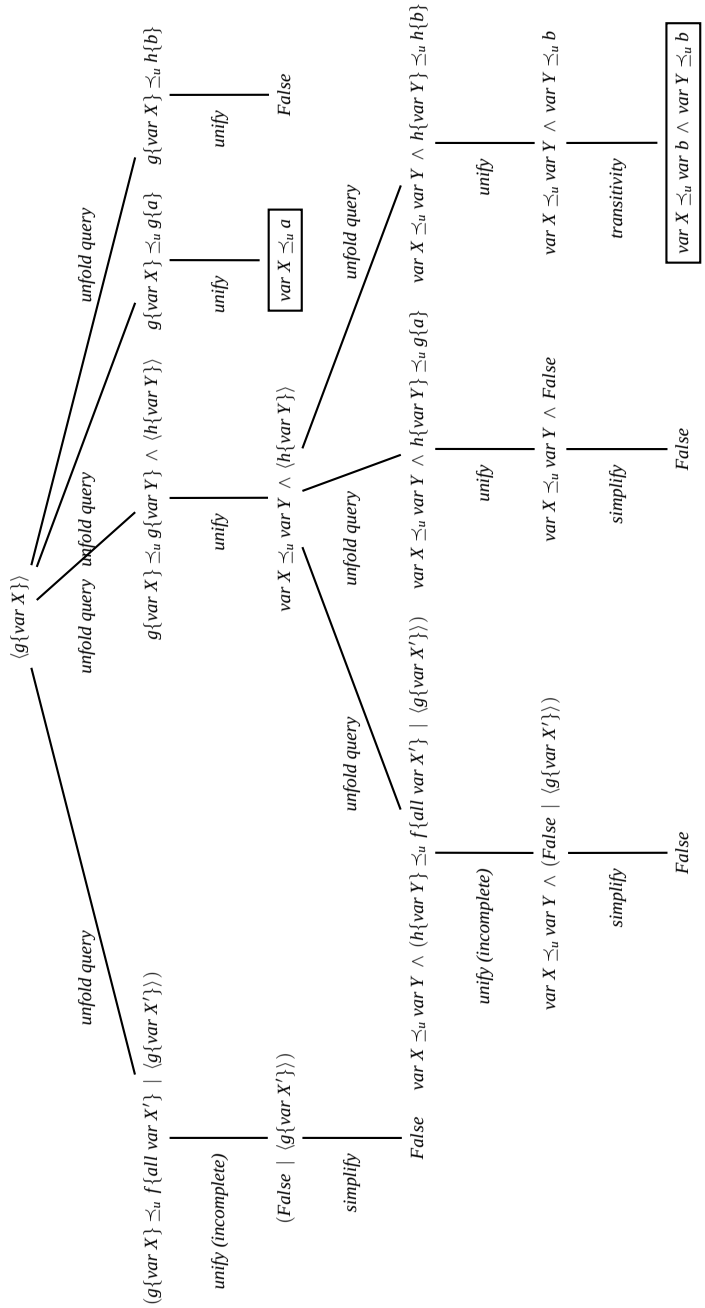

Figure 8.9: Derivation tree showing the auxiliary computation of $\langle g\{var\ X\}\rangle$ with the program in Example 8.14 (chaining with grouping constructs). The different resulting substitutions are highlighted by frames. Note that the incomplete unification with `all` is used in the dependency constraints to avoid unnecessary auxiliary computations.

Recall for the remainder of this section that goals differ from rules in that the ground instances of the goal heads cannot be queried by query terms. This difference is not reflected in the declarative semantics described in Chapter 7, but can be achieved by ensuring that no query term simulates into a ground instance of a goal head, e.g. by wrapping goal heads as subterms of a term with a label not used elsewhere in the program.

Soundness

Theorem 8.8 (Soundness of the Backward Chaining Algorithm)
Let P be a negation-free, grouping stratified Xcerpt program, and let $G = t^c \leftarrow_g Q$ be a goal in P. If the evaluation of Q in P terminates with a constraint store $\mathcal{CS}$ inducing a grounding substitution set $\Sigma = \mathit{substitutions}(\mathcal{CS})$, then $\Sigma(t^c)$ is a subset of the fixpoint M_P of P.

Proof. Let P be a negation-free, grouping stratified Xcerpt program, and let $G = t^c \leftarrow_g Q$ be a goal in P. Assume that $P' \subseteq P$ is P without the goals. According to Lemma 8.7, evaluation of $\langle Q \rangle$ in P' terminates with a constraint store $CS = D_1 \vee \cdots \vee D_n$ in disjunctive normal form such that the substitution set $\Psi = \Omega(CS)$ is a maximal substitution set with $M_{P'} \models \Psi(Q)$.

As the results of goals do not participate in rule chaining, adding the goals to P' does not influence the other rules in P' and only adds new data terms to $M_{P'}$. Thus, also for M_P holds that $M_P \models \Psi(Q)$, and Ψ is maximal. $\Psi(t^c) \subseteq M_P$ then follows from the definition of T_P. Furthermore, because P is range restricted, it holds that every variable X in t^c appears in every conjunct D_i in a simulation constraint of the form $X \preceq_u t$. Hence, with Corollary 8.2 follows that $\mathit{substitutions}(CS) = \Omega(CS)_{|V}$, where V is the set of variables occurring in t^c. Thus, $\mathit{substitutions}(CS)$ yields the same ground instances of t^c as $\Psi = \Omega(CS)$. The backward chaining algorithm is thus sound. □

Completeness

In general, backward chaining is incomplete with respect to the fixpoint semantics described in Chapter 7. This is easy to see on a small example. Consider the program

$$f\{a\} \leftarrow f\{a\}$$
$$f\{a\}$$

The fixpoint for this program obviously is simply $\{f\{a\}\}$. However, evaluation of e.g. $f\{\mathit{var}\ X\}$ does not terminate in the backward chaining evaluation, because the rule in the program above is applicable infinitely often. This problem is not particular to Xcerpt: other logic programming languages like Prolog terminate neither with such programs.

To solve this, SLD resolution [71] uses a *fairness* clause that states that every clause (i.e. rule or data term) must be used eventually, which ensures that SLD resolution determines an answer after finitely many steps, if an answer exists. Unfortunately, this fairness clause is not applicable in Xcerpt, because the grouping constructs require to retrieve *all* solutions to a query, whereas fairness only guarantees to find *one* solution after finitely many steps. Consider for example the program

$$g\{\mathit{all}\ \mathit{var}\ X\} \leftarrow f\{\mathit{var}\ X\}$$
$$f\{a\} \leftarrow f\{a\}$$
$$f\{a\}$$

This program is grouping stratifiable and the fixpoint of this program is obviously $\{f\{a\}, g\{a\}\}$. Construction of the result $g\{a\}$ however requires to retrieve all solutions to $f\{\mathit{var}\ X\}$; a single solution does not suffice because it violates the maximality requirement in the semantics of the `all` construct.

Hence, we restrict the statement of completeness to negation-free, grouping stratified Xcerpt programs *for which the evaluation algorithm terminates*. This result is obviously somewhat unsatisfactory, because any non-terminating program would be complete under this assumption. We therefore also give criteria and suggest enhancements that ensure that programs terminate (in case the fixpoint is finite).

Theorem 8.9 (Weak Completeness of the Backward Chaining Algorithm)
Let P be a negation-free, grouping stratified Xcerpt program, with a stratification $P = P_1 \uplus \cdots \uplus P_m$ ($m \geq 1$), and let $G = t^c \leftarrow_g Q$ be a goal in P such that the evaluation of Q terminates. Assume that P has a fixpoint

$M_P = T_P^\omega(P)$. If the evaluation of Q in P terminates with a constraint store CS, then CS induces a *maximal* substitution set Σ with $\Sigma(t^c) \subseteq M_P$ (i.e. there exist no other ground instances of t^c in M_P).

Proof. By Theorem 8.8, evaluation of Q in P yields a constraint store CS inducing a substitution set Σ with $\Sigma(t^c) \subseteq M_P$. Hence, we only have to show that Σ is also maximal wrt. t^c, i.e. there exists no Σ' with $\Sigma_{|V} \subseteq \Sigma'_{|V}$ for the set of variables V occurring in t^c.

From Lemma 8.7, we know that the evaluation of $\langle Q \rangle$ in P terminates with a constraint store CS such that $\Psi = \Omega(CS)$ is a maximal substitution with $M_P \models \Psi(Q)$, and thus $\Psi(t^c) \subseteq M_P$. Furthermore, Ψ is maximal wrt. to Q. As by definition of goals, no ground instances of t^c besides those produced by the goal may exist[5], Ψ is thus also maximal wrt. $\Psi(t^c) \subseteq M_P$. Also, we have already seen in the proof of Theorem 8.8 that $\Sigma = \mathit{substitutions}(CS) = \Omega(CS)_{|V}$ where Σ yields the same ground instances of t^c as $\Omega(CS)$. Thus, Σ is also maximal wrt. $\Sigma(t^c) \subseteq M_P$. □

Criteria for Termination

No Recursion. Disallowing recursion is an obvious way to ensure termination. This restriction appears very strict on a first glance. However, due to the powerful grouping constructs `all` and `some`, this restricted class still allows many useful programs that would require recursion in traditional logic programming. For example, the program computing the sum of rows and columns in an HTML table described in Section 5.1.3 didn't use recursion despite the rather complex task. Likewise, many of the other examples of Chapter 5 did not require recursion while still being useful programs.

Of course, as has been argued before, there are many applications that still require recursion. It is therefore important to study refinements of this restriction that disallow only certain kinds of recursion. A useful candidate are programs where only the ground instances of rules are non-recursive (so-called *locally hierarchical programs* [87]).

Retrieving only Some Solutions. In many cases, it is actually not necessary to retrieve all solutions of the constraint store, e.g. when the rules that depend on the recursion do not contain grouping constructs. Also, a user might be satisfied with results that can be delivered in a certain time span. For both cases, the change to the evaluation algorithm would only be minor: instead of iterating as long as a rule can be applied to the constraint store, the function `solve(·)` (Section 8.1.6) would need to terminate as soon as one of the conjuncts of the constraint store is completely solved. Also, a fair rule application strategy would be necessary (e.g. breadth-first search or some other complete search strategy).

Tabling. Tabling [37] is a technique (used e.g. in XSB Prolog) where redundant and non-terminating rule applications are avoided by caching the results of previous applications, and is known to terminate more often than the SLD resolution used in standard Prolog [105]. In particular, it avoids the problem described above.

[5] otherwise, disambiguation is possible because results of goals do not participate in rule chaining

Part III

Conclusion

CHAPTER

NINE

Perspectives

This thesis only provides the foundations for the language Xcerpt. In its current state, Xcerpt is not suited for usage in practical applications, because it lacks many desirable constructs and the current implementation is rather inefficient and implements the presented language constructs only in parts. This chapter provides perspectives for future research that might contribute to the success of the language.

9.1 Advanced Query Constructs

As shown in this thesis, the language Xcerpt in its current form is well suited for a number of Web applications. However, Xcerpt only provides a number of core constructs. Many queries are therefore either not possible or rather complicated to express. This section briefly describes areas where additional constructs might be desirable and in some cases suggests concrete language extensions that have been thought of.

9.1.1 Advanced Text Processing

Text Querying Beyond Markup Boundaries

Most XML documents on the Web do not represent semistructured "databases" but rather text content with markup serving various purposes, including layout (as in HTML), text structure, and annotations. In the form presented in this thesis, Xcerpt only provides regular expressions for text processing. However, for advanced text processing, e.g. searching for certain sentences regardless of whether they are interrupted by markup or not, regular expressions do not suffice. Consider for example the following fragment of Goethe's poem "Der Zauberlehrling"[1]. Markup is added to indicate beginning and end of a verse and beginning and end of a line.

```
<verse>
  <line>Hat der alte Hexenmeister</line>
  <line>sich doch einmal wegbegeben!</line>
  <line>Und nun sollen seine Geister</line>
  <line>auch nach <emph>meinem</emph> Willen leben.</line>
</verse>
```

For many applications, it is interesting to query ignoring the intermediate markup. For example, it might be desirable to query for verses where the words "nach meinem Willen" occur in a sentence. Advanced text querying constructs are necessary to allow for this kind of querying. While a simple modification of regular expressions appears possible, there might be implications when combining these with regular query

[1] english: "The Sorceror's Apprentice"

patterns. E.g. in the example above, one might want to query for the sentence “nach meinem Willen” where at least one of the words is emphasised.

Markup Overlap

Likewise, it is often necessary or interesting to consider documents with so-called *overlap*, i.e. where the enclosed range of one element partly overlaps with the boundaries of another element [125, 44]. Such representations are required when representing different kinds of markup for the same text document (e.g. layout, structure, and annotations), and often occur in computational linguistics. Consider for example the following (not well-formed) document (text by Mark Twain). Lines and sentences are marked up. Obviously, they overlap as not every sentence begins and ends in exactly one line:

```
<line><sentence>Don’t go around saying the world</line>
<line>owes you a living.</sentence><sentence>The world owes you</line>
<line>nothing.</sentence><sentence>It was here first.</sentence></line>
```

There are various means to represent such markup: in a single document (but then it is not well-formed XML), in several documents (containing much redundancy), using distinguished empty elements for start and end tags, or by adding “layers” of markup to a base document. Interesting queries to such documents would e.g. be “all lines ignoring sentences” or “sentences beginning in line 2”. Supporting such queries would be interesting for many text processing tasks.

9.1.2 Duplicate Elimination

The grouping constructs `all` and `some` currently do not eliminate duplicate variable bindings when building ground instances of construct terms, because such duplicate elimination is usually computationally expensive and often not desirable (e.g. if an address book contains two person entries with name “John Smith”, then they represent two distinguished persons). For some applications it might however be useful to let the programmer specify a grouping with duplicate elimination. A possible approach is to add a new keyword `unique` that may be used together with `all` and `some`. Consider for example the construct term:

```
unique-entries {
  all unique entry {
    name { var Name },
  }
}
```

This construct term would ensure that for every binding of `Name` there exists exactly one `entry` subterm in the resulting data term.

Extending the model theory and evaluation to take into account this construct is not difficult. However, further refinements are possible: for example, it might be desirable to eliminate duplicates as early as possible so as to keep the constraint store small.

9.1.3 Advanced Filter and Exclusion Mechanisms

Many applications demand to filter out certain subterms of a term while retaining the overall nesting structure of the remaining subterms. For example, one might create the table of contents of a book represented as an XML document (cf. the example in Section 2.4.3) by filtering out all subterms besides the `chapter`, `section`, and `title` subterms. Such filtering can be implemented in Xcerpt by using recursive rules for structural recursion over the input document. However, implementing this recursion can be rather inconvenient and inefficient, and thus special purpose constructs are desirable. In David Maier’s characteristics of a Web query language (Section 3.2), such constructs are summarised under the *reduction* query operation.

This section proposes two so-called *filtering constructs*. As subterms of a data term are selected in Xcerpt by binding to variables in a query term, both constructs may only be used inside a pattern restriction for a query term variable. To distinguish this "filtering" pattern restriction from the usual pattern restriction, it uses the arrow ⇝ instead of →. The first construct is called *plus* and filters out from the bindings of the variables all subterms besides those matched by subterms of the query term explicitly marked by the keyword `plus`. The second construct is the symmetric counterpart called *minus* and filters out from the bindings all subterms matched by subterms of the query term marked by the keyword `minus`, but leaves all others.

Both constructs may be used in front of every subterm within the ⇝ pattern restriction of a variable in a query term. This in particular includes subterms containing further variables. Every subterm in a ⇝ pattern restriction is still required to match, even if it is prefixed by `minus`. If a different behaviour is desirable, `plus` and `minus` may be combined with other Xcerpt constructs like `optional`. Also, a combination with constructs like `desc` is useful, as shown in the following examples:

Example 9.1 (`minus`)
Consider the XML document `reviews.xml` described in the bookstore scenario in Section 2.4.2. The following query term binds book entries to the variable `Book`, but excludes all `review` subterms from the bindings:

```
reviews {{
  var Book ⇝ entry {{
    minus review {{ }}
  }}
}}
```

Example 9.2 (`plus`)
Consider the "thesis" example from Section 2.4.3. From this thesis, the table of contents can be retrieved by using the following query term with the filtering construct `plus`. Only those sections are taken that contain at least one paragraph.

```
var TOC ⇝ plus report {{
  desc plus chapter {{
    plus title { plus /.*/ }
    optional desc plus section {{
      plus title { plus /.*/ }
      paragraph {{ }}
    }}
  }}
}}
```

Note that all subterms except for the one labelled `paragraph` are marked with `plus`. This means that only `section` subterms with a `paragraph` subterm are considered, but the `paragraph` subterms are not part of the variable binding for `TOC`.

Note that `plus` and `minus` influence the meaning of the unmarked subterms: if subterms are marked with `plus`, then unmarked subterms are not part of the variable binding, and if subterms are marked with `minus`, then unmarked subterms are part of the variable binding. Therefore, a reasonable restriction is that `plus` and `minus` may not be used together for subterms of the same parent.

Interestingly, `plus` and `minus` require considerable modifications to the ground query term simulation and simulation unification, because subterms marked with either `plus` or `minus` need to match with several subterms of a data term for a single binding of the variable containing the pattern restriction. For example, a query term of the form

$$var\ X \rightsquigarrow f\{\{a, \texttt{minus}\ b, c\}\}$$

requires to remove *all b* subterms of f from the bindings of the variable X (e.g. for the data term $f\{a,b,b,c\}$, whereas ground query term simulation would map the `minus` b to only one b in the data term.

The filtering described above is very basic. More sophisticated forms are conceivable. For example, there might be applications in which it is desirable to exclude all b subterms that have two children that are equal. A possible pattern restriction could be

$$var\ X \rightsquigarrow f\{\{a, \texttt{minus}\ b\{var\ Y, var\ Y\}, c\}\}$$

. However, as the variable Y can only be bound to a single value within one alternative, this would only exclude b subterms with that property for a certain binding of Y. Some sort of "term quantification" is therefore useful, like in

$$var\ X \rightsquigarrow f\{\{a, \texttt{minusall}\ b\{var\ Y, var\ Y\}, c\}\}$$

9.1.4 Advanced Constraint Solving

The evaluation algorithm described in Chapter 8 uses very simple constraints and a very straightforward constraint solver, which resembles constraint solvers for so-called *finite domains* [51], but works with terms instead of integers. Many more sophisticated constraint solvers for different application areas have been studied in literature, for a collection see e.g. [51].

9.2 Support for Special Theories and Reasoners

Many complex query tasks appear frequently in Web queries. Although many of these tasks can be implemented in Xcerpt, a special purpose construct built into the language is often easier and more natural to use. Also, this implementation is often rather inefficient compared with an optimised implementation in a language that is close to machines. Therefore, it might be desirable to support special theories and associated reasoners in future versions of Xcerpt. This Section describes two concrete applications where this is reasonable: Semantic Web reasoning and time reasoning.

Semantic Web Reasoning

With the rise of the Semantic Web, support for Semantic Web technologies like OWL [118] or RDF [119] is increasingly important. However, querying such data beyond the mere structure (as in Section 5.3) usually requires profound knowledge of the underlying concepts (e.g. *description logics*) and has to take into account different syntactical representations of the same data. Integrating support for such Semantic Web reasoning into Xcerpt would therefore be desirable. For example, it would be possible to connect efficient description logics reasoners like *FaCT* [58] or *RACER* [55] to support reasoning with OWL-DL ontologies[2] instead of using the rather basic and inflexible reasoner described in Section 5.3. Since there is no single standard reasoner for the Semantic Web, and since the Semantic Web is developed at a very rapid pace, it is also desirable to let the user specify the kind of reasoner (and ontology language) to use.

Time Reasoning

As most of the data on the Web is associated with some sort of time and date (e.g. timetables, creation dates of documents, validity periods, etc), being able to query based on time and date is often necessary. Unfortunately, there is no single format for representing time (even within a country, there are usually several representations for the same date and time), and there are different concepts that depend on culture and tradition and are not necessarily aligned with common calendar systems (e.g. "Full Moon", "Easter"). Querying time is therefore often a very complicated task. Being able to transparently query time would therefore be a very convenient property. Such support could be integrated into Xcerpt by e.g. using a time reasoner as described in [29].

[2] OWL-DL is the fragment of OWL that is covered by the description logic $\mathcal{SHIQ}$.

9.3 Meta-Programming and Meta-Querying

In *meta-programming* (or *meta-querying*), programs are considered to be data that can be queried and constructed by other programs. A prominent example of a language that allows meta-programming is LISP, but meta-programming has also been studied for e.g. relational query languages, e.g. in Meta-SQL [43].

9.3.1 Meta-Programming on the Web

On the Web, meta-programming is especially appealing, because it allows to consider programs (Xcerpt, XChange, XQuery, ...) as arbitrary resources. Meta-programming on the Web has several application areas:

Locating Web Services. The Web offers an increasing number of so-called "Web services" or "Web applications". A Web service is a resource offering a certain functionality, and differs from static resources like XML or HTML documents in that its content is usually generated dynamically based on user input, e.g. search engines, online stores, online databases, etc. In this "Web of services", locating Web services that provide a certain functionality is of interest, and first approaches that address this issue are currently investigated, like the *Linköping Semantic Web Butler* [97] or OWL-S [82].

As Web services are often implemented in Web languages like Xcerpt, XQuery, or XSLT, meta-programming with Xcerpt can play an important role in this area: imagine a collection of Web services implemented in Xcerpt. With meta-programming, a user could specify an Xcerpt query that queries all Web services and selects only those that fulfil a certain property, e.g. all Web services that produce RSS news feeds (cf. Section 5.2.1). Of course, certain semantic properties, like the termination of a program, are undecidable and thus not queryable.

This scenario can easily be spun further: since the average user is probably not capable of writing a complex Xcerpt program for querying Web services, it could be useful to use a more natural description of a Web service (like in the *Semantic Web Butler* or in OWL-S) and use an Xcerpt meta-program to create another Xcerpt meta-program that implements the actual query for the Web services. Combining this with built-in ontology reasoning (cf. Section 9.2) would even allow to reason with a "Web service ontology".

Software Development and Maintenance. In software development and maintenance, programmers are often interested in finding parts of a program that match certain properties. For example, a programmer might want to query (and modify!) all Xcerpt rules querying a certain resource that moved to a different Web site. Likewise, it is often useful to have a certain abstract model of a program (like UML or different modelling languages). Meta-programming with Xcerpt would allow to define rules that provide a simplified view on another Xcerpt (or XChange, XQuery) program.

A salient application of meta-programming with Xcerpt in this context would be *visXcerpt* [14, 16, 15]: the visual rendering of an Xcerpt program could be performed by using another Xcerpt program instead of XSLT and CSS as in the current implementation.

Automatic Program Construction. In many Web applications it is desirable to construct Xcerpt programs "on-the-fly" based on certain input data. Consider the following scenario: an online bookstore uses XChange and Xcerpt to process customer orders (as events). A customer orders a certain book, but the online bookstore does not have the book on stock. The order system could then automatically create an XChange rule telling the system that if the book arrives at the book store, it should be sent further to the customer.

Verification and Source-to-Source Transformations. Meta-programming can also be used to implement syntactic verifications like grouping and negation stratification (cf. Chapter 6) as an Xcerpt (meta-) program. Likewise, it might be possible to implement source-to-source transformations of Xcerpt programs (i.e. transforming an Xcerpt program to another Xcerpt program) within Xcerpt, e.g. for the purpose of optimisations, simplifications, or typing and type inference.

9.3.2 Supporting Meta-Programming in Xcerpt

Implementing meta-programming in Xcerpt appears straightforward on a first glance: simply take the XML representation of an Xcerpt program and use it as the resource of a query. However, this approach is not sufficient, because it does not allow to properly distinguish between Xcerpt constructs (e.g. variables) of the currently evaluated program and Xcerpt constructs of the queried program. For example, consider the following query term (in the XML syntax not described in this thesis):

```
<xcerpt:rule xmlns:xcerpt="http://xcerpt.org/1.0/programs" xcerpt:total="no">
  <xcerpt:construct>
    <f>
      <xcerpt:variable name="X"/>
    </f>
  </xcerpt:construct>
</xcerpt:rule>
```

In this query term, it is unclear whether the variable X is part of the evaluated program and thus needs to be bound to the content of the f element in the construct part of the queried rule, or whether the variable X is part of the queried program, in which case the query only matches with rules that contain a construct part with an f element containing a variable named X. In the following, two different approaches to solving this problem are suggested. Both seem worth investigating.

Quoting. Traditionally, this problem is addressed by implicitly (e.g. Prolog) or explicitly (e.g. LISP) quoting data, i.e. telling the system what is considered to be data and what is considered to be program. For example, the query term above could be quoted to say that it should only match rules with a construct part containing the variable X as follows:

```
<xcerpt:quote xmlns:xcerpt="http://xcerpt.org/1.0/programs">
  <xcerpt:rule xcerpt:total="no">
    <xcerpt:construct>
      <f>
        <xcerpt:variable name="X"/>
      </f>
    </xcerpt:construct>
  </xcerpt:rule>
</xcerpt:quote>
```

As a companion to quoting, it would also be necessary to have a construct “unquote” that reverts the effect of a “quote”. This would be necessary to say that the variable X is part of the evaluated program:

```
<xcerpt:quote xmlns:xcerpt="http://xcerpt.org/1.0/programs">
  <xcerpt:rule xcerpt:total="no">
    <xcerpt:construct>
      <f>
        <xcerpt:unquote>
          <xcerpt:variable name="X"/>
        </xcerpt:unquote>
      </f>
    </xcerpt:construct>
  </xcerpt:rule>
</xcerpt:quote>
```

Namespaces. Besides quoting, XML provides another means to address the problem: namespaces. Namespaces can be seen as a sophisticated way of quoting, because they allow to disambiguate elements from different (not restricted to two) resources. The query above could be addressed as follows (note that the namespace prefix definitions are deliberately not URIs):

```
<qp:rule xmlns:qp="queried program" xmlns:ep="evaluated program"
         ep:total="no">
  <qp:construct>
    <f>
      <ep:variable name="X"/>
    </f>
  </qp:construct>
</qp:rule>
```

The main problem with this approach is that, by the current XML specification, both Xcerpt programs are in the *same* namespace (`http://xcerpt.org/1.0/program`), whereas the query above would require them to be different. A possible solution to this problem would be to use the Xcerpt namespace only for the evaluated program and use the resource URI of the queried program as the namespace for the queried program, but other approaches are conceivable and interesting to investigate.

9.4 Distributed and Peer-to-Peer Evaluation

In the algorithms and implementations described in this thesis, Xcerpt programs are evaluated locally on a single system, which requires access to the whole program. In a distributed, open environment like the Web, this kind of evaluation is increasingly replaced by distributed or even peer-to-peer evaluation. Both kinds of evaluation can be advantageous over a local evaluation for several reasons, among others:

- *reduced network usage*: only the queries and their results need to be transferred over the network
- *increased performance*: queries that are evaluated on the Web site containing the data can make better use of the local organisation of the data, like index structures, etc. Also, several queries may be evaluated in parallel.
- *more fine-grained access control*: the Web site containing the data can decide which *parts* of the data to include in the result based on access rights of the requesting Web site instead of either admitting or denying access to the data as a whole.

In the following, distributed and peer-to-peer evaluation of Xcerpt programs are described in more detail:

9.4.1 Distributed Evaluation

In a distributed evaluation, parts of the program are sent to a remote Web site for evaluation. After evaluation, the remote Web site sends back the result to the requesting site for further processing. It is conceivable to distribute query terms, queries, or even rules in this manner. For query terms, the remote Web site only needs to implement the *simulation unification* algorithm, and sends back to the requesting site a set of substitutions. For queries or even rules, the remote Web site needs to implement parts of the backward chaining algorithm as well. The first approach has the advantage that simulation unification always terminates and Web site administrators do not need to worry about badly written queries. The second approach has the advantage that it allows to better distribute programs to the queried resources.

Distributed evaluation can be transparent to the programmer: the evaluation engine can automatically verify whether the remote site specified in the input resource of a query is capable of evaluating query terms or rules and then decide to merely send the query and wait for the result instead of retrieving the resource and performing a local evaluation.

9.4.2 Peer-to-Peer Evaluation

Peer-to-peer evaluation is a special kind of decentralised, distributed evaluation in which the participating sites (peers) are not known in advance, as peers connect and disconnect from the network at any time. In general, it is unknown which peers have the requested data, so queries are usually sent further by each peer to all or some of its neighbours in order to reach a large part of the network.

Peer-to-peer networks are usually distinguished by their degree of openness and structure. Openness means that any peer can connect and disconnect at any time. Structure means the amount of information that is available about the data contained in the network. For example, early peer-to-peer networks like *Napster* used a central database for storing information about the offered content of all peers. As this kind of network is considered to be vulnerable, more recent peer-to-peer networks are more decentralised: each peer only has knowledge about itself and about its immediate peers.

For Xcerpt, a peer-to-peer evaluation would thus in particular mean that queries are not evaluated with respect to a specific input resource but instead sent to a peer-to-peer network to request answers from those peers that can give them. Again, it is conceivable to distribute only query terms, queries, or rules. Also, different kinds of peer-to-peer networks with varying degrees of openness and structure can be used.

As peer-to-peer networks are open, it might be possible that there exists an answer to a query, although none is returned within a certain amount of time. Consequently, it would be interesting to investigate Xcerpt programs with a certain amount of uncertainty.

9.5 Optimised Evaluation and Implementation

In the evaluation algorithm and runtime system described in this thesis, optimised evaluation of Xcerpt was not the primary goal. Optimisations can address various parts of the evaluation algorithm. This section briefly discusses possible optimisations for simulation unification, for rule chaining, and in the constraint solver. Furthermore, a virtual machine implementation is suggested, which is currently worked on.

9.5.1 Identifying Complexity of Language Parts

A first step towards optimisations is to identify the complexity of various parts of the algorithm, in particular of the simulation unification algorithm. Preliminary investigations have shown that the simulation unification problem is NP hard, as the 3-SAT problem can be reduced to it[3]. On the other hand, it is known that rooted graph simulation can be computed in polynomial time [67].

Identifying restrictions of query terms that reduce the complexity of simulation unification therefore appear to be promising, as they can help to improve the implementation for those cases. For example, a possible restriction would be to only consider linear terms, i.e. terms where each variable name occurs at most once. Other restrictions could consider only ordered subterm specifications, in which case the number of possible combinations of subterms is reduced significantly.

9.5.2 Simulation Unification

The simulation unification algorithm is at the very heart of the program evaluation. It is therefore justifiable to investigate even rather complex optimisations. Three kinds of optimisations are proposed here:

Index Structures

A common technique in relational database systems is to use index structures to reduce the complexity for certain kinds of queries. An index structure is a certain kind of abstraction from the data (often in form of a tree structure or hash table) that is comparably small and easy to access, whereas the data itself is rather big and slow to access. Interest in index structures for XML data only started recently, but several promising approaches exist, an overview of which is given in [123]. Most of these approaches only address certain kinds of queries, and it is not yet clear how they can be integrated with simulation unification.

[3] Internal memo of Klaus Schulz

Streamed Evaluation

In a streamed evaluation of queries (usually expressed in XPath), the queried XML data is considered to have infinite or at least indefinite breadth. Such data can e.g. be produced by a newsfeed (cf. the example in Section 5.2.1) or by sensors that monitor certain processes constantly (e.g. weather sensors). Streamed evaluation of XPath queries is e.g. investigated in [80, 79]. Adopting the techniques described in [80, 79] to simulation unification appears to be possible. Although Xcerpt (currently) only considers XML documents of finite and known breadth, investigating a streamed evaluation of simulation unification can still be a useful optimisation, because streamed evaluation allows a one-pass evaluation of simulation unification and thus requires only a constant (plus space for the variables) amount of memory.

Schema Information

A third refinement of the evaluation algorithm could make use of the schema information that might be associated with certain XML documents to reduce the "amount of incompleteness". For example, if a query term $t = f\{\{a,b\}\}$ tries to match with a document whose schema states that `a`'s can only appear after `b`'s, then it is possible to refine the query term t to $t = f[[b,a]]$ and remove the (expensive) unordered term specification. Similar refinements can be performed for `desc`, for `optional`, and for `without`. A salient aspect is to combine this approach with a type inference as proposed in e.g. [124] to optimise unifications even with the heads of rules.

9.5.3 Rule Chaining

Optimising rule chaining has been of major interest in the context of other logic programming languages like Prolog. This section briefly summarises three areas that might be interesting for further investigations.

Clause Indexing

Clause indexing is a means to organise rules in some sort of index structure so as to more efficiently decide which rules of a program are relevant for the evaluation of a certain query, i.e. the heads of which rules might unify with a certain query. Applying clause indexing techniques to Xcerpt should in many cases be rather simple due to the similarities between Xcerpt and Prolog. Clause indexing for Prolog has been studied extensively; an overview over available literature can be found at [69].

Query and Clause Selection Strategies

A salient aspect of optimisation can be the selection strategy for the selection of the next query and/or clause that is evaluated in backward chaining. Since Xcerpt programs do not have a fixed evaluation order for the queries in a rule body (except for negation) or for clauses, Xcerpt provides much more freedom for such optimisations than languages like Prolog. Queries could be associated with a certain cost, and the evaluation algorithm could decide to first evaluate queries with a low cost in the expectation that some of the more expensive computations will not be necessary afterwards. For example, it might be reasonable to first evaluate queries against local resources and delay the evaluation of queries against remote resources as far as possible.

Program Rewriting

Program rewriting takes a program and transforms it to a simpler and/or faster program that preferably yields the same results as the original program. In Xcerpt, program rewriting can be used for several optimisations. For example, rules that obviously never participate in the evaluation can be eliminated, several rules that interact via chaining can be combined to a single rule, or several queries to the same resource can be combined to a single query selecting all of the required data. Many approaches to optimisation by program rewriting have been proposed. A survey is given in [88], Section 3.

Memoing/Tabling

Memoing [122] (also called *tabling*) is a technique that stores ("memos") results of previous computations to avoid unnecessarily repeated evaluations of the same program part. Also, memoing allows to detect cyclic computations in many cases, e.g. in the computation of the simulation unification algorithm for terms with cycles (cf. Section 8.2) or in certain cases of backward chaining (cf. Section 8.3.4). For these reasons, memoing is e.g. implemented in the XSB Prolog system[4]. Investigating memoing in Xcerpt is worthwhile, because redundancy on the Web usually means repeatedly retrieving remote resources, which is usually very time consuming.

9.5.4 Constraint Solver

The constraint solver used by the current implementation is lacking in several aspects. Most importantly, repeatedly creating the disjunctive normal form is very inefficient. An interesting enhancement of the constraint solver would therefore be to not consider the disjunctive normal form of a formula and then each disjunct separately in the constraint solver, but instead the constraint store as a whole. Such a constraint solver would need to be able to work with disjunctions in constraint stores. To the best of our knowledge, such constraint solvers have not been investigated much in literature, as disjunctions are usually implemented in the underlying host language (e.g. Prolog).

Another enhancement could be to support user-defined constraints and constraint solvers, e.g. expressed in the language CHR [50], to allow users to add their own theories to the constraint solver. In this manner, it would e.g. be possible to integrate time reasoning or algebraic theories in Xcerpt.

9.5.5 Virtual Machine

In order to establish Xcerpt as a Web query language that is usable in practice, the design and implementation of a virtual machine for program evaluation can be useful. A virtual machine provides a suitable low-level language into which programs implemented in a higher-level language (like Xcerpt) can be transformed. The advantages of this approach are manifold:

- a compiler for the language is easier to implement, as the low-level language is tailored to implement the high-level language
- the instruction set of the virtual machine is closer to the instruction set of the processor and thus easier to implement on different platforms
- the virtual machine can be used both in a compiler and in an interpreter of the language
- the language of the virtual machine allows for low-level optimisations

The virtual machine can be designed and implemented in two steps: for the simulation unification and for the rule chaining algorithm. A virtual machine for simulation unification is already worked on in a project thesis, and a virtual machine for rule chaining is planned.

9.6 Term Formulas as Integrity Constraints

A salient aspect of term formulas as introduced in Section 7.2 is the posibility to specify integrity constraints for XML or semistructured data by using universally or existentially quantified formulas and implications. One example has already been mentioned earlier:

Example 9.3
An integrity constraint that requires all books in the `bib.xml` document to have at least one author:

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

[4] `http://xsb.sourceforge.net/`

Integrity constraints would also allow to specify conditions that are (currently) not expressible in XML schema languages (like RelaxNG or XML schema). For example, they could be used to require that for every IDREF reference, there exists an element with the corresponding ID:

```
∀ Ref . desc attributes {{ idref { var Ref } }} ⇒
    ∃ Id . desc attributes {{ var Id → id { var Ref } }}
```

Furthermore, it would be possible to specify XML schemata that also depend on the content of the document. The following problem arised in the course of a project thesis that aimed at modelling a publication list in XML. In the publication list, there are many different entries, for example journal articles, articles in proceedings, proceedings, etc. All of these have much in common, but differ in some aspects. Furthermore, since the list needs to be easily extensible by new types of publications, the type (e.g. `journal` or `book`) is represented as the value of an element rather than by different parent elements (one might question this representation, but assume that it is like this). The following fragment could be part of such a publication list (in Xcerpt syntax):

```
publist {
  entry {
    type { "book" },
    title { "Data on the Web" },
    isbn { "1-55860-622-X" },
    ...
  },
  entry {
    type { "journal" },
    title { "Journal of the ACM" },
    issn { "0004-5411" },
    ...
  }
}
```

Obviously, books have an ISBN number, whereas journals have an ISSN number. A schema definition that would take this into account would be difficult and contain many redundancies. Instead, two integrity constraints of the following form could ensure this property:

```
∀ Book . publist {{ var Book → entry {{ type { "book" } }} ⇒
    ∃ ISBN . publist {{ var Book → entry {{ var ISBN → isbn {{ }} }}

∀ Journal . publist {{ var Journal → entry {{ type { "journal" } }} ⇒
    ∃ ISSN . publist {{ var Journal → entry {{ var ISSN → issn {{ }} }}
```

CHAPTER

TEN

Conclusion

Summary

This thesis investigated how logic programming techniques can be applied to querying (Semantic) Web data. For this purpose, a new Web query language called Xcerpt has been introduced, and its usefulness has been shown on many practical examples in both the standard and the Semantic Web. In addition to the language specification, a declarative and operational semantics has been proposed that follows closely the traditional logic programming approach. Soundness and weak completeness of the operational semantics with respect to the declarative semantics has been shown for the case of programs without negation.

Arguably, logic programming techniques are suitable for Web querying: queries based on derivation rules are often more intuitive than those using other modularisation techniques, they allow for a straightforward visualisation (in *visXcerpt*), and the reasoning capabilities bridge the gap between standard Web querying and Semantic Web querying. *However*, the particularities of data representation on the Web demand significant changes to traditional logic and logic programming: data does often not conform to a rigid schema, data might be incomplete or redundant, and many different data items might be grouped in a single document under a common root. To address these requirements, Xcerpt introduced incomplete query specifications for querying such data and grouping constructs for creating such data as integral parts of the language. A salient aspect of this thesis is therefore the development of a suitable unification algorithm that is capable of working with such incomplete query specifications, and an extensible backward chaining algorithm that integrates support for grouping constructs.

Concluding Remarks

Designing a programming or query language is a difficult and time consuming task, and this thesis only serves as the first building stone towards the Web query language Xcerpt. Much remains to be done (some of the possibilities have been sketched in Chapter 9), and the areas that may be addressed in future research are manifold.

Part IV

Appendix

APPENDIX

A

A Prototypical Runtime System

As part of this thesis, a prototypical runtime system for evaluating Xcerpt programs has been implemented. This runtime system (from now on called "the prototype") serves both as a testbed for new features and algorithms, and as a means to implement and test Xcerpt queries. Being a prototype, this implementation lacks some features that are desirable for practical applications (like the negation constructs `not` and `without`) and evaluation speed was not one of the primary goals (although evaluation is reasonably fast in many cases).

The runtime system is implemented in the functional language Haskell which, due to its purely functional approach, is particularly well suited for the purpose of prototypical implementations. Haskell allows to program at a very high level of abstraction and thus to stay close to the more formal definition of the evaluation algorithm(s) in Chapter 8.

The following sections illustrate various aspects of the prototype and its evaluation. Since the complete implementation is rather extensive (approximately 6500 lines of code), this chapter only highlights important aspects while the complete source code is provided in electronic form at `http://www.xcerpt.org`. Most of the code presented here is furthermore simplified over the real implementation for presentation purposes. The descriptions here are thus rather meant as a *guide* to the source code than as a standalone description and in most parts require to have the source code at hand. The documentation in this chapter is structured according to the module structure of the source code. Each section starts with a small illustration of the (sub-)module hierarchy.

The source code of the prototype is copyright of the authors and made available under the GNU General Public License (GPL), a copy of which is contained in the source archive. It uses several packages from third parties, particularly the HaXML and HXML XML parsers, and an implementation of the HTTP protocol. HaXML is available under GNU Library General Public Licence (LGPL), and HXML and HTTP under BSD license. All components are Open Source and may be distributed freely. The code is compiled with the *Glasgow Haskell Compiler* (GHC) and runs on both Unix and Windows systems. Makefiles for `make` on Unix are provided.

A.1 Usage of the Prototype

The Xcerpt prototype consists of two callable Unix or Windows binary programs:

- `xcerpt` (or `xcerpt.exe`) implements the command line interpreter
- `convert` (or `convert.exe`) converts between different Xcerpt syntaxes (i.e. XML and Xcerpt).

`xcerpt` can operate in two modes: either with a program as argument (evaluation mode), or in interactive command mode. The first mode of operation is most frequently used and simply evaluates the given program, which can be read either from a file or from standard input. The second mode of operation serves mainly debugging purposes and allows to test various aspects of the program evaluation (like unification).

A.1.1 Command Line Switches

The program `xcerpt` supports the following command line options. As is common on Unix systems, all options are prefixed by `-` and provided in both a short and a long form:

short option	long option	description
`-V`	`--version`	show version number
`-h, -?`	`--help`	show usage
`-I`	`--interactive`	launch interactive interface
`-c`	`--cgi`	add CGI headers in output
`-g term`	`--goal=term`	evaluate query term against program
`-p FILE`	`--program=FILE`	evaluate program
`-i <format>`	`--in=<format>`	input format
`-o <format>`	`--out=<format>`	output format

The command line switch `-I` starts the prototype with the interactive interface, otherwise, it is started in evaluation mode. The switch `-c` is useful when using Xcerpt programs as CGI[1] scripts that are evaluated on a Web server; it adds appropriate HTTP headers (like `Content-Type:`) to the output that allow browsers to render the result correctly.

As input format (switch `-i`), the prototype supports `xml` (the Xcerpt program is in XML syntax), `xcerpt1` (the Xcerpt program is in the old Xcerpt syntax), and `xcerpt2` (the Xcerpt program is in the new Xcerpt syntax). If no input format is specified, the default value of `xcerpt2` is used. Output formats can be specified only, if the goals of the program do not contain an explicit format specification. The `-o` switch supports the same arguments as `-i`. Other switches are explained in the following Section.

Running an Xcerpt Program

The basic command line syntax for running an Xcerpt program is:

`xcerpt (<program file>)` or `xcerpt -p (<program file>)`

The latter syntax is provided for symmetry with the `-I` switch. In both cases, the file *`<program file>`* is loaded as an Xcerpt program and all goals in it are evaluated.

In combination with these commands, it is possible to use the switches `-c`, `-i`, and `-o` described above. In all cases, the output of the Xcerpt program is written either to the resources specified in the program or to standard output (i.e. the current console) if no explicit output resources are given. The syntax of the output again is either specified in the output resource, or the syntax specified by `-o` is used.

In addition, it is possible to evaluate a query term specified at the command line against the rules of the program. In this case, the prototype is called with

`xcerpt -g <query term> -p <program file>`

Note that the switch `-p` is required, and that the specified query term must be in the syntax specified with `-i`. The query term is evaluated *only* against the rules of the program, not against its goals. This option is useful when developing Xcerpt programs. The output is a set of substitutions, always written to standard output, and in the syntax specified by the switch `-o`.

Xcerpt Programs as Unix Scripts

On Unix systems, it is possible to turn "text files" into executable scripts by providing in the first line a specification of the interpreter to use. In this case, it is sufficient to just call the script itself instead of specifying the complete command for the interpreter on the command line. For example, shell scripts for the standard Unix shell usually look as follows:

[1] *Common Gateway Interface*, a common standard for creating dynamic Web applications

```
#!/bin/sh
echo "Hello World."
```

The first line specifies where to find the executable of the interpreter (in the example above `/bin/sh`), the rest is the content of the script. The whole script is passed to the standard input of the interpreter. The Xcerpt prototype supports this behaviour. An Xcerpt program can look as follows:

```
#!/usr/local/bin/xcerpt
GOAL
  result { all var Book }
FROM
  in {
    resource { "file:bib.xml" },
    bib {{ var Book }}
  }
END
```

Assuming, the Xcerpt program is stored in a file with name `books`, it can be evaluated by just entering the command `books` instead of `xcerpt books` (assuming the permissions are set correctly). This is particularly useful when writing Web applications. In this case, the Web server does not need to be aware of Xcerpt and can simply treat the Xcerpt program as a CGI script.

Interactive Interface

The interactive interface can be started with the command `xcerpt -I`. It provides a *command prompt* indicated by the prefix symbols `?-`. The following commands are available in this interface:

Commands for program management:	
`:load <resource>`	load the program at the specified resource into memory
`:run`	run the loaded programs
`:clear`	remove all loaded programs from memory
Generic commands:	
`:quit`	leave the interactive interface
`:help`	show summary of commands
`:version`	show version information
`:reset`	remove all settings
`:set <key> = <value>`	set the property `<key>` to `<value>`
`:set`	show all options
Debugging commands:	
`:unify <t1> = <t2>`	unify `<t1>` and `<t2>` and return the resulting constraint store
`:parse <resource>`	print the term representation of the specified resource

An example session in this interactive interface (loading and running a program) looks as follows:

```
how may I help you?
?- :load prog.xcerpt
Loading prog.xcerpt ...
?- :run
<results>
```

Note that the interface might behave in unexpected ways due to Haskell's lazy evaluation. For example, the program is not actually loaded before the command `:run` is issued. As it is intended mainly for debugging the prototype, the interactive interface does not provide additional commands. It is, however, easy to add this functionality if desirable.

A.2 Overall Structure of the Source Code

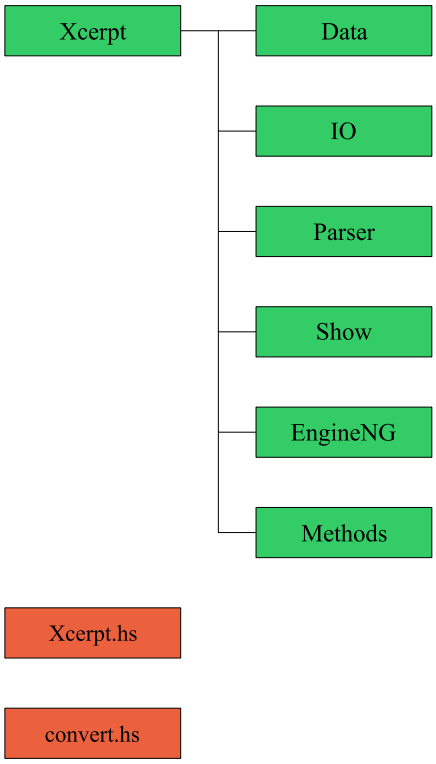

Figure A.1: Overall module and file structure; modules in green, files in red

The source code of the runtime system is structured using Haskell's hierarchical module mechanism. The outline of the structure is shown in Figure A.1. On the top level, there is the module `Xcerpt` containing the actual runtime system and the two files `Xcerpt.hs` and `convert.hs`, which implement the command line interface and the conversion program and both use parts of the module `Xcerpt`. The module `Xcerpt` consists of the following submodules:

Xcerpt.Data contains data structures and helper functions to operate on these structures

Xcerpt.IO contains functions for accessing local and remote resources and accessing them in Haskell

Xcerpt.Parser contains the various parser modules (currently Xcerpt version 1 and 2, XML and HTML) and provides functions for parsing strings into the data structures of `Xcerpt.Data`

Xcerpt.Show contains functions for formatting and pretty-printing the data structures of `Xcerpt.Data`

Xcerpt.EngineNG implements the core part of the runtime system the unification and the constraint-based backward chaining algorithm

Xcerpt.Methods contains the implementations of predefined functions, aggregations and comparisons that are available in Xcerpt programs

In the following, the respective modules are explained in more detail and certain aspects are highlighted to provide proficient programmers the means to modify the prototype as desired to test new features. Most of the code presented here is simplified: the prototype usually contains additional data structures or more complex function definitions that are needed for technical reasons or have been introduced for certain test cases. It is assumed that the reader is already proficient with programming in Haskell, and is familiar with tools like parser and lexer generators (like `yacc` and `lex`).

A.3 Module Xcerpt.Data: Data Structures

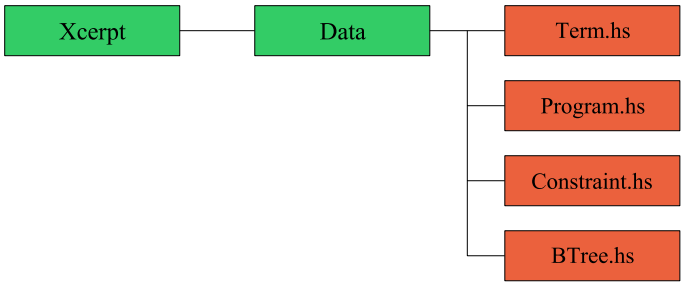

Figure A.2: Module and File Structure of the package Xcerpt.Data; modules in green, files in red

The structure of the module `Xcerpt.Data` is shown in Figure A.2. The module consists of four files:

Term.hs defines a unified data structure for representing data, query, and construct terms, and provides helper functions to perform various tasks on these terms (e.g. find all variables, test whether two terms are equal, ...).

Program.hs contains data structures for programs, rules, resources, and queries

Constraint.hs contains the internal data structures of the constraint solver

BTree.hs contains the definition of generic BTrees used internally in some aspects of program evaluation (see below)

`Term.hs` and `Program.hs` are the files that are most relevant to developers. Their data structures are explained in the following two sections.

A.3.1 Term.hs: Data Structures for Terms

Data Structures

Listing A.1 defines the data structure `Term`, which is used to represent data, construct, and query terms in a unified structure. The code is simplified in that it omits some constructs to improve readability.

Listing A.1: Data Definition of `Term`

```
1  data Term = Elem { label :: Term, namespace :: String,
2                     ordered, total :: Bool, children :: [Term] }
3            | Text String
4            | RegExp { pattern :: String, vars :: [Maybe String] }
5            | Var String
6            | String :→ Term
7            | Desc Term
8            | All [Term]
9            | Some Int [Term]
10           | Reference { identifier :: String, refers  :: Maybe Term }
11           | Anchor    { identifier :: String, content :: Term }
```

Lines 1 and 2 define the most common form of `Term`, i.e. compound terms (e.g. `f[a,b,c]`) that consist of a *label*, a *namespace*, a subterm specification (*ordered* and *total*), and a list of subterms (*children*). A label is of type `Term`, because this allows to represent text labels, variable labels, and regular expression labels in a uniform manner; the types of the other fields are straightforward. In Haskell, field names may be used as functions for retrieving the respective field value. Assuming that a compound term is bound to a variable `t`, the following code retrieves the label of `t`:

`label t`

Lines 3 and 4 define `Terms` for representing text content and regular expressions. The definition of a regular expression consists of a regular expression pattern (in POSIX syntax without Xcerpt extensions) and a list of variables associated with the subexpressions in that pattern. Processing of Xcerpt extensions is performed during parsing.

Lines 5 and 6 define variables and variables with restrictions. A variable is always identified by its name. Variable names in the runtime system are usually different to the original variable names in the Xcerpt program, because variable renaming is performed to avoid conflicts between different rule instances.

Line 7 defines the structure of descendant terms, and lines 8 and 9 define the structure of terms of the form `all t` and `some i t`.[2].

Lines 10 and 11 define referring occurrences and defining occurrences of references. A referring occurrence (constructor `Reference`) consists of a reference name and possibly the referred term (if the reference is already dereferenced). A defining occurrence simply associates an identifier with a term.

Terms can e.g. be created in the following manner:

```
let t = Elem { label = "f", namespace = "http://www.example.com",
               ordered = True, total = True,
               children = [Text a,  Text b, Text c] }
```

Besides the definition shown above, the file `Term.hs` contains definitions for arithmetic expressions and conditions.

Helper Functions

The file `Term.hs` contains two higher order helper functions based on which most other functions are defined. Both take a function as argument and implement a generic recursive traversal over the structure of `Term`, applying the function argument to each subterm.

collectInTerm takes as arguments a term, a transformation function, a merging function, and a default value, and returns a collection of information based on the transformation and merging functions; the transformation function maps subterms to arbitrary values and the merging function merges a list of these values to a single value

recurseTerm takes as arguments a term, a transformation function (transforming one term to another), and returns a transformed term with the same structure

These generic functions are best illustrated on some examples of helper functions that are defined based on them. The following function checks whether a term contains a grouping construct. It uses the function `collectInTerm` and merges the results using `or`:

Listing A.2: Helper function defined using `collectInTerm`

```
1 containsGrouping :: Term → Bool
2 containsGrouping (All _)     = True
3 containsGrouping (Some _ _) = True
4 containsGrouping t           = collectInTerm t containsGrouping or False
```

Lines 2 and 3 define that terms of the form `all t` and `some i t` contain a grouping construct. In a sense, these definitions overwrite the recursive traversal implemented by collectInTerm. Line 4 applies to all other cases and calls `collectInTerm` with `containsGrouping` as transformation function, `or` as merging function, and a default value of `False`. Assuming that the examined term is complex, `collectInTerm` applies the transformation function to all children and merges the list of results with the merging function.

Likewise, the following function uses `recurseTerm` to rename all variables in a term by adding a certain postfix (given as first argument):

[2] both take a list of terms as arguments for future extensions

Listing A.3: Helper function defined using `recurseTerm`

```
1 renameVariables :: String → Term → Term
2 renameVariables p (Var x)  = (Var x++p)
3 renameVariables p (x $\lambda$leadsto$ t) = (x++p) :→ renameVariables p t
4 renameVariables p t        = recurseTerm t (renameVariables p)
```

Lines 2 and 3 add the postfix `p` to variable names, and line 3 in addition applies the function to the pattern restriction. Line 4 implements for all other terms a recursive traversal in which `(renameVariables p)` is applied to all subterms.

One of the main advantage of these two generic functions is that modifications of the data structures (e.g. adding a new kind of terms) usually only need to be reflected in the definition of these two helper functions; all functions that are based on them work without further modification. Whenever changing the data structures, it is therefore important to modify at least these two functions as well.

A.3.2 Program.hs: Data Structures for Programs

Listing A.4: Data structures for programs

```
1 data Program = Prog [Rule] deriving Show
2 data Rule = Rule { rhead :: Term, rbody :: Query }
3           | Goal { output :: [Resource], rhead :: Term, rbody :: Query }
4             deriving Show
```

Consider Listing A.4. Programs are simply represented as lists of rules. A rule is either a goal (line 3) or a standard rule (line 2). Both rules and goals consist of a rule head (a term) and a rule body (a query part – see below); in addition, goals contain a (list of) output resources.

Listing A.5: Data structures for query parts

```
1 data Query = QTerm {resources :: [Resource], term :: Term }
2            | QAnd { resources :: [Resource], queries :: [Query] }
3            | QOr  { resources :: [Resource], queries :: [Query] }
4              deriving (Eq,Show)
```

A query part is either a query term, an `And`-connection of query parts, or an `Or`-connection of query parts.[3] Each query part has a list of associated resources (which might be empty), i.e. the `in` construct of Xcerpt is already resolved during parsing.

Listing A.6: Data structures for resources

```
1 data Resource = XML URI
2               | Xcerpt URI
3               | HTML URI
4               | Parsed Term
5                 deriving (Eq,Show)
```

Resources can be either in XML, Xcerpt, or HTML format (lines 1–3). The respective constructors are used by the parser to determine which parsing module to use. The resource is identified by a URI. Line 4 is used to represent data terms or XML/HTML documents that have already been parsed. The prototype retrieves all resources in a preprocessing step and replaces resource specifications of the first three kinds by a parsed representation. The advantage of this approach is technical: program evaluation does not need to perform I/O and thus avoids the complexity of Haskell's I/O system. Instead, it focusses on the complexity of program evaluation. While this might seem inefficient, Haskell's lazy evaluation guarantees that resources are only actually retrieved when needed. The only drawback is that it anticipates the use of variables in resource specifications.

[3] The file `Program.hs` defines some additional kinds of queries not mentioned here to improve readability.

A.4 Module Xcerpt.IO: Input/Output

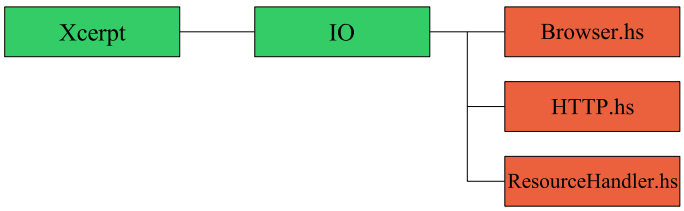

Figure A.3: Module and File Structure of the package Xcerpt.IO; modules in green, files in red

The module `Xcerpt.IO` contains functions for performing input/output operations to local files or over the network. The module contains the following files:

ResourceHandler.hs is the main file of this module; it defines functions for retrieving a resource into a term or string

Browser.hs and HTTP.hs implement access to network resources via the HTTP protocol; they are taken from a library implemented by Warrick Gray[4] and available under the BSD license

The two important functions exported by `ResourceHandler.hs` are the following:

parseResource takes a resource specification as defined above and returns a parsed term structure of the data using the parser for the specified format

writeResource takes a term and writes it to the specified resource. The first argument is a file handle used if the specified resource is standard output (i.e. `stdout:`), in which case the output can be redirected by the system as appropriate.

[4] `http://homepages.paradise.net.nz/warrickg/haskell/http/`

A.5 Module Xcerpt.Parser: Parser

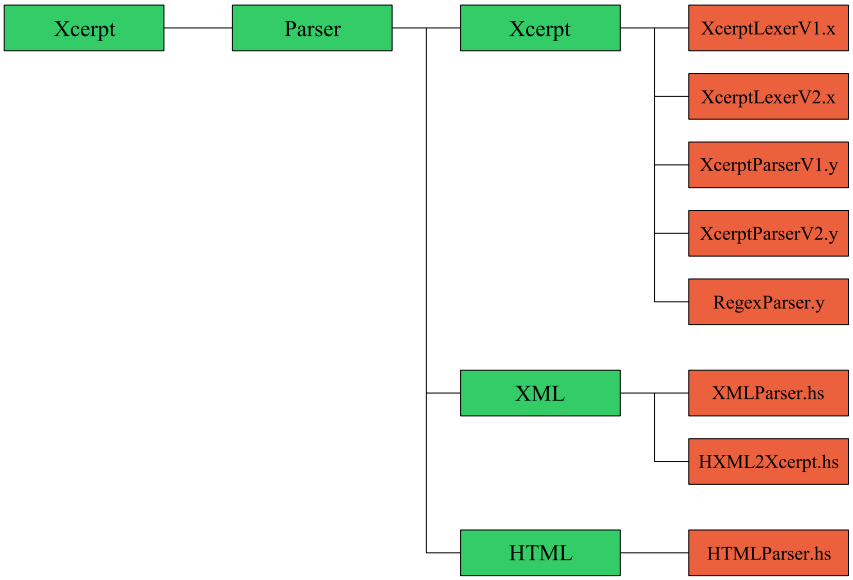

Figure A.4: Module and File Structure of the package Xcerpt.Parser; modules in green, files in red

The Xcerpt parser currently consists of three parsing modules:

Xcerpt.Parser.Xcerpt provides functions for parsing terms and programs in Xcerpt syntax (old and new)

Xcerpt.Parser.XML provides functions for parsing terms and programs in XML syntax (based on HXML[5])

Xcerpt.Parser.HTML provides functions for parsing HTML documents into Xcerpt terms; it differs from the XML parser in that it is somewhat error-resistant and tries to also parse documents that are not well-formed XHTML

All parser modules provide the function `parseTerm` for parsing (data) terms, and the Xcerpt and XML parser in addition provide the function `parseProgram` for parsing programs (in Xcerpt and XML syntax).

A.5.1 Xcerpt.Parser.Xcerpt: Xcerpt V1 and V2 Parser

The Xcerpt parser module consists of two separate parsers: one for the old Xcerpt syntax (V1) primarily used in publications before 2004 (e.g. [23]), and one for the new Xcerpt syntax (V2) used in 2004 and later (and also in this thesis). Both parsers are implemented using the Haskell lexer generator *alex*[6] and the Haskell parser generator *happy*[7].

Lexer Specifications

In *alex*, tokens are defined in terms of regular expressions, similar to other lexer generators. More specific instructions for using *alex* can be found in the *alex* documentation [46]. For example, the following code defines identifiers to begin with an alphabetic character and continue with alphanumeric characters. It returns a token `TIdentifier` which stores the current position in the input file and the value of the character sequence matching the token. The first lines define character classes, and the last two lines define the token `TIdentifier`.

[5] `http://www.flightlab.com/~joe/hxml`
[6] `http://www.haskell.org/alex/`
[7] `http://www.haskell.org/happy/`

```
$idchar     = [A-Z a-z 0-9 \- \_ \:]

tokens :-
  <0> $alpha $idchar*                    { tok (\p s -> TIdentifier p s) }
```

The files `XcerptLexerV1.x` and `XcerptLexerV2.x` contain the respective lexer definitions for the old and new Xcerpt syntax, including definitions for the various available tokens. Both files define a function `lexer` that takes as input a single string and returns as output a list of tokens.

Grammar Specifications

The parser generator *happy* uses LALR(1) grammars that consist of rules in a syntax similar to Backus-Naur Form (BNF), but extended by constructs that allow to define actions for grammar rules. Developers interested in extending or modifying the parser should consult *happy*'s documentation at [74]. For instance, the following code specifies the grammar rule for compound Xcerpt terms with partial and unordered term specification as a label (non-terminal), followed by two opening curly braces (terminals), a list of terms (non-terminal), and two closing curly braces (terminals); it returns a `Term` instance with constructor `Elem` and the field values set appropriately (occurrences of $n refer to the value of the n'th token of the rule). Furthermore, a list of terms (`PTermL`) is defined as either a term (non-terminal), followed by a comma and a list of terms, or a single term, or an empty list of terms; it returns a Haskell list of `Term` elements:

```
PTerm :: { Term }
PTerm : label '{' '{' PTermL '}' '}' { Elem {label=(Text $1), namespace="",
                                             total=False, ordered=False, children=$4}}

PTermL  : PTerm  ',' PTermL          { ($1:$3) }
        | PTerm                      { $1:[] }
        |                            { [] }
```

The files `XcerptParserV1.y` and `XcerptParserV2.y` contain the grammar definitions for parsing Xcerpt terms and programs. In particular, they define the functions `parseTerm` and `parseProgram`, which combine the lexer with the generated parser. Both take as input a single string and return a `Term` resp. a `Program`. In addition, the parser module contains the grammar definition `RegexParser.y`, which defines a grammar for parsing regular expressions with Xcerpt extensions. The regular expression parser is used internally inside the Xcerpt and XML parsers.

A.5.2 Xcerpt.Parser.XML: XML parser

The prototype's XML parser module uses the HXML parser for Haskell, which is very efficient and makes use of Haskell's lazy evaluation. The module consists of two files:

HXMLToXcerpt.hs provides transformation functions that convert XML data from HXML's internal data structures to the prototype's `Term` structure. In particular, these transformation functions take care of attributes and Xcerpt term constructs like term specifications or variables, and resolve namespaces.

XMLParser.hs provides transformation functions that transform a term containing appropriate constructs in the Xcerpt namespace (`http://xcerpt.org`) into a `Program`.

A.5.3 Xcerpt.Parser.HTML: HTML parser

Unfortunately, most of the HTML documents available in today's Web do not conform to the XHTML standard and are therefore not well-formed XML. To make use of existing Web pages, the Xcerpt prototype also contains an HTML parser module. This module uses the Haskell XML parser HaXML [121], which provides an error tolerant HTML parser that parses HTML documents into the same structure as XML documents. The HTML parser consists of the single file `HTMLParser.hs`, which defines a function `parseTerm` to parse HTML documents into a `Term` structure. A function `parseProgram` is not available for HTML, as Xcerpt programs cannot be represented in HTML.

A.6 Module Xcerpt.Show: Output Formatting

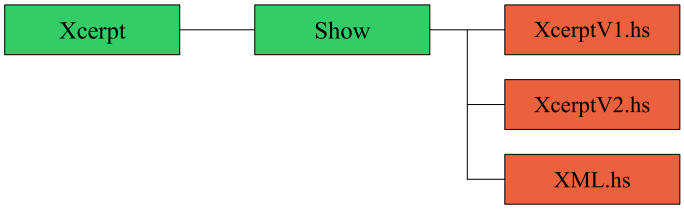

Figure A.5: Module and File Structure of the package Xcerpt.Show; modules in green, files in red

The module `Xcerpt.Show` contains functions for pretty-printing Xcerpt data structures. The main interfaces to these functions are the classes `XcerptPrintable` and `XMLPrintable`, which define pretty-printing in Xcerpt (V1 and V2) and in XML syntax.

Listing A.7: XcerptPrintable

```
1  class XcerptPrintable a where
2      asXcerpt :: Int → a → String
3
4  showXcerpt :: XcerptPrintable a ⇒ a → String
5  showXcerpt = asXcerpt 0
```

The class `XcerptPrintable` defines a function prototype `asXcerpt` that takes the current level of nesting (`Int`) and the data structure to be printed (`a`) as arguments and returns a `String`. The function `showXcerpt` is a convenient wrapper for the default nesting level of `0`.

Listing A.8: XMLPrintable

```
1  class XMLPrintable a where
2      asXML :: Bool → Int → a → String
3
4  showXML :: XMLPrintable a ⇒ a → String
5  showXML = asXML True 0
```

Likewise, the class `XMLPrintable` defines a function prototype `asXML`. `asXML` takes as arguments a `Bool` indicating whether to add Xcerpt attributes for *ordered/unordered* and *total/partial* term specifications in the resulting XML document, an `Int` for the current level of nesting, and the data structure to be printed (`a`). Again, the function `showXML` is a convenient wrapper for default level of nesting and adding Xcerpt attributes.

Both classes are instantiated for the data structures `Term`, `Rule`, and `Program`. The module is divided into the following files:

XcerptV1.hs contains the definition and implementation of the class `XcerptPrintable` for the old Xcerpt V1 syntax (before 2004)

XcerptV2.hs contains the definition and implementation of the class `XcerptPrintable` for the new Xcerpt V2 syntax (2004 and later)

XML.hs contains the definition and implementation of the class `XMLPrintable` for the XML syntax

A.7 Module Xcerpt.EngineNG: Program Evaluation

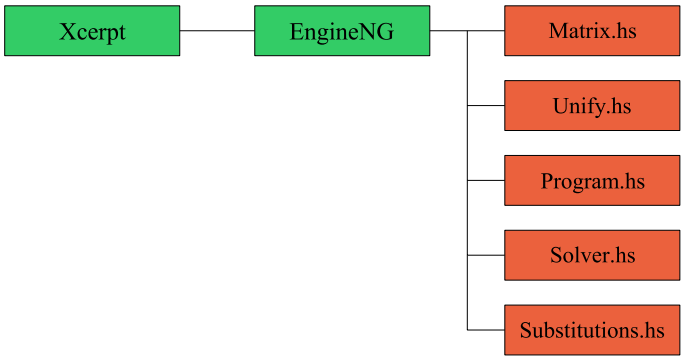

Figure A.6: Module and File Structure of the package Xcerpt.EngineNG; modules in green, files in red

The module `Xcerpt.EngineNG` is the "heart" of the runtime system: it contains the evaluation algorithms described in Chapter 8 and consists of the following parts:

Matrix.hs contains an auxiliary data structure used by the unification algorithm called the *memoisation matrix*; using it, simulation unification can be evaluated in a rather efficient manner.

Unify.hs contains the implementation of the *simulation unification* algorithm described in Section 8.2; it uses the memoisation matrix and the constraint solver described below.

Program.hs contains the implementation of the backward chaining algorithm described in Section 8.3; it uses the unification algorithm and the constraint solver described below.

Solver.hs contains the implementation of a simple and somewhat inefficient but reliable constraint solver

Substitutions.hs implements functions for converting constraint stores into substitutions, and for applying substitutions to terms (cf. Section 7.3)

The following Sections illustrate this implementation in more detail.

A.7.1 Constraint Solver

The constraint solver implemented in the file `Solver.hs` operates on a (conjunctive) list of `Constraints` and yields a list of consistent alternative conjunctions of constraints. It applies simplification rules (or "verification rules") to pairs of constraints. Each application of a verification rule yields a pair of two lists: a list of removed constraints and a list of new constraints.

```
type VerificationRule = (Constraint,Constraint) → ([Constraint], [Constraint])
```

In contrast to traditional constraint solvers, the result of simplification rules in this prototype may also contain disjunctions; in the results of verification rules, these are represented by a constraint of the form `Or` [...], and the incremental solver (in `verifyInc`) generates the disjunctive normal form represented by a list of lists of constraints (i.e. a disjunction of conjunctions of constraints).

The current implementation uses the two verification rules `consistency` and `transitivity` (both also defined in `Solver.hs`), which correspond to the respective rules in Section 8.1.4. The definition of `consistency` is given in Listing A.9:

Listing A.9: Consistency Rule

```
1 consistency :: VerificationRule
2
```

```
3 consistency (c1@(ts1@(Var v,_) :< ts2), c2@(ts1'@(Var v',_) :< ts2') )
4     | (v == v') = ([c1],[solve $ andFl [(unify t2 t2'), (unify t2' t2)] ])
5     where t2    = fst ts2
6           t2'   = fst ts2'
7 consistency _   = ([],[])
```

The definition in line 3 catches the case where the two constraints are of the forms $var\ v \preceq t_2$ and $var\ v' \preceq t_2'$ such that $v = v'$. In this case, one of the constraints is removed (`c1`), and the two upper bounds are unified, i.e. $t_2' \preceq t_2 \wedge t_2 \preceq t_2'$ is added. Line 7 matches all other cases and neither removes nor adds constraints, indicating that `consistency` is not applicable.

The main part of the constraint solver is implemented in the function `verifyInc` (which stands for *incremental verification*). `verifyInc` takes as parameters a list of verification functions (currently only `consistency` and `transitivity`), and two lists of constraints (the current constraint store and the *increment*, i.e. the newly added constraints). The increment must always be a part of the constraint store and the constraint store without the increment is considered to be consistent; in this way, it is sufficient to only consider pairs of constraints where at least one of the constraints is part of the increment. The function `verifyInc` is implemented as follows (note the comments in the source code):

Listing A.10: Constraint Solver

```
1  verifyInc :: [VerificationRule] → [Constraint] → [Constraint] → [[Constraint]]
2
3  verifyInc rules current [] = maybe [] (:[]) $ simplifyPath current
4
5  verifyInc rules current added = concat $ recVerify $ added'
6      where run = flatPair . unzip . filter (≠ ([],[])) $ map (applyRules rules) (pairs (
           id current) (added))
7
8            -- recursively call verifyInc for all conjuncts in the disjunctive
9            -- normal form (see added' below); first parameter to verifyInc is
10           -- the list of verification rules, second is the verified constraint
11           -- store minus the removed constraints and plus the new constraints,
12           -- third is the list of new constraints (increment)
13           recVerify = map (λx → verifyInc rules (old `addList` (new x)) (new x))
14
15           -- the new constraints of x are the constraints of x minus the
16           -- current list of constraints
17           new x = (dupelim x) `minusList` current
18
19           -- the remaining list of constraints is the current list of
20           -- constraints minus the removed constraints
21           old   = current `minusList` removed'
22
23           -- added' is a list of lists of constraints containing the
24           -- disjunctive normal form of all additions (generated by getPaths)
25           added' = simplifyPaths $ getPaths (And $ snd run)
26
27           removed' = fst run
28
29           -- generate all pairs of the elements of two lists. since the first
30           -- list always contains the second list as a tail, and the order of
31           -- the pairs is of no importance, we can drop all elements of the
32           -- second list if it contains the current element.
33           pairs l1 l2 = let l1' = filter isSimConstraint l1
34                             l2' = filter isSimConstraint l2
35                         in [(x,y) | x ← l1',
36                                     y ← (dropIfUntil l2' x l2'), x ≠ y]
```

Since it uses Haskell's function combinators (`$` and `.`), the definition of the function `verifyInc` (line 5) is best read from right to left, and begins with the auxiliary definition of `run` (line 6): `run` implements a complete run over all pairs of constraints from the old constraint store (`current`) and constraints from the increment (`added`). The result is a pair consisting of a list of constraints that need to be removed, and a list of constraints that need to be added in subsequent calls of `verifyInc`. From the result of `run`, the values `added'` (line 25, containing the disjunctive normal form of the new constraints) and `removed'` (line 27, containing a list of constraints to be removed) are extracted. With these lists, the function `recVerify` (line 13) is called, which calls `verifyInc` recursively for each of the conjuncts in `added'`. The recursion terminates upon saturation, i.e. when no new constraints are added (line 3). The application of `concat` to the results of recVerify merges the results of the separate recursive calls into a single list. The result is a list of consistent conjunctions, each representing an alternative solution.

Besides `verifyInc`, the file `Solver.hs` contains a function `simplify` that can be applied to any constraint or constraint store to create a simplified representation without considering dependencies between constraints. In particular, `simplify` eliminates the constraints with boolean values of `True` or `False`. The file `Solver.hs` defines two additional convenience functions used below:

solveCS takes an arbitrary constraint (in general a constraint store), and returns a consistent constraint store in disjunctive normal form, or the boolean constraint `False`.

solveM takes a memoisation matrix containing constraints or sub-matrices (usually created in a unification), and returns a consistent constraint store in disjunctive normal form, or the boolean constraint `False`.

A.7.2 Unification

The file `Unify.hs` contains a prototypical implementation of the Simulation Unification algorithm described in Chapter 8.2. This Section first introduces a naïve implementation, which is straightforward but has a very bad time and space behaviour. As an improvement over this approach, the so-called *memoisation matrix* (defined in the file `Matrix.hs`) is then introduced. Unification with the memoisation matrix is considerably more efficient both with respect to time and space. A further refinement of the memoisation matrix is *matrix compactisation* (a pruning method to exclude parts that never contribute to a valid answer), with which this Section is concluded.

The implementation is described in a very simplified manner; the actual code in the prototype contains many further constructs that improve efficiency or cover some of the more complex constructs, but anticipate a clean presentation. The algorithms are described in a Haskell-like notation, with some syntactic additions that are not available in Haskell but useful for readability. In particular, it uses the Xcerpt term notation instead of the prototype's data structure.

Due to the potentially exponential size of the desired result, time and space complexity are in general exponential. However, an important measure is the number of *unification steps*, i.e. recursive calls of the `unify` function, that are performed. Each such step is computationally expensive, as it requires string comparisons of the labels and recursive calls of `unify` for (in the worst case) all possible combinations of children of the unified subterms. Thus, the number of unification steps is a measure of the number of comparisons that need to be done.

Naïve approach

When unifying two (compound) terms with matching labels, the naïve approach simply builds a disjunction of all alternative combinations of recursive unifications of the subterms and solves each separately (like the declarative description of Simulation Unification in Section 8.2). `unify` is thus a function that takes two terms as arguments and returns a `Constraint` representing the disjunction of combinations of subterm unifications and has the following signature:

```
unify :: Term → Term → Constraint
```

In the following, let `mappings` be the set of functions Π as defined in Definition 8.5 (this list can be created in Haskell in a straightforward manner). The function `unify` for two compound terms $l_1\{\{t_1,\ldots,t_n\}\}$ $l_2\{s_1,\ldots,s_m\}$ can be implemented as follows:

Listing A.11: Naïve Implementation of `unify`

```
1 unify l1{{t1,...,tn}} l2{s1,...,sm} =
2     if l1 ≠ l2 then False
3     else Or [ And (zipWith unify [t1, ... ,tn] [sπ(1), ... ,sπ(n)]})
4                                                 | π ←mappings ]
```

So, in the case of a label mismatch, the result is the atomic constraint *False* (see rule 4 in Chapter 8). In any other cases, for each mapping π in `mappings`, a conjunctive constraint store is created by recursively applying the `unify` function to the list of children $[t_1,\ldots,t_n]$ and their mapping $[s_{\pi(1)},\ldots,s_{\pi(n)}]$.

Example A.1
Consider a unification of the two terms $t_1 = f\{\{var\ X, c\}\}$ and $t_2 = f\{a,b,c,d\}$. Applying the naïve `unify` to t_1 and t_2 yields (in mathematical notation):

$$(unify(var\ X,a) \wedge unify(c,b)) \vee (unify(var\ X,a) \wedge unify(c,c)) \vee (unify(var\ X,a) \wedge unify(c,d)) \vee$$
$$(unify(var\ X,b) \wedge unify(c,a)) \vee (unify(var\ X,b) \wedge unify(c,c)) \vee (unify(var\ X,b) \wedge unify(c,d)) \vee$$
$$(unify(var\ X,c) \wedge unify(c,a)) \vee (unify(var\ X,c) \wedge unify(c,b)) \vee (unify(var\ X,c) \wedge unify(c,d)) \vee$$
$$(unify(var\ X,d) \wedge unify(c,a)) \vee (unify(var\ X,d) \wedge unify(c,b)) \vee (unify(var\ X,d) \wedge unify(c,c))$$

or after evaluating the recursive calls of `unify`:

$$(var\ X \preceq a \wedge False) \vee (var\ X \preceq a \wedge True) \vee (var\ X \preceq a \wedge False) \vee$$
$$(var\ X \preceq b \wedge False) \vee (var\ X \preceq b \wedge True) \vee (var\ X \preceq b \wedge False) \vee$$
$$(var\ X \preceq c \wedge False) \vee (var\ X \preceq c \wedge False) \vee (var\ X \preceq c \wedge False) \vee$$
$$(var\ X \preceq d \wedge False) \vee (var\ X \preceq d \wedge False) \vee (var\ X \preceq d \wedge True) \vee$$

It is easy to observe that this implementation contains many redundancies (e.g. $unify(c,c)$ is computed thrice).

After unification, it is necessary to apply the constraint solver to the resulting constraint store in order to eliminate conjunctions that are inconsistent (either because one of the recursive unification steps fails or because two constraints exclude each other). The constraint solver in `Solver.hs` provides a function `solveCS`, which takes an arbitrary constraint store and creates a consistent constraint store in disjunctive normal form.
Complexity. As there are $\frac{m!}{(m-n)!}$ different total injective mappings from $\{t_1,\ldots,t_n\}$ to $\{s_1,\ldots,s_m\}$, the cardinality of `mappings` is $\frac{m!}{(m-n)!}$. As a consequence, the resulting disjunctive constraint store will contain $\frac{m!}{n!}$ conjunctive subformulas, and require $n \cdot \frac{m!}{(m-n)!}$ unification steps. In particular, many recursive unifications will be performed on the same pairs of subterms, leading to much redundancy.

The Memoisation Matrix

An optimisation over the naïve appriach is to remove redundant unification steps by only performing each unification of pairs of subterms once. The results of these recursive calls are stored in a matrix called the *memoisation matrix* (as it memos the results of unifications for further processing). In this manner, it is possible to reduce the number of necessary unification steps significantly; whereas the naïve approach required $n \cdot \frac{m!}{(m-n)!}$ unification steps, the memoisation matrix requires at most $n \cdot m$ unification steps at one level. Nonetheless, the desired exponential result can be created in later steps by collecting the appropriate unification results in the matrix.

The `unify` function uses the following additional data structure (defined in `Matrix.hs`) to store unification results (the actual implementation in `Matrix.hs` is much more complex, as it allows to use nested matrices and stores additional properties needed for respecting ordered and/or total term specifications):

```
data MMatrix = MMatrix [[Constraint]]
```

The matrix is initialised with all possible combinations of unifications of children from one term with children of the other term.[8] Using the terms $l_1\{\{t_1,\ldots,t_n\}\}$ and $l_2\{s_1,\ldots,s_m\}$ as above, the matrix is thus of size $n \times m$:

Listing A.12: Memoisation Matrix Creation

```
1 initMatrix :: [Term] → [Term] → [[Constraint]]
2 initMatrix [] l = []
3 initMatrix (t:ts) [s1,...,sm] =
4    ((map (unify t) [s1,...,sm]) : initMatrix ts [s1,...,sm])
```

Usage of the matrix is best illustrated on an example:

Example A.2
Consider a unification of the two terms $f\{\{var\ X, c\}\}$ and $f\{a,b,c,d\}$. The matrix for the children is initialised as follows:

$t_1 \backslash t_2$	a	b	c	d
$var\ X$	$unify(var\ X, a)$	$unify(var\ X, b)$	$unify(var\ X, c)$	$unify(var\ X, d)$
c	$unify(c, a)$	$unify(c, b)$	$unify(c, c)$	$unify(c, d)$

Immediate evaluation gives the following matrix:

$t_1 \backslash t_2$	a	b	c	d
$var\ X$	$var\ X \preceq a$	$var\ X \preceq b$	$var\ X \preceq c$	$var\ X \preceq d$
c	$False$	$False$	$True$	$False$

□

Creating the different total mappings of subterms of the one term to subterms of the other term from this matrix is straightforward; informally, each mapping corresponds to a different "path" through the matrix such that a single cell of every row is collected. Note that this method is similar to the "Connection Method" described in [20]. The function `getPaths` serves to create all mappings:

Listing A.13: Path Generation in Memoisation Matrix

```
1 getPaths :: [[Constraint]] → [[Constraint]]
2 getPaths []  = [[]]
3 getPaths [l] = map (λx → [x]) l
4 getPaths (x:xs) = [ (x':xs') | x' ← x, xs' ← (getPaths xs) ]
```

Using Haskell's *list comprehension*, the set of paths is expressed in a very compact manner. However, bear in mind that there are $\frac{m!}{(m-n)!}$ possible paths in the matrix. In case the term specification of the corresponding query term is ordered or total, the function `getPaths` needs to be modified appropriately to only generate monotonic or surjective mappings; this modification is straightforward and not described here.

Example A.3
The following table shows some of the top-down paths through the memoisation matrix of Example A.2. Each path represents a total mapping of subterms of t_1 to subterms of t_2. Note that the leftmost path (in green colour) is not a valid mapping, as it is not injective. The second path beginning with $unify(var\ X, a)$ (in red colour) in contrast even represents a total, injective and *monotonic* mapping and would thus be suitable for ordered matchings. The third path (in blue colour) is not monotonic, but it is injective.

$t_1 \backslash t_2$	a	b	c	d
$var\ X$	$unify(var\ X, a)$	$unify(var\ X, b)$	$unify(var\ X, c)$	$unify(a, d)$
c	$unify(c, a)$	$unify(c, b)$	$unify(c, c)$	$unify(c, d)$

[8] Note that, using Haskell's lazy evaluation, the actual values of the cells are only computed upon use; implementations in other languages should reflect this appropriately

□

The result of a unification is a constraint store in disjunctive normal form, the conjunctions of which each correspond to a top-down path in the memoisation matrix. In the example above, e.g. the red path represents the conjunction $var\ X \preceq a \wedge True$.

Combining the pieces introduced separately above, the function `unify` with memoisation matrix is thus implemented as follows (in simplified form):

Listing A.14: `unify` with Memoisation Matrix

```
1 unify l1{{t1,...,tn}} l2{s1,...,sm} =
2     if l1 ≠ l2 then False
3     else Or . map And . getPaths $ initMatrix [t1, ... ,tn] [s1, ... ,sm]
```

During or after unification, it is necessary to resolve inconsistencies by calling the constraint solver for the resulting matrix. For this purpose, the file `Solver.hs` (described above) provides a function `solveM` that takes a (filled) memoisation matrix as input and solves each of the conjunctive paths in it. As an optimisation, the implementation in the prototype instead solves even while collecting the different paths. To this aim, the function `solveM` reverts to a different implementation for `getPaths` called `getConsistentPaths`, which uses the incremental constraint solver `verifyInc` to only generate paths that are consistent.

Complexity. The overall space and time complexity is still exponential, as the possible size of the desired result is exponential as well. However, the complexity measured in the number of unification steps in this approach is reduced to at most $n \cdot m$, where n, m are the number of nodes in t_1, t_2 respectively. Each node from t_1 is at most unified with each node from t_2, but in many practical cases less – depending on the depth and breadth of the term structure.

Matrix Compactisation

An important observation is that a large part of the fields of the matrix will evaluate to *False* in many applications. Since a path will be translated into a conjunctive constraint store, each path containing at least one *False* is immediately *False* itself. It is thus desireable to not consider such paths at all.

The Xcerpt Prototype uses a *matrix compactisation* such that paths containing *False* will not be considered. This compatisation can be implemented as follows:

Listing A.15: Matrix Compactisation

```
1 compactise :: [[Constraint]] → [[Constraint]]
2 compactise matrix = map (filter (λx → x ≠ False)) matrix
```

Each row of the matrix is compactised such that it no longer contains any *False* constraints. Thus, each path is valid if it is *and*-connected. Obviously, applying the `getPaths` function to a matrix that is compactised in such a way still returns the same (valid) paths as it would have returned without the compactisation. All missing paths are those that would have evaluated to *False* when the *and*-connector would have been applied.

However, using the inexpensive compactisation can reduce the time and space consumption in many practical cases, as a large amount of paths usually contains at least one *False*. This can easily be seen on the example used above.

Example A.4
Since many of the child unifications evaluate to false, the compactised matrix looks as follows:

$t_1 \backslash t_2$				
$var\ X$	$unify(var\ X, a)$	$unify(var\ X, b)$	$unify(var\ X, c)$	$unify(var\ X, d)$
c	$unify(c, c)$			

Obviously, the number of paths is reduced significantly (of the paths in Example A.3, there is only the red path left), while the overall result is not changed. □

Tests conducted using the Haskell implementation as described here have shown an execution time improvement by a factor of 30 in average for the considered data. Note that this compactisation does not allow to determine whether the mapping corresponding to a path is injective and/or monotonic, so generating the correct paths for ordered unification requires to add additional information to the matrix cells. In the current prototype, this is done by adding the subterm positions for the second term.

The complete `unify` function with memoisation matrix and matrix compactisation looks as follows:

Listing A.16: `unify` with matrix compactisation

```
1 unify l1{{t1,...,tn}} l2{s1,...,sm} =
2   if l1 ≠ l2 then False
3   else Or . map And . getPaths . compactise $ initMatrix [t1,...,tn] [s1,...,sm]
```

A.7.3 Backward Chaining

The backward chaining algorithm is implemented in the file `Program.hs`. The main functions exported by this file are `runProgram` (evaluate a program and write resulting terms to resources specified in program or standard output if no resource is given), `hRunProgram` (evaluate a program and write resulting terms to resources specified in program or the handle provided to this function if no resource is given), and `tRunProgram` (evaluate a program and return a list of all resulting terms, disrespecting potential resource specifications). Furthermore, this file provides the functions `evalQuery` and `evalQueryCompat` for evaluating a query part against a program instead of evaluating the goals in the program

The main data structure used by the backward chaining algorithm is a tree (structure `BTree` of module `Xcerpt.Data.BTree`) representing the current constraint store. In this tree, each leaf node represents a conjunct of the disjunctive normal form, but unlike the decomposition trees of Chapter 8, this tree does not convey the history of applications of simplification rules. The sole purpose of this tree is to provide an efficient method for building the DNF by splitting a leaf node into two or more successors if a disjunction needs to be inserted. To operate on this tree, `Program.hs` provides four internal functions `insertAtC` (to insert a constraint in a certain leaf node), `deleteAtC` (to remove a constraint in a certain leaf node), `replaceAtC` (to replace a constraint in a certain leaf node), and `replaceC` (to replace a constraint in all leaf nodes). All functions ensure that a conjunct is consistent by calling the constraint solver described above.

When evaluating a program, the algorithm loops over all conjuncts (function `runC`) in a breadth-first fashion, selects constraints that are not yet fully evaluated (function `selectC`), and applies simplification rules (function `eval`) until no more simplification rules can be applied. For this purpose, `runC` uses a data structure called `EvalContext` as helper (it mainly contains the current program and the current position in the constraint store).

The function `eval` decides, depending on the kind of constraint, how to evaluate the constraint and applies unification of query unfolding if necessary. The results are combined and the tree representing the constraint store is updated. Of particular interest is the treatment of the dependency constraint, which requires to perform an auxiliary computation before the "waiting" constraint can be evaluated. Depending on the result of this auxiliary computation, either the resulting substitutions are applied, or the constraint fails.

Query unfolding and standardisation apart is performed by the function `unfoldQuery`, which takes as an additional argument a prefix used for variable renaming. This prefix is composed depending on the current level of recursion and the position of queries in a conjunction/disjunction such that it is sufficiently unique to avoid conflicts during evaluation. Note that unfolding a query term may yield dependency constraints in case the query term is evaluated against the head of a rule containing a grouping construct.

A.8 Module Xcerpt.Methods: User-Defined Functions

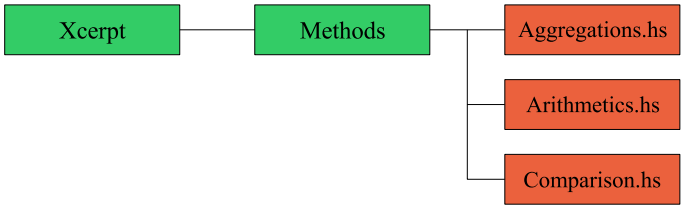

Figure A.7: Module and File Structure of the package Xcerpt.Methods; modules in green, files in red

The module `Xcerpt.Methods` contains the definitions of functions that are available in Xcerpt either as arithmetic/string functions in construct terms (file `Arithmetics.hs`), or as aggregation functions (file `Aggregations.hs`, or as sorting specification in `order by` (file `Comparisons.hs`). So as to not add every function explicitly to the parser, all functions are stored in associated lists in which each entry consists of a pair of string and function definition.

Listing A.17: Definition of Aggregation Functions

```
1  type AggregationFunction = [Term] → [Term]
2
3  aggregations :: [ (String,AggregationFunction) ]
4  aggregations =
5      [ ("count",   ((:[]) . Text . show . length )              ),
6        ("sum",     ((:[]) . Text . show . sum .  map parseFloat) ),
7        ("avg",     ((:[]) . Text . show . avg .  map parseFloat) ),
8        ("min",     ((:[]) . Text . show . min' . map parseFloat) ),
9        ("max",     ((:[]) . Text . show . max' . map parseFloat) ),
10       ("reverse", reverse                                       ),
11       ("first",   take 1                                        ),
12       ("last",    take 1 . reverse                              ),
13       ("rest",    tail                                          ),
14       ("prefix",  reverse . tail . reverse                      )
15     ]
```

Listing A.17 shows the definition of the associated list `aggregations`, which contains the definition of the currently available aggregation functions. Helper functions (like `map'`) are omitted for space reasons. The lists in the files `Arithmetics.hs` and `Comparison.hs` are defined in a similar manner.

Extending the prototype by new user-defined functions can be achieved easily by adding new function definitions to these lists.

APPENDIX

B

Proofs

B.1 Proof of Theorem 4.9 (Reflexivity and Transitivity of $\preceq$)

Theorem 4.9
$\preceq$ is reflexive and transitive.

In order to prove Theorem 4.9, the following Lemma is first shown:

Lemma B.1
Given three sequences of ground query terms L, M, and N, and two (partial or total) mappings $\pi : L \to M$ and $\tau : M \to N$. Furthermore, let $\pi \circ \tau$ denote function composition such that $(\tau \circ \pi)(x) = \tau(\pi(x))$. The following properties hold:

1. if both π and τ are index bijective, so is $\pi \circ \tau$.
2. if both π and τ are index monotonic, so is $\pi \circ \tau$.
3. if both π and τ are position preserving, so is $\pi \circ \tau$.
4. if π is position preserving and τ is position respecting, then $\pi \circ \tau$ is position respecting.
5. if π is position respecting and τ is index monotonic and index bijective, then $\pi \circ \tau$ is position respecting.

Proof.

1. Trivial.
2. Trivial.
3. Let $r \in L$ such that $term(r)$ is a subterm of the form *position* i r'. From the hypothesis, it follows that $s = \pi(r)$ such that $term(s)$ is a subterm of the form *position* k s' with $i = k$. Likewise, it follows that $term(\tau(s))$ is a subterm of the form *position* j t' with $k = j$. Consequently, $i = j$ and $\pi \circ \tau$ is position preserving.
4. Let $r \in L$ such that $term(r)$ is a subterm of the form *position* i r'. From the hypothesis, it follows that $s = \pi(r)$ such that $term(s)$ is a subterm of the form *position* k s' with $i = k$. Likewise, it follows that $\tau(s)$ maps to a subterm with index j such that $k = j$. Thus $i = j$ and $\pi \circ \tau$ is position respecting.
5. Let $r \in L$ such that $term(r)$ is a subterm of the form *position* i r'. From the hypothesis, it follows that $index(\pi(r)) = i$. Since τ is index monotonic and index bijective, it follows that τ maps every s_k to a subterm t_j such that $index(s_k) = index(s_j)$. Thus $i = index(s_j)$ and $\pi \circ \tau$ is position respecting.

□

Proof of Theorem 4.9.

1. *Reflexivity*
   The Lemma is proved by induction over the structure of ground query terms. In all cases, label matching is reflexive. It thus suffices to show that there exists a subterm mapping π with the correct properties. Let t be a ground query term:

- if t is of the form $f[t_1,\ldots,t_n]$, and all of the t_i are reflexive, then the mapping π with $\pi(t_i)=t_i$ for all t_i is trivially index monotonic and index bijective.
- if t is of the form $f\{t_1,\ldots,t_n\}$, and all of the t_i are reflexive, then the mapping π with $\pi(t_i)=t_i$ for all t_i is trivially index bijective; it is furthermore position preserving, as each subterm of the form *position j t* is mapped to the subterm *position j t*.
- if t is of the form $f\{\{t_1,\ldots,t_n\}\}$, and all of the t_i are reflexive, then the mapping π with $\pi(t_i)=t_i$ for all t_i is position respecting (as above), and it furthermore holds that for each $t_i \in SubT^+$ holds $t_i \preceq \pi(t_i)$ (as t_i is reflexive by assumption), and for each $t_i \in SubT^-$ holds $\pi(t_i) \preceq t_i$ (again because t_i is reflexive by assumption).
- if t is of the form $f[[t_1,\ldots,t_n]]$, and all of the t_i are reflexive, then the mapping π with $\pi(t_i)=t_i$ for all t_i is trivially index monotonic and position preserving; the remaining conditions are in analogy to above

All other cases are trivially satisfied.

2. *Transitivity*

It is easy to see that label matching is transitive. Let t_1, t_2, and t_3 be ground query terms such that $t_1 \preceq t_2$ and $t_2 \preceq t_3$. It is to show that $t_1 \preceq t_3$. The lemma is proved by induction over the term structure of t_1, t_2, and t_3. Not all combinations are given; the remaining cases are proved in a similar manner to cases listed here.

- if t_1 is of the form $f_1\{r_1,\ldots,r_l\}$, then t_2 is either of the form (1) $f_2\{s_1,\ldots,s_m\}$ or (2) $f_2[s_1,\ldots,s_m]$ because $t_1 \preceq t_2$, and there exists a mapping π from the successors of t_1 to the successors of t_2 that is index bijective and position preserving.

  Case (1): t_3 is of the forms (1.1) $f_3\{u_1,\ldots,u_n\}$ or (1.2) $f_3[u_1,\ldots,u_n]$

  Case (1.1): there exists a mapping τ from the successors of t_2 to the successors of t_3 that is index bijective and position preserving; with Lemma B.1, it follows that $\pi\circ\tau$ is index bijective and position preserving, and with the induction hypothesis it follows that $t_1 \preceq t_3$.

  Case (1.2): there exists a mapping τ from the successors of t_2 to the successors of t_3 that is index bijective and position respecting; with Lemma B.1, it follows that $\pi\circ\tau$ is index bijective and position respecting, and with the induction hypothesis it follows that $t_1 \preceq t_3$.

  Case (2): t_3 is of the form $f_3[u_1,\ldots,u_n]$, and there exists a mapping τ that is index monotonic and index bijective; with Lemma B.1, it follows that $\pi\circ\tau$ is index bijective and position respecting, and with the induction hypothesis it follows that $t_1 \preceq t_3$

- if t_1 is of the form $f_1\{\{r_1,\ldots,r_l\}\}$, then t_2 is either of the form (1) $f_2\{s_1,\ldots,s_m\}$, (2) $f_2\{\{s_1,\ldots,s_m\}\}$, (3) $f_2[[s_1,\ldots,s_m]]$, or (4) $f_2[s_1,\ldots,s_m]$

  Case (1): there exists a mapping π of $SubT^+(t_1)$ to $SubT(t_2)$ that is position preserving and not completable to subterms of the form *without s*; furthermore, t_3 is of the forms (1.1) $f_3\{u_1,\ldots,u_n\}$ or (1.2) $f_3[u_1,\ldots,u_n]$

  Case (1.1): there exists a mapping τ from the successors of t_2 to the successors of t_3 that is index bijective and position preserving; with Lemma B.1, it follows that $\pi\circ\tau$ is also position preserving and as τ is index bijective, $\pi\circ\tau$ is not completable; with the induction hypothesis it follows that $t_1 \preceq t_3$

  Case (1.2): there exists a mapping τ from the successors of t_2 to the successors of t_3 that is index bijective and position respecting; with Lemma B.1, it follows that $\pi\circ\tau$ is also position respecting and as τ is index bijective, it follows that $\pi\circ\tau$ is not completable; with the induction hypothesis it follows that $t_1 \preceq t_3$.

  Case (2): there exists a total mapping π of $SubT(t_1)$ to $SubT(t_2)$ that is position preserving such that:

  (a) for all $r \in SubT^+(t_1)$ holds that $r \preceq \pi(r)$

  (b) for all $r \in SubT^-(t_1)$ of the form `without` r' holds that $\pi(r)$ is of the form `without` s' and $s' \preceq r'$

  furthermore, t_3 is of the forms (2.1) $f_3\{u_1,\ldots,u_n\}$, (2.2) $f_3[u_1,\ldots,u_n]$, (2.3) $f_3\{\{u_1,\ldots,u_n\}\}$, or (2.4) $f_3[[u_1,\ldots,u_n]]$

  Case (2.1): there exists a mapping τ from the successors of t_2 to the successors of t_3 that is position preserving and not completable; with Lemma B.1, it follows that $\pi\circ\tau$ is also position preserving; with (b), it follows that $\pi\circ\tau$ is also not completable, because a completion to a subterm *without* r' of t_1 would require that there exists a subterm u_i of t_3 such that $r' \preceq u_i$, but no such subterm can exist, because (b) requires that there exists a subterm *without* s' of t_2 such that $s' \preceq r'$ and t_2 is not completable (i.e. *without* s' excludes all subterms for which $r' \preceq u_i$ would hold).

Case (2.2): similar to 2.1 and 1.2

Case (2.3): there exists a mapping τ from the successors of t_2 to the successors of t_3 that is position preserving and for which holds that

- for all $s \in SubT^+(t_2)$ holds that $s \preceq \pi(s)$
- for all $s \in SubT^-(t_2)$ of the form `without` s' holds that $\pi(s)$ is of the form `without` u' and $u' \preceq s'$

with Lemma B.1, it follows that $\pi \circ \tau$ is also position preserving; with (a), (b), and the induction hypothesis follows that

- for all $r \in SubT^+(t_1)$ holds that $r \preceq (\pi \circ \tau)(r)$
- for all $r \in SubT^-(t_1)$ of the form `without` r' holds that $(\pi \circ \tau)(r)$ is of the form `without` u' and $u' \preceq s'$

consequently, $t_1 \preceq t_3$.

Case (2.4): similar to 2.3 and 1.2

Case (3): similar to 2

Case (4): similar to 1

3. if t_1 is of the form *desc* t_1', then either (1) $t_1' \preceq t_2$, or (2) t_2 has a subterm t_2' such that $t_1' \preceq t_2'$, or (3) t_2 is of the form *desc* t_2' such that $t_1' \preceq t_2'$.

   Case (1): as $t_2 \preceq t_3$ holds also $t_1' \preceq t_3$ and thus $t_1 \preceq t_3$.

   Case (2): t_2 has a subterm t_2' such that $t_1' \preceq t_2'$. As $t_2 \preceq t_3$, there exists a subterm t_3' of t_3 such that $t_2' \preceq t_3'$. Thus, $t_1' \preceq t_3'$, and thus $t_1 \preceq t_3$.

   Case (3): trivial

□

B.2 Proof of Theorem 8.6 (Soundness and Completeness of Simulation Unification)

Theorem 8.6 (Soundness of Simulation Unification)
Let t^q be a query term without subterm negation and optional subterms and let t^c be a construct term without grouping constructs, functions/aggregations, and optional subterms. A substitution set Σ is a most general simulation unifier of t^q and t^c if and only if simulation unification of $t^q \preceq_u t^c$ terminates with a constraint store CS such that $\Sigma = \Omega(CS)$.

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

Lemma B.2 (Termination of Simulation Unification)
Let t^q be a query term without subterm negation and optional subterms and let t^c be a construct term without grouping constructs, functions/aggregations, and optional subterms. Simulation unification of $t^q \preceq_u t^c$ terminates.

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

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

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

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

5. application of *deref*

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

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

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

6. application of *memoing*

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

□

Proof of Theorem 8.6.

We prove theorem 8.6 by induction over the number k of applications of decomposition rules to the constraint store C initialised by $C = t^q \preceq_u t^c$. In every case, it is to show that $\Omega(C)$ is the most general simulation unifier of t^q in t^c.

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

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

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

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

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

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

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

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

Induction Step. Assume now that the number of decomposition steps is k. By induction hypothesis, Theorem 8.6 holds for all $i < k$. We have to consider the following cases:

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

   let $t^q = l[t^1_1, \ldots, t^1_m]$ and $t^c = l\{t^2_1, \ldots, t^2_n\}$
   or let $t^q = l[[t^1_1, \ldots, t^1_m]]$ and $t^c = l\{t^2_1, \ldots, t^2_n\}$

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

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

   - let $t^q = l\{\{\ \}\}$ and $t^c = l\{t^2_1, \ldots, t^2_n\}$ or
     let $t^q = l\{\{\ \}\}$ and $t^c = l[t^2_1, \ldots, t^2_n]$ or
     let $t^q = l[[\ ]]$ and $t^c = l[t^2_1, \ldots, t^2_n]$ and $n \geq 1$

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

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

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

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

3. application of *decomp.3* (general decomposition)
   Let $t^q = l\{\{t_1^1,\ldots,t_m^1\}\}$ and let $t^c = l\{t_1^2,\ldots,t_n^2\}$.
   The mgsu of t^q in t^c is the set Σ of all all-grounding substitutions σ such that $\sigma(t^q) \preceq \sigma(t^c)$. According to Definition 4.8, it thus holds that there exists a total, index injective, and position preserving mapping π from $SubT(\sigma(t^q)) = \langle t_1^1,\ldots,t_m^1\rangle$ to $SubT(\sigma(t^c)) = \langle t_1^2,\ldots,t_m^2\rangle$ such that for each $t_i^1 \in SubT(\sigma(t^q))$ holds that $t_i^1 \preceq \sigma(t_i^1)$, and Σ consists of all such σ.
   Application of *decomp.3* to $t^q \preceq_u t^c$ yields $C = \bigvee_{\pi\in\Pi_{pp}} \bigwedge_{1\leq i\leq m} t_i^1 \preceq_u \pi(t_i^1)$. Thus, as by definition, $\Omega(C) = \Omega(\bigvee C') = \bigcup \Omega(C')$, $\Omega(C)$ substitutions for all possible total, index injective, and position preserving functions π. Consider now some $C' = \bigwedge_{1\leq i\leq m} t_i^1 \preceq_u \pi(t_i^1)$ for some mapping π. By definition, we know that $\Omega(C') = \bigcap_{1\leq i\leq n} \Omega(t_i^1 \preceq_u \pi(t_i^1))$, and by induction hypothesis, each $\Omega(t_i^1 \preceq_u \pi(t_i^1))$ is the most general simulation unifier of t_i^1 in $\pi(t_i^1)$. $\Omega(C')$ is thus the maximal all-grounding substitution set that is a simulation unifier for each of the t_i^1 in $\pi(t_i^1)$. Thus, $\Omega(C) = \bigcup \Omega(C')$ is the maximal all-grounding set that is a simulation unifier for any of the mappings π, and as the labels of t^q and t^c match, $\Omega(C)$ is the most general simulation unifier of t^q in t^c.
   The argumentation is identical in the other cases with the exception of the chosen set of functions Π, which is obviously correct.
4. application of *decomp.4* (label mismatch)
   Let t^q and t^c be terms such that the labels mismatch. Hence, ground instances of t^q will not simulate in ground instances of t^c regardless of the substitutions. Thus, the mgsu of t^q in t^c, defined as the union of all simulation unifiers, is empty. *decomp.1* reduces $t^q \preceq_u t^c$ to the constraint store *False*. By definition, $\Omega(False) = \{\ \}$, and thus the theorem is correct.
5. application of *var* ($\rightarrow$ elimination)
   Let $t^q = var\ X \rightarrow t^1$ and let $t^c = t^2$.
   An all-grounding substitution set Σ has to satisfy the following conditions to be a simulation unifier of t^q in t^c:
   (a) Σ must be applicable to $var\ X \rightarrow t^1$, i.e. it may only contain substitutions σ for which holds that $\sigma(t^1) \preceq \sigma(X)$
   (b) it must be a simulation unifier of $var\ X$ in t^2, i.e. for every substitution set σ in Σ holds that $\sigma(X) \preceq \sigma(t^2)$

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

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

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

   $B \cap C$ describes exactly the mgsu of t^q in t^c, because it fulfils the requirements (1) and (2) given above and is maximal, because B and C are maximal.
   As, by induction hypothesis, $t^1 \preceq_u t^2$ computes the mgsu of t^1 in t^2, $A \cap B \cap C = B \cap C$ (i.e. $t^1 \preceq_u t^2$ does not remove further substitutions from $B \cap C$). Note that this corresponds to the fact that $t^1 \preceq_u t^2$ is merely used to improve the evaluation performance.
   Thus, the theorem is correct for this case.

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

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

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

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

   By induction hypothesis, A is the mgsu of $t \preceq_u t^c$, and B is the union of the mgsus of $t^q \preceq_u t_i^2$ for some subterm t_i^2 of t^c. By Definition 4.8, $\Omega(C)$ is thus the maximal set of all-grounding substitutions that is a simulation unifier of t^q in t^c and thus the mgsu.

7. application of *memoing* (termination in case of constraints that have already been treated)
   It suffices to consider the rule *memoing*; the rule *deref* is trivially correct, it simply implements the definition of dereferencing in ground query term graphs (cf. Definition 4.2).

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

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

   - let $t^q = desc\ t$; a substitution set Σ is the mgsu of t^q in t^c, if it contains exactly the substitutions σ for which holds that $\sigma(t^q) \preceq \sigma(t^c)$.
     Evaluation of $C = t^q \preceq_u t^c$ for the first time yields $C = t \preceq_u t^c \vee \bigvee_{1\leq i\leq n} desc\ t \preceq_u t_i^2$ by applying the rule *desc*. Assume that further evaluation of C eventually yields a constraint store (in DNF) of the form $C_1 \vee \cdots \vee C_i \vee \cdots \vee C_m$ for some $m \geq 1$, and that C_i again is of the form $t^q \preceq_u t^c$, because the $desc\ t \preceq_u t_j^2$ leading to C_i contains a cyclic reference to id. Evaluating $t^q \preceq_u t^c$ again then obviously does not yield substitutions that are not already induced by $C_1 \vee \cdots \vee C_{i-1} \vee C_{i+1} \vee \cdots \vee C_m$, and thus replacing C_i by the neutral element for disjunction has no influence on $\Omega(t^q \preceq_u t^c)$. Simulation algorithm is thus sound and complete in this case.
   - let t^q be an arbitrary query term of the form $id'@t$
     Decomposition with any of the rules except *desc* reduces $t^q \preceq_u t^c$ to either an atomic constraint or to a disjunction of conjunctions (in DNF), i.e.

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

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

□

B.3 Proof of Lemma 8.7 (Soundness and Completeness of the Backward Chaining Algorithm)

Lemma 8.7
Let P be a negation-free, grouping stratified Xcerpt program without goals, let M_P be the fixpoint of P, and let Q be a negation-free query (composed of one or more query terms). If the evaluation of $\langle Q \rangle$ terminates with a constraint store CS, then $\Sigma = \Omega(CS)$ is a maximal substitution set with $M_P \models \Sigma(Q)$.

Proof. The proof is done by induction over the number n of unfolding steps (applications of reduction rules) that are performed until a constraint store CS initialised by $CS = \langle Q \rangle$ is completely solved (i.e. no further rules are applicable).
Induction Base. Let first $n = 1$, i.e. there is exactly one query term unfolding step. Then P contains only data terms, and Q is of the form t^q (a query term), i.e. the constraint store is initialised with $CS_0 == \langle t^q \rangle$.

Application of query term unfolding thus reduces CS_0 to

$$CS_1 = t^q \preceq_u t_1^d \vee \cdots \vee t^q \preceq_u t_n^d$$

for all data terms $t_i^d \in P$ $(1 \leq i \leq n)$.

As simulation unification is sound and complete and the t_i^d are already ground, the algorithm computes a constraint store representing substitution sets $\Sigma_1 = \Omega(t^q \preceq_u t_1^d), \ldots, \Sigma_n = \Omega(t^q \preceq_u t_n^d)$ such that $\bigwedge_{\sigma \in \Sigma_i} \sigma(t^q) \preceq t_i^d$ holds for every t_i^d $(1 \leq i \leq n)$ (*). Note that some of the Σ_i may be empty, in which case it is not required that a ground instance of the query term simulates into the data term t_i^d.

By definitions 7.8 and 7.9, M_P contains all data terms t_i^d that are in P. Thus, $M_P \models t_i^d$ for every t_i^d, and with (*) holds that $M_P \models \Sigma(Q)$, where $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n = \Omega(CS_1)$. As P only contains data terms, M_P only consists of the data terms in P. Since t^q is unified with every data term in P, Σ is also a maximal substitution set with $M_P \models \Sigma(Q)$.

Induction Step. Now assume that the number of query term unfolding steps for a constraint store initialised with $CS_0 = \langle Q \rangle$ is n. By the induction hypothesis, Lemma 8.7 holds for all derivations of length $k < n$.

1. Suppose Q is of the form $and\{Q_1, \ldots, Q_m\}$ with $m \geq 2$.
   It suffices to consider the case $m = 2$, i.e. $Q = and\{Q_1, Q_2\}$, as it is always possible to transform Q to an equivalent query of the form $and\{Q_1, and\{Q_2, \ldots, Q_m\}\}$.
   Unfolding $\langle Q \rangle$ to $\langle Q_1 \rangle \wedge \langle Q_2 \rangle$ requires one unfolding step. Hence, solving each of the $\langle Q_i \rangle$ requires k_i query unfolding steps with $k_i < n$. By induction hypothesis, $M_P \models \Sigma_1(Q_1)$ and $M_P \models \Sigma_2(Q_2)$ with $\Sigma_1 = \Omega(\langle Q_1 \rangle)$ and $\Sigma_2 = \Omega(\langle Q_1 \rangle)$, and Σ_1 and Σ_2 are maximal.
   It is to show that

   $$\Sigma = \Omega(\langle Q_1 \rangle \wedge \langle Q_2 \rangle) = \Omega(Q_1) \cap \Omega(Q_2) = \Sigma_1 \cap \Sigma_2$$

   is the maximal substitution set with $M_P \models \Sigma(Q)$.
   We first show that $M_P \models \Sigma(Q)$, i.e. $M_P \models \Sigma(Q_1 \wedge Q_2)$, i.e. $M_P \models \Sigma(Q_1)$ and $M_P \models \Sigma(Q_2)$. For each $\sigma \in \Sigma$ holds that $\sigma \in \Sigma_1$ and $\sigma \in \Sigma_2$, because $\Sigma = \Sigma_1 \cap \Sigma_2$. As $M_P \models \Sigma_1(Q_1)$, it also holds that $M_P \models \Sigma(Q_1)$. Similar for Q_2.
   We then show that Σ is also maximal. Assume that Σ is not maximal. Then there exists a substitution $\sigma \notin \Sigma$ such that $M_P \models \sigma(Q)$, i.e. $M_P \models \sigma(Q_1)$ and $M_P \models \sigma(Q_2)$. As Σ_1 and Σ_2 are maximal for Q_1 and Q_2 by induction hypothesis, $\sigma \in \Sigma_1$ and $\sigma \in \Sigma_2$. Contradiction with $\Sigma = \Sigma_1 \cap \Sigma_2$ and $\sigma \notin \Sigma$ ↯.
2. Suppose Q is of the form $or\{Q_1, \ldots, Q_m\}$ with $m \geq 2$.
   Unfolding $\langle Q \rangle$ to $\langle Q_1 \rangle \vee \langle Q_2 \rangle$ requires one unfolding step. Hence, solving each of the $\langle Q_i \rangle$ requires k_i query unfolding steps with $k_i < n$. By induction hypothesis, $M_P \models \Sigma_i(Q_i)$ with $\Sigma_i = \Omega(solve(\langle Q_i \rangle))$ for each i with $1 \leq i \leq m$, and each of the Σ_i is maximal (*).
   Also, by definition of the solution set $\Omega(\cdot)$, it follows trivially that

   $$\Sigma = \Omega(\langle Q_1 \rangle \vee \cdots \vee \langle Q_m \rangle) = \Omega(\langle Q_1 \rangle) \cup \cdots \cup \Omega(\langle Q_n \rangle_)) = \Sigma_1 \cup \cdots \cup \Sigma_m$$

   With (*), it is easy to see that $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_m$ is a maximal substitution set with $M_P \models \Sigma(Q)$.
3. Suppose Q is of the form t^q, i.e. a query term. Application of the query term unfolding rule to CS_1 yields

   $$CS_1 = \bigvee_{t^c \leftarrow Q' \in \mathcal{P}_{grouping}} (t^q \preceq_u t^c \mid \langle Q' \rangle) \vee \bigvee_{t^c \leftarrow Q' \in \mathcal{P}_{nongrouping}} t^q \preceq_u t^c \wedge \langle Q' \rangle \vee \bigvee_{t^d \in P} t^q \preceq_u t^d$$

   Obviously, none of the $\langle Q' \rangle$ requires more than $n-1$ unfolding steps to solve. Hence, for each rule $t^c \leftarrow Q'$ the algorithm computes a constraint store $C_{Q'}$ such that $\Sigma_{Q'} = \Omega(C_{Q'})$ is by induction hypothesis a maximal set

with $M_P \models \Sigma_{Q'}(Q')$. By Definition 7.8, M_P thus contains $\Sigma_{Q'}(t^c)$ for the construct term t^c corresponding to Q'. Also, M_P does not contain ground instances of t^c beyond those in $\Sigma_{Q'}(t^c)$, because $\Sigma_{Q'}$ is maximal for Q' (*).

Corresponding to the structure of CS_1, we now partition M_P into three (possibly overlapping) sets M_g, M_{ng}, and M_d, where M_g contains the data terms resulting from rules with grouping, M_{ng} contains the data terms resulting from rules without grouping, and M_d contains the data terms occurring in P. To show that the algorithm computes a maximal substitution set Σ with $M_P \models \Sigma(Q)$, it suffices to show that the algorithm computes maximal substitution sets Σ_g for M_g, Σ_{ng} for M_{ng}, and Σ_d for M_d.

(a) Consider $CS'_1 = \bigvee_{t^c \leftarrow Q' \in \mathcal{P}_{grouping}} (t^q \preceq_u t^c \mid \langle Q' \rangle)$. It is to show that the algorithm computes a constraint store $C_g = D_1 \vee \cdots \vee D_n$ from CS'_1 such that $\Sigma_g = \Omega(C_g)$ is a maximal substitution set with $M_g \models \Sigma_g(t^q)$, and each D_i corresponds to the evaluation of a dependency constraint of the form $(t^q \preceq_u t^c \mid \langle Q' \rangle)$ in CS'_1.

Let $t^c \leftarrow Q'$ be one of the rules in $\mathcal{P}_{grouping}$, i.e. with grouping construct. Simplification of the dependency constraint $(t^q \preceq_u t^c \mid \langle Q' \rangle)$ yields $\bigvee_{t^{c'} \in \Sigma_{Q'}(t^c)} t^q \preceq_u t^{c'}$ for the substitution $\Sigma_{Q'} = \Omega(\langle Q' \rangle$ resulting from the evaluation of $\langle Q' \rangle$. As simulation unification is correct, the algorithm computes a substitution set Φ such that each ground instance $t^{q'} \in \Phi(t^q)$ simulates into a ground instance $t^{c'} \in \Sigma'(t^c)$. Thus, $M_g \models \Phi(t^q)$ for every such Φ, and in particular for $\Sigma_g = \bigcup \Phi$ holds that $M_g \models \Sigma_g(t^q)$.

As M_g does not contain data terms not produced by one of the rules in $\mathcal{P}_{grouping}$, Σ_g is also maximal in that respect.

(b) Consider $CS''_1 = \bigvee_{t^c \leftarrow Q' \in \mathcal{P}_{nongrouping}} t^q \preceq_u t^c \wedge \langle Q' \rangle$. It is to show that the algorithm computes a constraint store C_{ng} such that $\Sigma_{ng} = \Omega(C_{ng})$ is a maximal substitution set with $M_{ng} \models \Sigma_{ng}(t^q)$.

Let $t^c \leftarrow Q'$ be one of the rules in $\mathcal{P}_{grouping}$, i.e. without grouping construct. Evaluation of this rule yields $D = t^q \preceq_u t^c \wedge \langle Q' \rangle$. Let C be the constraint store resulting from the unification $t^q \preceq_u t^c$, and let $C_{Q'}$ be the constraint store resulting from the evaluation of $\langle Q' \rangle$. By induction hypothesis, it holds that $C_{Q'}$ induces a substitution set $\Sigma_{Q'} = \Omega(C_{Q'})$ such that $M_P \models \Sigma_{Q'}(Q')$ and $\Sigma_{Q'}$ is maximal. Also, $\Omega(C)$ is the most general simulation unifier of t^q in t^c.

By definition, M_P contains all ground instances of t^c by $\Sigma_{Q'}$, and thus, as t^c does not contain grouping constructs, all ground instances of t^c by $\Sigma_{Q'} \cap \Omega(C)$. As $\Omega(C)$ is a simulation unifier of t^q in t^c, it thus also holds that $\Phi = \Omega(D) = \Omega(C) \cap \Omega(C_{Q'})$ is a substitution set with $M_P \models \Phi(t^q)$, and Φ is maximal, because $\Omega(C)$ is the most general simulation unifier and $\Omega(C_{Q'})$ is maximal by induction hypothesis.

(c) Consider $CS'''_1 = \bigvee_{t^d \in P} t^q \preceq_u t^d$. It is to show that the algorithm computes a constraint store C_d such that $\Sigma_d = \Omega(C_d)$ is a maximal substitution set with $M_d \models \Sigma_d(t^q)$.

Like *Induction Base*.

□

LIST OF EXAMPLES

INDEX

BIBLIOGRAPHY

[1] JTC 1/SC 34. *Standard Generalized Markup Language (SGML)*. International Organization for Standardization (ISO), 1986. ISO 8879:1986.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[3] Serge Abiteboul. Querying Semi-Structured Data. In *Proceedings of ICDT*, pages 1–18, 1997.

[4] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[5] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proceedings of 15th International Conference on Very Large Data Bases (VLDB)*, Amsterdam, 1989.

[6] ANSI. *Database Language SQL (Structured Query Language)*, 1999. ISO/IEC 9075:1999.

[7] Chutiporn Anutariya, Vilas Wuwongse, and Vichit Wattanapailin. An Equivalent-Transformation-Based XML Rule Language. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML02)*, Sardinia, Italy, June 2002. Asian Institute of Technology.

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

[9] Krzysztof Apt and Roland Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[10] N. Bassiliades and I. Vlahavas. Intelligent Querying of Web Documents Using a Deductive XML Repository. In *2nd Hellenic Conference on Artificial Intelligence (SETN-2002)*, Thessaloniki, Greece, April 2002.

[11] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. The Elog Web Extraction Language. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, LNCS 2250, Havana, Cuba, December 2001. Springer-Verlag.

[12] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual Web Information Extraction with Lixto. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, Rome, September 2001.

[13] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.

[14] Sacha Berger. Conception of a Graphical Interface for Querying XML. Master's thesis, Institute for Informatics, University of Munich, 2003. (Diplomarbeit).

[15] Sacha Berger, François Bry, and Sebastian Schaffert. A Visual Language for Web Querying and Reasoning. In *Proceedings of Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR'03)*, LNCS 2901, Mumbai, India, December 2003. Springer-Verlag.

[16] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Proceedings of the International Conference on Very Large Databases (VLDB'03)*, Berlin, Germany, September 2003.

[17] Alexandru Berlea and Helmut Seidl. fxt – A Transformation Language for XML Documents. *Journal of Computing and Information Technology, Special Issue on Domain-Specific Languages*, 2001.

[18] Tim Berners-Lee. *Weaving the Web – The Past, Present and Future of the World Wide Web by its Inventor*. Orion Publishing Group, London, UK, 1999.

[19] Tim Berners-Lee, James Handler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[20] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg Verlag, Wiesbaden, second edition, 1987.

[21] François Bry and Peer Kröger. A Computational Biology Database Digest: Data, Data Analysis, and Data Management. *Distributed and Parallel Databases*, 13(1):7–42, 2002.

[22] François Bry, Dan Olteanu, and Sebastian Schaffert. Grouping constructs for semistructured data. In *Proceedings of WebH2001 at DEXA*, Munich, Germany, September 2001.

[23] François Bry and Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML'02)*, Sardinia, Italy, June 2002. (invited article).

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

[25] François Bry, Sebastian Schaffert, and Andreas Schroeder. A contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Proceedings of the 18th Workshop Logische Programmierung (WLP04)*, LNCS, Potsdam, Germany, 2004. Springer-Verlag.

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

[27] François Bry. XML and Databases. Lecture Notes, 2002. `http://www.pms.ifi.lmu.de/publikationen/#LN_xmldatabases`.

[28] François Bry and Sebastian Schaffert. An Entailment for Reasoning on the Web. In *Proceedings of Workshop on Rules and Rule Markup Languages for the Web (RuleML'03)*, LNCS 2876, Sanibel Island, Florida, USA, 2003. Springer-Verlag.

[29] François Bry and Stephanie Spranger. Towards a Multi-Calendar Temporal Type System for (Semantic) Web Query Languages. In Hans-Jürgen Ohlbach and Sebastian Schaffert, editors, *Proceedings of the International Conference on Logic Programming (ICLP'04)*, LNCS 3208, St. Malo, France, September 2004. Springer-Verlag.

[30] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *DBLP*, 1995.

[31] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[32] Peter Buneman, Vladimir Gapeyev, Haruo Hosoya, Michael Levin, Benjamin Pierce, Jérôme Vouillon, and Philip Wadler. XDuce: A Typed XML Processing Language. Project Website. `http://xduce.sourceforge.net`.

[33] Stefano Ceri, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In *Sistemi Evoluti per Basi di Dati*, pages 151–165. 1999.

[34] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML query use cases. W3C Working Draft 20, December 2001. `http://www.w3.org/TR/xmlquery-use-cases`.

[35] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of *LNCS*. Springer-Verlag, 2000.

[36] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

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

[38] Yi Chen and Peter Revesz. Cxquery: A novel xml query language.

[39] James Clark and Makoto Murata. *RELAX NG Specification*. `http://relaxng.org/spec-20011203.html`, 2001. ISO/IEC 19757-2:2003.

[40] K.L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Database*, pages 293–322. Plenum, New York, 1978.

[41] E. F. Codd. A Database Sublanguage Founded on the Relational Calculus. In E. F. Codd and A. L. Dean, editors, *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971*, pages 35–68. ACM, 1971.

[42] E. F. Codd. Relational Completeness of Data Base Sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[43] Jan Van den Bussche, Stijn Vansummeren, and Gottfried Vossen. Towards practical meta-querying. Technical report, University of Limburg, Belgium, October 2003.

[44] Steven DeRose. Markup Overlap: A Review and a Horse. In *Extreme Markup Languages 2004*, Montréal, Canada, August 2004. IDEAlliance. `http://www.extrememarkup.com/extreme/2004/`.

[45] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, 1998. W3C submission.

[46] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex User Guide*. `http://www.haskell.org/alex/doc/html/alex.html`.

[47] Martin Erwig. A Visual Language for XML. In *IEEE Symp. on Visual Languages*, pages 47–54, 2000.

[48] Richard Fikes, Patrick Hayes, and Ian Horrocks. OWL-QL – A Language for Deductive Query Answering on the Semantic Web. Technical report, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 2003.

[49] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 2nd edition, 2002.

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

[51] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, Heidelberg, 2003.

[52] Norbert Fuhr and Kai Großjohann. XIRQL – An Extension of XQL for Information Retrieval. In *Proceedings ACM SIGIR 2000 Workshop on XML and Information Retrieval*, 2000.

[53] C. F. Goldfarb, E. J. Mosher, and T. I. Peterson. An online system for integrated text processing. In *Proceedings of the American Society for Information Science*, volume 7, 1970.

[54] R. Goldman, S. Chawathe, A. Crespo, and J. McHugh. A standard textual interchange format for the object exchange model (oem). Technical report, Stanford University, October 1996.

[55] Volker Haarslev and Ralf Möller. RACER System Description. In *International Joint Conference on Automated Reasoning (IJCAR'2001)*, Siena, Italy, June 2001.

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

[57] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[58] I. Horrocks. FaCT and iFaCT. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 133–135, Linköping, Sweden, 1999.

[59] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–263, 2000.

[60] IEEE / The Open Group. *POSIX Regular Expressions*, The Open Group Base Specifications Issue 6 edition, 2001. `http://www.opengroup.org/onlinepubs/007904975/basedefs/xbd_chap09.html`.

[61] IETF. *Internationalized Resource Identifiers (IRIs)*, October 2003. Draft 5, `http://www.w3.org/International/iri-edit/draft-duerst-iri-05.txt`.

[62] R. Durbin J. Thierry-Mieg. Syntactic definitions for the ACeDB data base manager. Technical report, MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge, 1992.

[63] Rick Jelliffe. *The Schematron Assertion Language 1.5*. Academia Sinica Computing Centre, 2002. Language Specification, `http://xml.ascc.net/resource/schematron/Schematron2000.html`.

[64] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A Declarative Query Language for RDF. In *The Eleventh International World Wide Web Conference (WWW'02)*, Honolulu, Hawaii, USA, May 2002.

[65] Howard Katz, editor. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2003.

[66] Stephan Kepser. A Proof of the Turing-completeness of XSLT and XQuery. In *Extreme Markup Languages 2004*, Montréal, Canada, August 2004. IDEAlliance. `http://www.extrememarkup.com/extreme/2004/`.

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

[68] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. A Declarative Language for Querying and Restructuring the Web. In *6th IEEE Intl Workshop on Research Issues in Data Engineering – Interoperability of Nontraditional Database Systems (RIDE-NDS)*, pages 12–21, New Orleans, Louisiana, February 1996.

[69] Michael Ley. Prolog clause indexing (overview). website, 2004. `http://www.informatik.uni-trier.de/~ley/db/prolog/indexing.html`.

[70] Mengchi Liu. A Logical Foundation for XML. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE '02)*, LNCS 2348, Toronto, Canada, 2002. Springer-Verlag.

[71] J.W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, second, extended edition, 1987.

[72] B. Ludöscher, Y. Papakonstantinou, and P. Velikhov. A Brief Introduction to XMAS. Technical report, Database Group at University of California, San Diego, 1999.

[73] David Maier. Database Desiderata for an XML Query Language. In *Proceedings of QL'98 - The Query Languages Workshop*, 1998. `http://www.w3.org/TandS/QL/QL98/`.

[74] Simon Marlow and Andy Gill. *Happy User Guide*. `http://www.haskell.org/happy/doc/html/happy.html`.

[75] Wolfgang May. *A Logic-Based Approach to XML Data Integration*. PhD thesis, Albert-Ludwigs-Universität Freiburg i.Br., April 2001. Habilitationsschrift.

[76] Holger Meuss. *Logical Tree Matching with Complete Answer Aggregates for Retrieving Structured Documents*. PhD thesis, University of Munich, 2000.

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

[78] Kevin D. Munroe and Yannis Papakonstantinou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In *VDB*, 2000.

[79] Dan Olteanu, Tim Furche, and François Bry. An Efficient Single-Pass Query Evaluator for XML Data Streams. In *Data Streams Track, 19th Annual ACM Symposium on Applied Computing*, Nicosia, Cyprus, March 2004. ACM.

[80] Dan Olteanu, Tim Furche, and François Bry. Evaluating Complex Queries against XML streams with Polynomial Combined Complexity. In *21st Annual British National Conference on Databases*, Edinburgh, United Kingdom, July 2004.

[81] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proceedings of Workshop on XML Data Management (XMLDM)*, http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-4, 2002. Springer-Verlag LNCS.

[82] The OWL Services Coalition. *OWL-S: Semantic Markup for Web Services*, Dec 2003.

[83] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[84] Yannis Papakonstantinou, Michaelis Petropoulos, and Vasilis Vassalos. QURSED: Querying and Reporting Semistructured Data. In *ACM SIGMOD*, 2002.

[85] Steve Pepper and Graham Moore. *XML Topic Maps (XTM) 1.0*. TopicMaps.Org, 2001. TopicMaps.Org Specification, `http://www.topicmaps.org/xtm/index.html`.

[86] E. Pietriga, V. Quint, and J.-Y. Vion-Dury. VXT: A Visual Approach to XML Transformations. In *ACM Symp. on Document Engineering*, 2001.

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

[88] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[89] Glenn C. Reid, editor. *PostScript Language: Program Design*. Addison-Wesley, 1988. Adobe Systems Inc.

[90] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). In *Proceedings of QL'98 – The Query Languages Workshop*, 1998.

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

[92] Sebastian Schaffert. Grouping constructs for semistructured data. Master's thesis, University of Munich, April 2001.

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

[94] Andreas Schroeder. An Approach to Backward Chaining in Xcerpt (Projektarbeit). Master's thesis, Institute for Informatics, University of Munich, 2004.

[95] Andy Seaborne. RDQL – A Query Language for RDF. W3C Member Submission, January 2004. `http://www.w3.org/Submission/RDQL/`.

[96] Dietmar Seipel. Processing XML Documents in PROLOG. In *Proceedings of the 17th Workshop Logische Programmierung (WLP02)*, Dresden, Germany, December 2002.

[97] Nahid Shahmehri, Juha Takkinen, and Cecile Åberg. Towards Creating Workflows On-the-Fly and Maintaining Them Using the Semantic Web: The sButler Project at Linköping University. In *12th International World Wide Web Conference (WWW 2003)*, 2003.

[98] John C. Shepherdson. Negation in logic programming. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 1, pages 19–88. Morgan Kaufmann, 1988.

[99] Michael Sintek and Stefan Decker. TRIPLE—An RDF Query, Inference, and Transformation Language. In *Proceedings of DDLP'2001*, Japan, October 2001.

[100] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems (Volume I+II)*. Computer Science Press, 1988.

[101] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[102] Eric van der Vliest. *Relax NG*. O'Reilly, 2003.

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

[104] Guido van Rossum et al. *Python*. `http://www.python.org/`.

[105] Sofie Verbaeten, Konstantionos Sagonas, and Danny de Schreye. Termination Proofs for Logic Programs with Tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, January 2001.

[106] W3 Consortium. *Extensible Stylesheet Language Transformations (XSLT)*, November 1999. W3C Recommendation, http://www.w3.org/TR/xslt.

[107] W3 Consortium. *HTML 4.01: The HyperText Markup Language*, 1999. W3C Recommendation, http://www.w3.org/TR/html401/.

[108] W3 Consortium. *XML Path Language (XPath)*, November 1999. http://www.w3.org/TR/xpath.

[109] W3 Consortium. *XHTML 1.0: The Extensible HyperText Markup Language*, 2000. W3C Recommendation, http://www.w3.org/TR/xhtml1/.

[110] W3 Consortium. *XML Linking Language (XLink)*, June 2001. W3C Recommendation, http://www.w3.org/TR/xlink/.

[111] W3 Consortium. *XML Schema Part 0: Primer*, 2001. W3C Recommendation, http://www.w3.org/TR/xmlschema-0/.

[112] W3 Consortium. *XML Schema Part 2: Datatypes*, 2001. W3C Recommendation, http://www.w3.org/TR/xmlschema-2/.

[113] W3 Consortium. *XQuery: A Query Language for XML*, February 2001. W3C Recommendation, http://www.w3.org/TR/xquery/.

[114] W3 Consortium. *XQuery 1.0 and XPath 2.0 Functions and Operators*, November 2003. http://www.w3.org/TR/xpath-functions/.

[115] W3 Consortium. *Cascading Style Sheets, level 2 revision 1*, February 2004. W3C Candidate Recommendation, http://www.w3.org/TR/CSS21/.

[116] W3 Consortium. *Extensible Markup Language (XML) 1.1*, February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xml11-20040204/.

[117] W3 Consortium. *Namespaces in XML 1.1*, February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xml-names11-20040204/.

[118] W3 Consortium. *OWL Web Ontology Language*, February 2004. W3C Recommendation, http://www.w3.org/TR/owl-ref/.

[119] W3 Consortium. *Resource Description Framework*, February 2004. W3C Recommendation, http://www.w3.org/TR/rdf-primer/.

[120] W3 Consortium. *XML Information Set (Second Edition)*, February 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xml-infoset-20040204/.

[121] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proceedings of the International Conference on Functional Programming*, Paris, September 1999. http://www.cs.york.ac.uk/fp/HaXml/icfp99.html.

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

[123] Felix Weigel. Content-Aware DataGuides for Indexing Semi-Structured Data. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2003.

[124] Artur Wilk and Włodzimierz Drabent. On Types for XML Query Language Xcerpt. In *Proceedings of the International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR'03)*, LNCS 2901. Springer Verlag, December 2003.

[125] Andreas Witt. Multiple Hierarchies: New Aspects of an Old Solution. In *Extreme Markup Languages 2004*, Montréal, Canada, August 2004. IDEAlliance. http://www.extrememarkup.com/extreme/2004/.

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

[127] Moshe M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

Curriculum Vitae

Finn Sebastian Schaffert was born on 18th of March, 1976 in Trostberg, Bavaria, Germany, and is of German and Swedish nationality. He attended primary school from 1982-1986 in Siegsdorf and high school from 1986-1995 in Traunstein, where he received his high school degree (Abitur) in 1995. From July 1995 to June 1996 he served in the mandatory military service at the 232nd Alpine Batallion (*Gebirgsjägerbatallion 232*) in Berchtesgaden, Germany.

From 1996 to 2001, he studied Computer Science with a minor of Education Science at the Institute for Informatics, University of Munich (LMU). He finished his studies with honours and his diploma thesis was on *Grouping Constructs for Semistructured Data* (partly published in an article at DEXA'01, [92, 22]), supervised by Prof. Dr. François Bry. During his studies, Sebastian worked in the institute's system administration unit, and performed contract work for various clients including Siemens AG. Also, he is the main developer of a successful Open Source project. From 2001 to 2004, he authored/coauthored 20 articles presented at scientific conferences and gave 3 invited talks.

Since May 2001, Sebastian is working as a research and teaching assistant at the research and teaching unit for *Programming and Modelling Languages* (PMS) at the University of Munich headed by Prof. Dr. François Bry. His research interests include query and programming languages, semistructured data, and the (Semantic) Web.

Sebastian is married and has two daugthers. Besides his professional interests, he likes the mountains and the sea, where he sometimes goes hiking, sailing, and windsurfing. He is also interested in history.