Descriptive Types for XML Query Language
Xcerpt

by

Artur Wilk

Submitted to Linköping Institute of Technology at Linköping University in partial
fulfilment of the requirements for the degree of Licentiate of Engineering

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2006
Descriptive Types for XML Query Language

Xcerpt

by

Artur Wilk

January 2006
ISBN 91-85497-10-X
Linköping Studies in Science and Technology
Thesis No. 1228
ISSN 0280-7971
LiU-Tek-Lic-2006:9

ABSTRACT

The thesis presents a type system for a substantial fragment of XML query language Xcerpt. The system is descriptive; the types associated with Xcerpt constructs are sets
of data terms and approximate the semantics of the constructs. A formalism of Type Definitions, related to XML schema languages, is adopted to specify such sets. The type system is presented as typing rules which provide a basis for type inference and type checking algorithms, used in a prototype implementation. Correctness of the type system wrt. the formal semantics of Xcerpt is proved and exactness of the result types inferred by the system is discussed. The usefulness of the approach is illustrated by example runs of the prototype on Xcerpt programs.

Given a non-recursive Xcerpt program and types of data to be queried, the type system is able to infer a type of results of the program. If additionally a type specification of program results is given, the system is able to prove type correctness of a (possibly recursive) program. Type correctness means that the program produces results of the given type whenever it is applied to data of the given type. Non existence of a correctness proof suggests that the program may be incorrect. Under certain conditions (on the program and on the type specification), the program is actually incorrect whenever the proof attempt fails.

This work has been partially 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). It was also supported by CUGS (The National Graduate School in Computer Science) and SWEBPROD (Semantic Web for Products) by Vinnova.
Acknowledgments

I would like to express my gratitude to my advisor Professor Jan Mahuszyński. His guidance and constant encouragement throughout this work have been invaluable.

I am very grateful to Włodek Drabent for his engagement in this work and for discussions concerning almost all issues of the presented thesis. The work would not be possible without his valuable comments, clarifications and ideas.

I thank Sacha Berger and Emmanuel Coquery for our joint work on some issues of the presented type system and for all stimulating discussions.

This research has been strongly influenced by interaction with the Xcerpt group at the University of Munich. I would like to thank Sebastian Schaffert for his clarifications of several difficult aspects of Xcerpt. I also appreciate interesting discussions with François Bry and Tim Furche.

Furthermore, I would like to thank Ulf Nilsson and the other members of Theoretical Computer Science Laboratory for the creative atmosphere and help I have received.

Final thanks go to my family and friends for believing in me, supporting and encouraging me during this work.

Artur Wilk
Linköping, January 2006

This research has been partially 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). It was also supported by CUGS (The National Graduate School in Computer Science) and SWEBPROD (Semantic Web for Products) by Vinnova.
Contents

1 Introduction ................................................................. 1
  1.1 The Problem and the Motivation .................................. 1
  1.2 The Approach .......................................................... 2
  1.3 The Results .............................................................. 4
  1.4 Thesis Overview ....................................................... 5

2 Background ................................................................. 7
  2.1 Introduction to Xcerpt ............................................... 7
    2.1.1 Data Terms .......................................................... 7
    2.1.2 Query Terms ........................................................ 9
    2.1.3 Queries .............................................................. 11
    2.1.4 Construct Terms .................................................. 12
    2.1.5 Query Rules ........................................................ 12
    2.1.6 Programs ............................................................ 14
  2.2 XML Schema Languages ............................................. 16
    2.2.1 DTD ................................................................. 16
    2.2.2 XML Schema ....................................................... 19
    2.2.3 Relax NG ............................................................ 22

3 Type Specification ....................................................... 25
  3.1 Type Definitions ...................................................... 26
    3.1.1 Proper Type Definitions ....................................... 28
  3.2 Operations on Types ................................................... 29
    3.2.1 Emptiness Check .................................................. 29
    3.2.2 Intersection of Types .......................................... 30
    3.2.3 Type Inclusion .................................................... 33
  3.3 Type Definitions and XML Schema Languages ................... 34
    3.3.1 DTD ................................................................. 35
    3.3.2 XML Schema ....................................................... 36
    3.3.3 Relax NG ............................................................ 41

4 Reasoning about Types .................................................... 45
  4.1 Motivation ............................................................. 45
  4.2 Variable-type Mappings .............................................. 46
4.3 Typing Rules for Xcerpt
  4.3.1 Query terms 47
  4.3.2 Queries 49
  4.3.3 Construct terms 49
  4.3.4 Xcerpt query rules 51
  4.3.5 Xcerpt programs 53
  4.3.6 Exactness of the Typing Rules 54

4.4 Type Inference Algorithm 58
  4.4.1 Type Inference for a Query Rule 58
  4.4.2 Type Inference for a Multiple Query Rule Program 64

4.5 Type Correctness of Programs 65

4.6 Possible Extensions
  4.6.1 Type Correctness of Recursive Programs 66
  4.6.2 The Most General Type 66
  4.6.3 Correctness Checking Based on Types of Variables 66

5 The Prototype 69
  5.1 Usage of the Prototype 69
  5.2 Overall Structure of the Source Code 73

6 Use Cases 77
  6.1 Type Inference 78
    6.1.1 Manual Type Checking 78
    6.1.2 Empty Result Type 79
    6.1.3 Program Documentation 81
  6.2 Result Type Correctness 85

7 Conclusions 89

A Proofs 93
  A.1 Type System Correctness 93
  A.2 Exactness of Inferred Type 93

B Typechecker Printouts 99
Chapter 1

Introduction

1.1 The Problem and the Motivation

The work presented in this thesis is related to XML [24] (Extensible Markup Language) developed by World Wide Web Consortium (W3C)\(^1\). XML has become a dominant standard in the Internet and majority of data being exchanged on the Web is encoded using it. In order to retrieve XML data from the Web, query languages are needed. In most proposals, including the W3C query language XQuery [19], retrieval is based on path navigation. A different approach using pattern matching instead of path navigation is developed in the declarative rule-based query language Xcerpt [10,11,9,4], inspired by logic programming. Further development of Xcerpt is one of the objectives of the Network of Excellence REWERSE\(^2\). This thesis is a contribution to this effort.

In applications using XML data the structure of the data is usually described in schema languages, like DTD [24], XML Schema [25] or RELAX NG [17], in order to validate the data. Xcerpt has no underlying type system nor any provision for taking advantage of the existing schema information. This thesis addresses the problem how such information can be used in Xcerpt. We define a type system where existing structural information is to be used for (1) inferring types of query results, (2) checking type correctness of Xcerpt queries and Xcerpt programs consisting of several rules. In this way we make possible discovery and elimination of type errors in Xcerpt programs. Additionally, such a type system can be used for documentation purposes, providing a result type for programs.

Another application of a type system, which is however not studied in this thesis, is using it for optimization of Xcerpt programs.

Type systems have proved to be very useful in many programming languages for detection of programming errors at compile time. For example, most type systems can check statically that the arguments of primitive arithmetic operations are always numbers (which prevents e.g. adding an integer to a boolean).
The ability to eliminate many errors during early phases of the implementation of an application makes a type system an invaluable tool for checking correctness of programs. On the other hand, experience with untyped programming languages, like Prolog, shows how lack of typing makes many simple errors difficult to discover. B. J. Pierce presents in [35] some other ways in which type systems prove their usefulness in programming process. One of them is enforcement of disciplined programming, in particular in the context of software composition where typing leads to a more abstract style of design. Type information provided by a type system can also be used to improve efficiency of program evaluation. An optimization can be achieved e.g. by eliminating many of the dynamic checks that would be needed without type information. The price we need to pay for the benefits of a type system includes a necessity for a developer to understand the type system in order to work effectively with it. Another thing is an additional effort which must be put to annotate programs with type information. However, the price seems to be worth to pay.

1.2 The Approach

This thesis presents a type system for a substantial fragment of the Web and Semantic Web query language Xcerpt [10, 11, 9, 4]. The considered fragment includes basic and the most important constructions from Xcerpt (Section 2.1 provides a list of neglected Xcerpt features). The type system is still under development and a long range objective is to extend it towards full Xcerpt. We provide a formal semantics of the fragment of Xcerpt we deal with. The semantics (partially presented earlier in [5]) is substantially simpler than that of a full Xcerpt [37] (as it does not use the notion of simulation unification), and may be of separate interest. Similarly to other work related to Xcerpt [38, 37] we use data terms as an abstraction of semi-structured data [2] on the Web. Data terms generalize the notion of term: the number of arguments of a symbol is not fixed, moreover a symbol may have an (unordered) set of arguments, instead of an ordered sequence. In our approach we do not deal with data terms representing graphs which are not trees.

In our approach types are sets of data terms. To specify them we use a formalism of Type Definitions [43, 12]. Type Definitions are similar to unranked tree automata [7] (and equivalent formalisms), but deal also with the case of unordered children of a tree node. More information about other work related to formalisms specifying tree structured data can be found in the introduction to Chapter 3. We introduce a restriction on Type Definitions which allows efficient algorithms for primitive operations on types. The restricted formalism is closed under intersection and makes the type inclusion relation decidable.

Type Definitions define sets (classes) of data terms, thus they play a similar role as schema languages for XML. Type Definitions are not meant to be a next competitive schema language but rather a kind of abstraction of the existing schema languages providing a common view of them. They abstract from the features of schema languages which are not related to defining types (sets) of
1.2. THE APPROACH

XML documents. Thus, we neglect such features of schema languages as an ability to describe default attribute values or to specify processing instructions (notations in DTD). As the formalism of Type Definitions is focused on defining allowable tree structure of XML documents, it leaves out the aspect related to defining specific types of text nodes, like Integer, Date, etc. Thus, the thesis does not discuss the simple types available in XML schema languages. However, we believe that the formalism is flexible enough to be extended with a mechanism handling simple types and we plan to address this aspect in continuation of the presented work.

The type system for Xcerpt is based on a method of computing the type of results for an Xcerpt program, given a type of the database i.e. the type of XML documents to which the program may be applied. The method can be applied to non recursive Xcerpt programs to infer their result type and it can be used to check type correctness of arbitrary programs. We define the type system by means of derivation rules which have been first introduced in [5]. The rules are similar to proof rules of logic, rules used in operational semantics, and those used in prescriptive typing [14]. Employing rules makes it possible to specify a type system in a formal and concise way. Such an approach facilitates formal reasoning. Based on it we present a soundness proof of the type system (an extension of the proof presented [6]) with regard to the declarative semantics of Xcerpt. The rules abstract from lower level details and may be seen as an abstraction of an algorithm for type inference. The algorithm is also presented in the thesis.

The thesis deals with a static type system. Static typing means that type errors are being detected before a program execution e.g. during compilation. This is in contrast to dynamic typing where type errors are being detected at runtime by checking if the actual values are of the required types. The type system is descriptive which means that typing approximates the semantics of a program (in an untyped programming language). In descriptive typing, type inference means computing an approximation of the semantics of the given program; type checking means proving program correctness with respect to a specification expressed by means of types. In our case, for a given Xcerpt program and a type of the database the type system provides a type of the program’s results (i.e. a superset of the set of the program’s results). This is type inference; if a type of expected results is given then type checking can be performed by checking if the obtained type of results is a subset of the the given one. In contrast to descriptive typing, prescriptive typing is related to a typed programming language for which types are important part of its semantics.

The thesis is based on the work presented earlier in [43][16][5][6]. Our approach was inspired by the work [36][22] where the authors present a descriptive type system intended to locate errors in (constraint) logic programs. The main underlying idea was to verify partial correctness of a program with respect to a given type specification describing the intended semantics of the program. To specify types they used regular term grammars.
1.3 The Results

The main contributions of the thesis are:

- A formal semantics of a fragment of Xcerpt. The semantics is substantially simpler than that of full Xcerpt [37] as it does not use the notion of simulation unification. The semantics was introduced earlier in the joint papers [43, 5]; here we have extended it to multiple query rules Xcerpt programs.

- Adaptation of the formalism of Type Definitions (introduced in [12]) to the needs of the presented type system. We have presented efficient algorithms for performing basic operations on types. In particular we have adapted the algorithm of type inclusion from [12]. The algorithms for checking type emptiness and computing type intersection were presented earlier in the joint paper [14]. Here, the algorithm for computing type intersection is extended to handle non-proper Type Definitions.

- A method for inferring an approximation of a type of results of a non-recursive Xcerpt program given a type of a database. The method is presented in two ways:
  - using derivation rules that are abstraction of a type inference algorithm,
  - as a type inference algorithm.

The typing rules were introduced earlier in the joint paper [5]. Here we extend them to deal with multiple query rule Xcerpt programs. We present a correctness proof of the type system which is an extension of the proof presented in [6]. We also provide a discussion on exactness of the result type inferred by the type system.

- A method for checking type correctness of Xcerpt programs given a type of the database and a type of expected results. It can be checked whether the inferred type of results is included in the specified type of expected results. A positive result of such typechecking is a proof of a program correctness with respect to the type specification. On the other hand a negative result can be a hint that the program results may not be of the intended type. Generally, a typechecking failure is not a proof of a type error because the inferred program result type is only an approximation (a superset) of the real result type. However, for some restricted form of Xcerpt programs and a type specification the inferred result type is precise enough for a typechecking failure to be a proof of an unquestionable type error.

The method cannot be directly applied to recursive programs. However, we discuss a way for extending it to such programs.
1.4. THESIS OVERVIEW

- An implementation of type checker for Xcerpt which has been integrated with the Xcerpt runtime system. The type checker prototype is restricted to the fragment of Xcerpt covered by our type system. Moreover, it is restricted to programs consisting only of one query rule.

1.4 Thesis Overview

The remainder of this thesis is organized as follows.

Chapter 2 provides some background knowledge about the query language Xcerpt. It introduces a substantial fragment of Xcerpt and describes its semantics. Additionally, it presents a short introduction to the major XML schema languages: DTD, XML Schema and Relax NG.

Chapter 3 introduces Type Definitions, the formalism for defining types, and provides algorithms for some basic operations on types i.e type intersection, type inclusion etc. Furthermore, it contains a discussion on the relation between Type Definitions and major XML schema languages.

Chapter 4 presents a descriptive type system for Xcerpt. First, it is presented inductively in terms of typing rules based on the syntax of Xcerpt. Then, the chapter provides a description of a practical algorithm for type inference which is based on the typing rules.

Chapter 5 demonstrates the prototype of a type checker, implemented as a part of this thesis. It describes the use of the prototype and its implementation.

Chapter 6 illustrates the use of the type system on examples of Xcerpt programs.

Chapter 7 provides summary of the work presented in this thesis and discusses directions for further studies.

Appendix A provides proofs for theorems and propositions presented in the thesis.

Appendix B contains printouts from the type checker prototype. The printouts are results of typing the Xcerpt programs presented in Chapter.
Chapter 2

Background

2.1 Introduction to Xcerpt

This section introduces Xcerpt, a rule-based query and transformation language for XML. In contrast to other XML query languages Xcerpt employs patterns instead of paths to query XML and semistructured data. This approach is inspired by logic programming. A query term is matched against a data term from a database. A successful matching results in binding the variables in the query term to certain subterms of the data term. This operation is called simulation unification.

We consider here a somehow simplified version of Xcerpt, a fragment containing basic and the most important Xcerpt constructions. The section provides a formal semantics of a fragment of Xcerpt we deal with. The semantics (partially presented earlier in [5]) is substantially simpler than that of a full Xcerpt as it does not use the notion of simulation unification. Another difference is that our data terms represent trees while in full Xcerpt terms are used to represent graphs (by adding unique identifiers to some tree nodes and introducing nodes which are references to these identifiers). Other neglected Xcerpt features in respect to the Xcerpt version described in [38, 37] are: functions and aggregations, non-pattern conditions, optional subterms, position specifications, negation, regular expressions and label variables.

We assume that a database is a data term or a multiset of data terms. There are two other kinds of terms in Xcerpt: query terms and construct terms. The role of query terms is to be matched against a database. Construct terms are used in constructing data terms which are query results.

2.1.1 Data Terms

XML data is modelled using a formalism of data terms similar to that defined in [38]. Data terms can be seen as mixed trees which are labelled trees where children of a node are either linearly ordered or unordered. This is related to existence of two basic concepts in XML: tags which are nodes of an ordered tree
and attributes that attach attribute-value mappings to nodes of a tree. These mappings are represented as unordered trees. Unordered children of a node may also be used to abstract from the order of elements, when this order is inessential. We assume that there is no syntactic difference between XML tag names and attribute names and they both are labels of nodes in our mixed trees (and symbols of our data terms). The infinite alphabet of labels will be denoted by \( \mathcal{L} \).

A content of an element is a sequence of other elements or **basic constants**. Basic constants are basic values such as attribute values and all “free” data appearing in an XML document – all data that is between start and end tag except XML elements, called PCDATA (short for *parseable character data*) in XML jargon. Basic constants occur as strings in XML documents but they can play a role of data of other types depending on an adequate definition in DTD (or other schema languages) e.g. IDREF, CDATA, … The set of basic constants will be denoted by \( \mathcal{B} \). In our notation we will enclose all basic constants in quotation marks “ ”.

**Definition 1.** A **data term** is an expression defined inductively as follows:

- Any basic constant is a data term,
- If \( l \) is a label and \( t_1, \ldots, t_n \) are \( n \geq 0 \) data terms, then \( [t_1, \ldots, t_n] \) and \( \{t_1, \ldots, t_n\} \) are data terms.

The linear ordering of children of the node with label \( l \) is denoted by enclosing them by brackets \([\ ]\), while unordered children are enclosed by braces \( \{\} \).

A **subterm** of a data term \( t \) is defined inductively: \( t \) is a subterm of \( t' \), and any subterm of \( t_i \) (\( 1 \leq i \leq n \)) is a subterm of \( t'[t_1, \ldots, t_n] \) and of \( t'\{t_1, \ldots, t_n\} \). Data terms \( t_1, \ldots, t_n \) will be sometimes called the arguments of \( t' \), or the **direct subterms** of \( t'[t_1, \ldots, t_n] \) (and of \( t'\{t_1, \ldots, t_n\} \)). The **root** of a data term \( t \), denoted \( \text{root}(t) \), is defined as follows. If \( t \) is of the form \( [t_1, \ldots, t_n] \) or \( \{t_1, \ldots, t_n\} \) then \( \text{root}(t) = l \); for \( t \) being a basic constant we assume that \( \text{root}(t) = $ \).

To show how XML elements are represented by data terms, consider an XML element

\[
E = <\text{tag} \; \text{attr}_1=\text{value}_1 \cdots \text{attr}_1=\text{value}_k> E_1 \cdots E_n </\text{tag}>,
\]

\( (k \geq 0, n \geq 0) \) where each \( E_i \) (for \( i = 1, \ldots, n \)) is an element or the text occurring between two elements or before the first element or after the last element. \( E \) is represented as a data term \( \text{tag} [\text{attributes}, \; \text{child}_1, \cdots, \text{child}_n] \), where the data terms \( \text{child}_1, \ldots, \text{child}_n \) represent \( E_1, \ldots, E_n \), and the data term

\[
\text{attributes} = \text{attr}_1 [\text{value}_1], \cdots, \text{attr}_k [\text{value}_k]
\]

represents the attributes of \( E \). Subterms representing attributes are not ordered and this is denoted by enclosing them by braces.
Example 1. This is an XML element and the corresponding data term.

```xml
<CD price="9.90" year="1985">
  Empire Burlesque
  <subtitle></subtitle>
  <artist>Bob Dylan</artist>
  <country>USA</country>
</CD>
```

2.1.2 Query Terms

Query terms are (possibly incomplete) patterns matched against XML data represented by data terms.

Definition 2. Query terms are inductively defined as follows:

- Any basic constant is a query term.
- A variable $X$ is a query term.
- If $q$ is a query term, then desc $q$ is a query term.
- If $X$ is a variable and $q$ is a query term, then $X; q$ is a query term.
- If $l$ is a label and $q_1, \ldots, q_n$ ($n \geq 0$) are query terms, then $l[q_1, \ldots, q_n]$, $l\{q_1, \ldots, q_n\}$, $l[[q_1, \ldots, q_n]]$ and $l\{\{q_1, \ldots, q_n\}\}$ are query terms (called rooted query terms).

For a rooted query term $q = l\alpha q_1, \ldots, q_m\beta$, where $\alpha\beta$ are parentheses $[\cdot\cdot\cdot]$ or $\{\cdot\cdot\cdot\}$ or $\{\cdot\cdot\cdot\}$, root($q$) = $l$ and $q_1, \ldots, q_n$ are the child subterms of $q$. If $q$ is a basic constant then root($q$) = $\$$.

A subterm of a query term is defined in a natural way. In particular, the subterms of $X \sim q$ are $X \sim q$ and all the subterms of $q$.

To informally explain the role of query terms, consider a query term $q = l\alpha q_1, \ldots, q_m\beta$ and a data term $d = l'\alpha' d_1, \ldots, d_m\beta'$, where $\alpha, \beta, \alpha', \beta'$ are parentheses. In order to $q$ match $d$ it is necessary that $l = l'$. Moreover the child subterms $q_1, \ldots, q_m$ of $q$ should match certain child subterms of $d$. Single parentheses in $d$ ($[]$ or $\{\cdot\cdot\cdot\}$) mean that $m = n$ and each $q_i$ should match some (distinct) $d_j$. Double parentheses mean that $m \leq n$ and $q_i, \ldots, q_m$ are matched against some $m$ terms out of $d_1, \ldots, d_n$. Curly braces ($\{\cdot\cdot\cdot\}$ or $\{\cdot\cdot\cdot\}$) in $q$ mean that the order of the child subterms in $d$ does not matter; square brackets in $q$ mean that $q_1, \ldots, q_m$ should match (a subsequence of) $d_1, \ldots, d_n$ in the same order.

A variable matches any data term, desc $q$ matches a data term $d$ whenever $q$ matches some subterm of $d$. A query term $X \sim q$ matches any data term matched by $q$. A side effect of a query term $X$ or $X \sim q$ matching a data term $d$ is that variable $X$ obtains a value $d$.

Now we formally define which query terms match which data terms and what are the resulting assignments of data terms to variables. We do not follow the
original definition of simulation unification. Instead we define a notion of answer substitution for a query term \( q \) and a data term \( d \). As usual, by a substitution (of data terms for variables) we mean a set \( \theta = \{ X_1/d_1, \ldots, X_n/d_n \} \), where \( X_1, \ldots, X_n \) are distinct variables and \( d_1, \ldots, d_n \) are data terms; its domain \( \text{dom}(\theta) \) is \( \{ X_1, \ldots, X_n \} \), its application to a (query) term is defined in a standard way.

**Definition 3 ([43])**. A substitution \( \theta \) is an answer substitution (shortly, an answer) for a query term \( q \) and a data term \( d \) if \( q \) and \( d \) are of one of the forms below and the corresponding condition holds. (In what follows \( m, n \geq 0 \), \( X \) is a variable, \( l \) is a label, \( q, q_1, \ldots \) are query terms, and \( d, d_1, \ldots \) are data terms; set notation is used for multisets, for instance \( \{ d, d \} \) and \( \{ d \} \) are different multisets).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>condition on ( q ) and ( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>( b )</td>
<td>( b ) is a basic constant</td>
</tr>
<tr>
<td>( l[q_1, \ldots, q_n] )</td>
<td>( l[d_1, \ldots, d_n] )</td>
<td>( \theta ) is an answer for ( q_i ) and ( d_i ), for each ( i = 1, \ldots, n )</td>
</tr>
<tr>
<td>( l[q_1, \ldots, q_m] )</td>
<td>( l[d_1, \ldots, d_n] )</td>
<td>for some subsequence ( d_{i_1}, \ldots, d_{i_m} ) of ( d_1, \ldots, d_n ) (i.e. ( 0 &lt; i_1 &lt; \ldots &lt; i_m \leq n )) ( \theta ) is an answer for ( q_j ) and ( d_{i_j} ), for each ( j = 1, \ldots, m ),</td>
</tr>
<tr>
<td>( l{q_1, \ldots, q_n} )</td>
<td>( l{d_1, \ldots, d_n} )</td>
<td>for some permutation ( d_{i_1}, \ldots, d_{i_m} ) of ( d_1, \ldots, d_n ) (i.e. ( { d_{i_1}, \ldots, d_{i_m} } = { d_1, \ldots, d_n } )) ( \theta ) is an answer for ( q_j ) and ( d_{i_j} ), for each ( j = 1, \ldots, m ),</td>
</tr>
<tr>
<td>( l{q_1, \ldots, q_m} )</td>
<td>( l{d_1, \ldots, d_n} )</td>
<td>or for some ( { d_{i_1}, \ldots, d_{i_m} } \subseteq { d_1, \ldots, d_n } ) ( \theta ) is an answer for ( q_j ) and ( d_{i_j} ), for each ( j = 1, \ldots, m ),</td>
</tr>
<tr>
<td>( \text{desc } q )</td>
<td>( d )</td>
<td>( \theta ) is an answer for ( q ) and some subterm ( d' ) of ( d )</td>
</tr>
<tr>
<td>( X )</td>
<td>( d )</td>
<td>( X\theta = d )</td>
</tr>
<tr>
<td>( X \leadsto q )</td>
<td>( d )</td>
<td>( X\theta = d ) and ( \theta ) is an answer for ( q ) and ( d )</td>
</tr>
</tbody>
</table>

We say that \( q \) matches \( d \) if there exists an answer for \( q, d \).

Thus if \( q \) is a rooted query term (or a basic constant) and \( \text{root}(q) \neq \text{root}(d) \) then no answer for \( q, d \) exists. If \( q = d \) then any \( \theta \) is an answer for \( q, d \). A query \( l\{\} \) matches any data term with the label \( l \). If \( \theta, \theta' \) are substitutions and \( \theta \subseteq \theta' \) then if \( \theta \) is an answer for \( q, d \) then \( \theta' \) is an answer for \( q, d \). If a variable \( X \) occurs in a query term \( q \) then queries \( X \leadsto q \) and \( X \leadsto \text{desc } q \) match no data term, provided that \( q \neq X \) and \( q \) is not of the form \( \text{desc } \cdots \text{desc } X \).
2.1. INTRODUCTION TO XCERT

Example 2. Query term \( q_1 = a[c[[d[]",e"]], f[[g[], h["i"]]]] \) matches data terms \( a[c["e", d[], g[]], f[g[], f[[], h["i"]]]] \) and \( a[c[d[], g[], "e"], f[g[], h["i"]]] \). In contrast, data terms \( f[h["i"]], g[] \) and \( f[g[], h["i"]] \) are not matched by \( f[[g[], h["i"]]] \). Query term \( q_2 = \text{desc} w[{}] \) matches data terms \( a[b[w[[]]]] \) and \( w["s"] \). Query term \( q_2 = a[[X_1 \rightarrow c[[d[]]], X_2, "p"]], \) matches \( a["s", c[d[]], "r"] \), \( h[j[]], "p"] \), with an answer which binds \( X_1 \) to \( c[d[]] \), "r" and \( X_2 \) to \( h[j[]] \).

Each answer for a query term \( q \) binds all the variables of the query to some data terms. For any such answer \( \theta' \) (for \( q \) and \( d \)) there exists an answer \( \theta \subseteq \theta' \) (for \( q \) and \( d \)) binding exactly these variables. We will call such answers non redundant. From Definition 3 one can derive an algorithm which produces non redundant answers for a given \( q \) and \( d \). Construction of the algorithm is rather simple, we skip the details. Non redundant answers are actually those of interest; we consider a more general class of answers to simplify Definition 3.

2.1.3 Queries

A query is a connection of zero or more query terms using the connectives \( \text{and} \) and \( \text{or} \). It may furthermore be associated with resources against which the query terms are evaluated.

A targeted query term is a pair \( \text{in}(db, q) \), of a URI and a query term. We assume that the URI locates on the Web a data term \( d(db) \). An answer substitution for \( q \) and \( d(db) \) is called an answer substitution for \( \text{in}(db, q) \) (and any data term).

Definition 4. A query is inductively defined as follows.

- Any query term and any targeted query term is a query.
- If \( Q_1, \ldots, Q_n \) (\( n \geq 0 \)) are queries then \( \text{and}(Q_1, \ldots, Q_n) \) and \( \text{or}(Q_1, \ldots, Q_n) \) are queries.

A substitution \( \theta \) is an answer substitution for \( \text{and}(Q_1, \ldots, Q_n) \) (respectively for \( \text{or}(Q_1, \ldots, Q_n) \)) and a data term \( d \) iff \( \theta \) is an answer substitution for each of (some of) \( Q_1, \ldots, Q_n \) and \( d \).

A subquery is defined in a natural way. In particular a subqueries of \( \text{in}(db, q) \) are \( \text{in}(db, q) \) and all the subterms of \( q \).

A query can be transformed into equivalent one in disjunctive normal form \( \text{or}(Q_1, \ldots, Q_n) \), where each \( Q_i \) is of the form \( \text{and}(Q_{i1}, \ldots, Q_{ik}) \) and each \( Q_{ij} \) is a (targeted) query term (cf. [37] Proposition 6.4]).

Proposition 1. Let \( Q \) be a query, \( d \) a data term and \( \Theta \) a set of answers for \( Q \) and \( d \). If \( Q' \) is a disjunctive normal form of \( Q \) then \( \Theta \) is a set of answers for \( Q' \) and \( d \).

Proof. A sketch. To obtain \( Q' \) we can treat \( Q \) as a propositional formula and transform it iteratively to an equivalent formula. Each such transformation
preserves the set of answers. For instance, the queries and\((Q_1, or(Q_2, Q_3))\) and or\((and(Q_1, Q_2), and(Q_1, Q_3))\) are equivalent formulas, and by Definition 4, they have the same set of answers.

Usually a query is applied to some data term \(d\). However, an answer for a targeted query term \(in(db, q)\) and \(d\) does not depend on \(d\) but only on an external data term specified by \(db\). Thus, for some queries \(Q\) (e.g. consisting only of targeted query terms), we can use a notion of answer for \(Q\) and no data term.

**Definition 5.** Let or\((Q_1, \ldots, Q_n)\) be a disjunctive normal form of \(Q\). A substitution \(\theta\) is an answer substitution for \(Q\) and no data term iff there exists \(Q_i \in \{Q_1, \ldots, Q_n\}\), of the form and\((q_{i1}, \ldots, q_{ik})\) such that each \(q_{i1}, \ldots, q_{ik}\) is a targeted query term and \(\theta\) is an answer substitution for each \(q_{i1}, \ldots, q_{ik}\) (and some data term).

**Proposition 2.** \(\theta\) is an answer substitution for \(Q\) and any data term iff \(\theta\) is an answer substitution for \(Q\) and no data term.

### 2.1.4 Construct Terms

Construct terms are used in constructing data terms which are results of queries.

**Definition 6.** A construct term and the set \(FV(c)\) of free variables of a construct term \(c\) are defined recursively. If \(b\) is a basic constant, \(X\) a variable, \(l\) a label, \(c, c_1, \ldots, c_n\) construct terms \((n \geq 0)\), and \(k\) a natural number then

\[
\begin{align*}
&b, X, l[c_1, \ldots, c_n], l\{c_1, \ldots, c_n\}, \text{ all } c, \text{ some } k c,
\end{align*}
\]

are construct terms. \(FV(b) = \emptyset\), \(FV(X) = \{X\}\), \(FV(l[c_1, \ldots, c_n]) = FV(l\{c_1, \ldots, c_n\}) = \bigcup_{i=1}^{n} FV(c_i)\), \(FV(\text{all } c) = FV(\text{some } k c) = \emptyset\).

Notice that any data term is a construct term. (Also, a construct term without any all and some construct is a query term).

### 2.1.5 Query Rules

Before we define the notion of a query rule and its result we need to provide some auxiliary definitions.

By a substitution set we mean a set of substitutions of data terms for variables, e.g. of answers for a query and a data term.

**Definition 7.** Given a substitution set \(\Theta\) and a set \(V\) of variables, such that \(V \subseteq \text{dom}(\theta)\) for each \(\theta \in \Theta\), the equivalence relation \(\simeq_V \subseteq \Theta \times \Theta\) is defined as: \(\theta_1 \simeq \theta_2\) iff \(\theta_1(X) = \theta_2(X)\) for all \(X \in V\). The set of equivalence classes of \(\simeq_V\) is denoted by \(\Theta/\simeq_V\).

The concatenation of two sequences \(S_1, S_2\) of data terms will be denoted by \(S_1 \circ S_2\). We do not distinguish between a data term \(d\) and the one element sequence with the element \(d\).
2.1. INTRODUCTION TO XSCRIPT

Definition 8. Let $c$ be a construct term and $\Theta$ be a substitution set containing the same assignments for the free variables $FV(c)$ of $c$ (i.e. $\theta_1 \simeq_{FV(c)} \theta_2$ for any $\theta_1, \theta_2 \in \Theta$). The application $\Theta(c)$ of the substitution set $\Theta$ to $c$ is a sequence of data terms defined as follows

- $\Theta(b) = b$, where $b$ is a basic constant.
- $\Theta(X) = X\theta$, where $\theta \in \Theta$.
- $\Theta(l\{c_1, \ldots, c_n\}) = l\{\Theta(c_1) \circ \cdots \circ \Theta(c_n)\}$.
- $\Theta(l[c_1, \ldots, c_n]) = l[\Theta(c_1) \circ \cdots \circ \Theta(c_n)]$.
- $\Theta(\text{all } c') = \Theta_1(c') \circ \cdots \circ \Theta_k(c')$, where $\{\Theta_1, \ldots, \Theta_k\} = \Theta/\simeq_{FV(c')}$.
- $\Theta(\text{some } k c') = \Theta_1(c') \circ \cdots \circ \Theta_m(c')$, where $\{\Theta_1, \ldots, \Theta_m\} \subseteq \Theta/\simeq_{FV(c')}$ and $m = k$ if $|\Theta/\simeq_{FV(c')}| \geq k$ or $m = |\Theta/\simeq_{FV(c')}|$ otherwise.

For a construct term $c$ containing neither all nor some, $\Theta(c) = c\theta$ for any $\theta \in \Theta$. Notice that $\Theta(c)$ is defined uniquely unless $c$ contains all or some (and $\Theta(c)$ is defined uniquely up to reordering provided $c$ does not contain some). Notice also that $\Theta(c)$ is a one element sequence unless $c$ is of the form all $c'$ or some $k c'$.

Definition 9. A construct-query rule (shortly, query rule) is an expression of the form $c \leftarrow Q$, where $c$ is a construct term not of the form all $c'$ or some $k c'$, $Q$ is a query and every variable occurring in $c$ also occurs in $Q$. Moreover, if $or(Q_1, \ldots, Q_n)$ is a disjunctive normal form of $Q$ then every variable of $c$ occurs in each $Q_i$, for $i = 1, \ldots, n$. The construct term $c$ will be sometimes called the head and $Q$ the body of the rule.

If $\Theta$ is the set of all answers for $Q$ and a data term $d$, and $\Theta' \in \Theta/\simeq_{FV(c)}$ then $\Theta'(c)$ is a result for a query rule $c \leftarrow Q$ and $d$.

Each result of a query rule is a data term, as an answer for a query term binds all the variables of the rule to data terms.

Example 3. Consider a database which is a data term:

\[
\text{catalogue}\{ \text{cd[title["Empire Burlesque"], artist["Bob Dylan"], year["1985"]], cd[title["Hide your heart"], artist["Bonnie Tyler"], year["1988"], cd[title["Stop"], artist["Sam Brown"], year["1988"]]} \}
\]

Here is a rule which extracts titles and artists for the CD’s issued in 1988 and presents the results in a changed form (title as name and artist as author). TITLE and ARTIST are variables.

\[
\text{result[name[TITLE], author[ARTIST]]} \leftarrow \text{catalogue}\{ \text{cd[title[TITLE], artist[ARTIST], year["1988"]]} \}
\]

The results returned by the rule are:
The next query rule is similar. It uses all for grouping all the results together and another all for grouping together the CD’s from the same year.

\[
\text{results} \left[ \text{all result} \left[ \text{cds} \left[ \text{year} \left[ \text{YEAR} \right] \right], \text{all name} \left[ \text{TITLE} \right] \right] \right] \leftarrow \\
\text{catalogue}\left\{ \text{cd}\left\{ \text{title} \left[ \text{TITLE} \right], \text{year} \left[ \text{YEAR} \right] \right\} \right\}
\]

The rule returns the following result:

\[
\text{results} \left[ \text{result} \left[ \text{year} \left[ \text{"1988"} \right], \text{cds} \left[ \text{name} \left[ \text{"Hide your heart"} \right], \text{name} \left[ \text{"Stop"} \right] \right] \right] \right], \\
\text{result} \left[ \text{year} \left[ \text{"1985"} \right], \text{cds} \left[ \text{name} \left[ \text{"Empire Burlesque"} \right] \right] \right] \right]
\]

We have defined a result for a query rule and a data term. However a query rule \( p \) can be applied to many data terms at once. Because of the grouping constructs all and some an application of \( p \) to each of data terms separately will not give the same result. Thus we define result for a query rule and a set of data terms.

**Definition 10.** Let \( c \leftarrow Q \) be a construct query rule and \( Z \) a finite set of data terms. Let \( \Theta \) be

- the set of answers for \( Q \) and no data term, if \( Z = \emptyset \),
- the set of answers for \( Q \) and \( d \in Z \), otherwise.

Let \( \Theta' \in \Theta /_{\sim_{FV \left( \Theta \right)}} \). A data term \( \Theta'(c) \) is a result for the query rule \( c \leftarrow Q \) and the set of data terms \( Z \).

### 2.1.6 Programs

Here we present further definitions related to Xcerpt programs.

**Definition 11.** An Xcerpt program \( P \) is a pair \( (P, G) \) where \( P \) and \( G \) are sets of query rules such that \( G \subseteq P \) and \( |G| > 0 \). The query rules from \( G \) are called goals.

Query rules in Xcerpt programs can depend on each other i.e. a query rule can be applied to the results obtained from other query rules in the same program. Here we define the conditions needed to be satisfied in order for one query rule to depend on the other query rule. To avoid complications the presented definition of dependency is simplified and it is not precise i.e. it may happen that even if one query rule \( c_1 \leftarrow Q_1 \) depends directly on the other \( c_2 \leftarrow Q_2 \), the results produced by \( c_2 \leftarrow Q_2 \) cannot be matched by the query \( Q_1 \).

**Definition 12.** Let \( (P, G) \) be an Xcerpt program. Let \( c \leftarrow Q \in P \), \( c' \leftarrow Q' \in P \backslash G \) and \( \text{or}(Q_1, \ldots, Q_n) \) be a disjunctive normal form of \( Q \). The query rule \( c \leftarrow Q \) directly depends on the query rule \( c' \leftarrow Q' \) (shortly, \( c \leftarrow Q \ll c' \leftarrow Q' \)) iff at least one of the following holds:
there exists \( Q_i \) \((1 \leq i \leq n)\) of the form \( \text{and}(q_{i1}, \ldots, q_{ik}) \) such that there exists \( q_{ij} \) \((1 \leq j \leq k)\) which is a variable, a restricted variable, or a query term of the form \( \text{desc} q' \).

- \( c' \) is a variable and there exists \( Q_i \) \((1 \leq i \leq n)\) of the form \( \text{and}(q_{i1}, \ldots, q_{ik}) \) such that there exists \( q_{ij} \) \((1 \leq j \leq k)\) which is not a targeted query term.

- \( c' \) is of the form \( l[\ldots] \) and there exists \( Q_i \) \((1 \leq i \leq n)\) of the form \( \text{and}(q_{i1}, \ldots, q_{ik}) \) such that there exists \( q_{ij} \) \((1 \leq j \leq k)\) which is of the form \( l[\ldots], l[[\ldots]], l\{\ldots\} \) or \( l\{\ldots\} \).

- \( c' \) is of the form \( l\{\ldots\} \) and there exists \( Q_i \) \((1 \leq i \leq n)\) of the form \( \text{and}(q_{i1}, \ldots, q_{ik}) \) such that there exists \( q_{ij} \) \((1 \leq j \leq k)\) which is of the form \( l\{\ldots\} \) or \( l\{\ldots\} \).

- \( c' \) is a basic constant \( b \) and there exists \( Q_i \) \((1 \leq i \leq n)\) of the form \( \text{and}(q_{i1}, \ldots, q_{ik}) \) such that there exists \( q_{ij} = b \) \((1 \leq j \leq k)\).

Example 4. Let \( u \) be some URL and

- \( p_1 = X, Y, Z \leftarrow \text{and}(X, \text{in}(u,b[Y]), d\{Z\}) \),
- \( p_2 = X, Y \leftarrow c\{X, Y\} \),
- \( p_3 = X \leftarrow b[X] \).

Then, \( p_1 \) directly depends on \( p_1, p_2, p_3 \), \( p_2 \) directly depends on \( p_1, p_3 \), and \( p_3 \) does not depend on any of \( p_1, p_2, p_3 \).

Definition 13. Given an Xcerpt program \((P, G)\) a query rule \( p \in P \) depends on a query rule \( p' \in P \) iff there exist query rules \( p_1, \ldots, p_k \in P \), \( k \geq 0 \) such that \( p \ll p_1 \ll \ldots \ll p_k \ll p' \).

Definition 14. An Xcerpt program \((P, G)\) is non recursive iff there is no query rule \( p \in P \) which depends on itself.

Definition 15. Let \((P, G)\) be a non recursive Xcerpt program and \( p \in P \) be a query rule. Let \( P_d \subseteq P \) be the set of query rules on which \( p \) directly depends. A result for the query rule \( p \) in \( P \) (shortly, a result for \( p \)) is defined recursively:

- If \( P_d = \emptyset \) then a result for \( p \) and \( \emptyset \) is a result for \( p \) in \( P \).
- Otherwise, let \( Z \) be the set of all data terms being results for query rules from \( P_d \). A result for \( p \) and \( Z \) is a result for \( p \) in \( P \).

Definition 16. Let \( P = (P, G) \) be a non recursive Xcerpt program. A result of the program \( P \) is a result for a goal \( g \in G \).
2.2 XML Schema Languages

An XML schema language is a metalanguage used to describe classes of XML documents. It is used to specify the structure of a document i.e. the possible arrangement of tags and text. For example, the schema of a book catalog may specify that all entries contain a title and an author, but the publisher is optional. Despite XML documents are not required to have a schema, often they have. If they conform to their schema they are called valid with respect to the schema. The ability to test the validity of documents is an important aspect of large web applications that receive/send information to and from many sources. Independent developers can agree to use a common schema for exchanging XML data and an application can use this agreed upon schema to verify the data it receives.

Many languages for defining schemata are available. This section briefly surveys the most important ones: DTD, XML Schema and Relax NG. Besides them there is a number of less known schema languages like Schematron, Document Structure Description (DSD), Examplotron, Schema for Object-Oriented XML (SOX), Document Definition Markup Language (DDML).

2.2.1 DTD

DTD (Document Type Definition) is a simple and most popular XML schema language. DTD standard is defined by the World Wide Web Consortium (W3C) and it is included in the W3C XML recommendation. DTDs allow to define possible structure of XML documents using the following markup declarations:

- Element Declarations, which are of the form:

  `<ELEMENT element-name content-model>`

  They associate a content model with the elements of the given name. The content model may have the following structure:

  - **EMPTY**: the element has no content
  - **ANY**: the element can have any content
  - `(#PCDATA | element-name | . . .)*` : the element content is an arbitrary sequence of character data and listed elements. This kind of content model is called mixed.
  - deterministic regular expression over element names, which can contain the standard operators: choice `|`, sequence ``, zero or more `*`.

---

2[^2]: [http://www.brics.dk/DSD/dsd2.html](http://www.brics.dk/DSD/dsd2.html)
3[^3]: [http://examplotron.org](http://examplotron.org)
4[^4]: [http://www.w3.org/TR/NOTE-SOX/](http://www.w3.org/TR/NOTE-SOX/)
5[^5]: [http://www.w3.org/TR/NOTE-ddml](http://www.w3.org/TR/NOTE-ddml)
6[^6]: The formal meaning of this requirement is that the regular expressions are 1-unambiguous in the sense of [8].
one or more ‘+’, zero or one ‘?’). The element content is a sequence of elements matching the expression.

- Attribute List Declarations, which are of the form:

  ```xml
  <!ATTLIST element−name attr−name1 attr−type1 qualifier1
  ... attr−name_n attr−type_n qualifier_n >
  ```

  where the `element−name` is the name of the element for which the list of attributes is being defined, `attr−name_i` is the name of the `i`th attribute being defined, `attr−type_i` defines the type of data that may be used for the value. The possible types of attributes are:
  
  - CDATA - used to specify a string type.
  - ENTITY - reference to an external file such as a graphic file for importing an image.
  - ENTITIES - used to include multiple entities.
  - ID - used for defining occurrences of identifiers. There can be only one unique value used as identifier.
  - IDREF - used for referring occurrences of identifiers.
  - IDREFS - used for referring occurrences of multiple identifiers.
  - `(val_1 | ... | val_k)` - used for an enumeration type. This is a list of allowed values of the attribute.
  - NMTOKEN - a string type with some additional restrictions.
  - NMTOKENS - a list of multiple name tokens.
  - NOTATION - see below.

  A qualifier `qualifier_i` is used in the declaration of an attribute to additionally specify its value. It can be:
  
  - a default value - the character data (CDATA) in a quoted string form
  - #FIXED value - used to fix the value of the attribute
  - #IMPLIED - used if the value of the attribute is optional
  - #REQUIRED - used if the value of the attribute is mandatory

- Entity Declarations, which are of the form:

  ```xml
  <!ENTITY entity−name "entity−value" >
  ```

  Entities are variables used to define shortcuts to common text (when used in the DTD, the string value is substituted for the entity name). They can be also used to include binary data in an XML document, like a PNG (Portable Network Graphics).
Notation Declarations, which are of the form:

```xml
<!NOTATION notation-name SYSTEM location >
```

Notation declarations can be used to identify external binary formats and to specify helper applications for processing the format. The reference is given by the `location` which is a universal resource identifier (URI) for a file name which may specify a local path or a complete path over the Internet e.g.

```xml
<!NOTATION pl SYSTEM /usr/bin/perl >
```

**Example 5.** The following DTD defines a structure of an XML document for a book store:

```xml
<!ELEMENT bib (book* )>
<!ELEMENT book (title, (author+ | editor+ ), publisher, price )>
<!ATTLIST book year CDATA #REQUIRED >
<!ELEMENT author (last, first )>
<!ELEMENT editor (last, first, affiliation )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
<!ELEMENT affiliation (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price (#PCDATA )>
```

A document conforming to this schema has `bib` as a main element. `bib` element contains zero or more `book` elements, each of them having a `title`, a list of authors or a list of editors (of the size one at least), a `publisher` and `price`. Additionally, each `book` element has a mandatory attribute `year`. Each element `author` contains elements `last` and `first`; an element `editor` besides elements `last` and `first` contains an element `affiliation`. The content of the remaining elements is text.

DTD is a simple XML schema language and it has a number of obvious limitations:

- DTD schemata are written in a non-XML syntax.
- They do not allow context dependent definitions of elements; it is thus not possible to define an element `title` as a child of an element `book` and then define another element `title` with different structure for a chapter.
- They have no support for namespaces.
- They only support a limited number of simple datatypes i.e. types restricting the values of text nodes.
2.2. XML SCHEMA LANGUAGES

2.2.2 XML Schema

XML Schema [25, 31, 41] is an alternative schema language which is more powerful but also more complex than DTD. It provides more precision in describing document structures and contents of text nodes. In contrast to DTD, it allows for context dependent definitions of elements. An important advantage of XML Schema is that schemata are specified in XML so no special syntax is needed.

XML Schema uses two kinds of types: simple types and complex types, both of which constrain the allowable content of an element or attribute.

Simple Types

Simple types restrict the text that is allowed to appear as an attribute value, or text-only element content (text-only elements do not carry attributes or contain child elements). Simple types can be primitive (hardwired meaning) or derived from existing simple types. Derivation may be

- by a list: white-space separated sequence of elements of other simple types
- by a union: union of other simple types
- by a restriction, for instance a restriction on a list length (\textit{minLength}, \textit{maxLength}), bounds on numbers (\textit{minInclusive}, \textit{maxInclusive}), restriction on text using patterns (Perl-like regular expressions)

XML Schema provides a number of predefined simple types (all the primitive and some derived) that are often used such as: string, integer, float, date, etc.

Example 6. This is an example of a declaration of a simple type \textit{april\_date}, which is a restriction of a simple type \textit{date}.

\begin{verbatim}
<simpleType name="april_date">
  <restriction base="date">
    <pattern value="\d{4}-04-\d{2}"/>
  </restriction>
</simpleType>
\end{verbatim}

The elements of the type \textit{april\_date} are those elements of type \textit{date} which match the given pattern i.e. they have "04" as a substring corresponding to the month number.

Complex Types

Complex types restrict the allowable content of elements, in terms of the attributes they can carry, and child elements they can contain. A Complex Type declaration may contain:

- attribute declarations:
CHAPTER 2. BACKGROUND

– <attribute name=".." type=".." use=".."/>
  where type refers to a simple type definition and use is either optional, required, or prohibited.

– <anyAttribute ... /> allows the insertion of any attribute.

• one of the following content model kinds:

  – empty content model
  – simple content model (only text is allowed)
  – complex content model: a (restricted) combination of
    * <sequence> ... </sequence>
    * <choice> ... </choice>
    * <all> ... </all>
    containing element references or declarations of the form
    * <element ref=".." minOccurs=".." maxOccurs=".."/> where ref refers to an element definition, and minOccurs and maxOccurs constrain the number of occurrences
    * <element name=".." type=".." .../>
    * <any .../> allows the insertion of any element.
  – mixed content model: implemented through a mixed attribute in complexType element definition. The effect of this attribute when its value is set to "true" is to allow any text nodes within the content model.

XML Schema requires complex content models to be deterministic i.e. they must satisfy "Unique Partition Attribution Rule" [41]. This restriction is similar to the restriction put on content models in DTD and can be interpreted as the requirement that the content models are 1-unambiguous in the sense of [8]. Another restriction related to a complex content model is called "Consistent Declaration rule". It says that the content model cannot contain two different types of elements with the same name.

XML Schema allows to define element and attribute groups which are containers in which sets of elements and attributes may be embedded and manipulated as a whole. They are very convenient for defining fragments of content models that can be reused in multiple locations, such as e.g. choice structure.

Example 7. This is a definition of a type OrderType with a mixed content model. Any element of this type must contain id attribute and an element address or an element phone with zero or more email elements:

<complexType name="OrderType" mixed="true">
<choice>
2.2. XML SCHEMA LANGUAGES

XML Schema provides a type derivation mechanism also for complex types. New complex types may be derived by extending or by restricting a content model of an existing type.

- Derivation by extension: The effective content model of a new type is the content model of the base type concatenated with the content model specified in the type derivation declaration. Elements added via extension are treated as if they were appended to the content model of the base type in sequence. For instance, the type USAddress has been derived by extension from the type Address. The content model of USAddress is the content model of Address plus the declarations of state and zip elements:

```xml
<complexType name="Address">
  <sequence>
    <element name="name" type="string"/>
    <element name="street" type="string"/>
    <element name="city" type="string"/>
  </sequence>
</complexType>

<complexType name="USAddress">
  <complexContent>
    <extension base="Address">
      <sequence>
        <element name="state" type="USState"/>
        <element name="zip" type="positiveInteger"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

- Derivation by restriction: The values of the new type are a subset of the values of the base type (as is the case with restriction of simple types). The new type is defined in an usual way but with declaration that it is a restriction of some other type. For example, a full definition of the type RestrictedPurchaseOrderType is provided but with the indication that it is derived by restriction from the base type PurchaseOrderType. Indeed the new type RestrictedPurchaseOrderType is a subset of the base type.
PurchaseOrderType as a purchase order of the new type must contain a child element comment while a purchase order of the base type may not contain it:

```xml
<complexType name="PurchaseOrderType">
  <sequence>
    <element name="shipTo" type="Address"/>
    <element name="billTo" type="Address"/>
    <element ref="comment" minOccurs="0"/>
    <element name="items" type="Items"/>
  </sequence>
</complexType>

<complexType name="RestrictedPurchaseOrderType">
  <complexContent>
    <restriction base="PurchaseOrderType">
      <sequence>
        <element name="shipTo" type="Address"/>
        <element name="billTo" type="Address"/>
        <element ref="comment" minOccurs="1"/>
        <element name="items" type="Items"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

2.2.3 Relax NG

XML schema language Relax NG [17, 42] has been defined by the Oasis consortium. It is much more expressive than DTD and it allows to specify things which are not expressible in XML Schema. While still being simple and easy to learn and maintain, Relax NG is capable to describe XML documents of high structural complexity and it is able to handle a huge range of applications. It has two syntaxes: XML syntax, which can be used by many existing tools like XML editors or browsers, and a compact non-XML syntax which is well readable for human beings. For this reason we will use the latter in this thesis.

Relax NG has a solid theoretical foundation in the theory of tree automata. A schema is built based on the production rules from the regular tree grammar. The left-hand side of a rule is a nonterminal symbol and the right hand side can be text, a datatype from an external library (e.g. XML Schema Datatypes [1]), an ordered or an unordered list of element definitions, attribute definitions, and alternatives of the former constructs.

To introduce elements the keyword element is used, followed by a label and a content model. For example,

```
Title = element title { text }
```
defines a nonterminal symbol Title which describes elements labelled title with only text content.

The content model of an element is a list of nonterminals or further definitions separated by ',' (ordered sequence), by '&,' (unordered groups), or by '|' (alternatives). To specify repetitions of elements operators * and + can be used, similarly as in regular expressions.

The following grammar defines element book containing an unordered sequence of an element title and one or more elements author:

```
Book = element book{
    element title { text } &
    element author { text }+

`

The operator '&,' is called *interleave* operator and it has a complex semantics. It is not only used to define groups of elements which can occur in any order but also it allows the elements of separate groups to interleave. Consider the following example:

```
Book = element book{
    element title { text } &
    (element author { text }+,
    element editor { text }+)

`

According to the abovementioned definition, an element book contains a list of authors followed by a list of editors. Additionally, it contains an element title which can occur at any position e.g. between elements author and editor.

Attributes are introduced by the keyword attribute followed by the attribute name and a specification of its allowable value.

The following fragment of a grammar defines element book as in the previous example but additionally it specifies its attribute id whose allowable value is of type ID imported from the external library XML Schema Datatype (prefix xsd:). Note that the children of the element book have been defined outside its content model by introducing new nonterminal symbols Title and Author:

```
Book = element book{
    attribute id { xsd:ID } &
    Title &
    Author+
}
Title = element title { text }
Author = element author { text }
```
Chapter 3

Type Specification

This chapter introduces a formalism called Type Definitions which we use for defining classes of data terms. Type Definitions play a similar role as schema languages for XML. They are not meant to be a yet another competitive schema language but rather a kind of abstraction of the existing schema languages providing a common view of them. Such an abstraction is necessary to be able to handle types defined by different schema languages in one application and to be able to compare them.

Our intended application requires that basic operations on sets expressed in the formalism (like intersection and checks for membership, emptiness and inclusion) are decidable and efficient algorithms for them exist. A well known such formalism is that of tree automata [18] (or tree grammars, which are just another view of tree automata). However tree automata deal with terms where each symbol has a fixed arity. This is not sufficient in our case since in XML, the number of elements between a given pair of a start-tag and end-tag is not fixed. That is why our Type Definitions are based on unranked tree automata [7, 32] which combine tree grammars with regular expressions. The latter are used to describe the possible sequences (or sets) of children of a single node in a tree. The novelty of our approach is that we deal with mixed trees where the order of children of a node may be irrelevant.

An important problem in a type system for an XML query language is type checking: checking whether the results of queries (or transformations) applied to XML data from a given type are within an another given type. Existence of efficient algorithms for type checking of XML queries is of great importance and it has been quite intensively investigated. Various cases of such type checking problems for different XML query languages have been studied e.g in [3, 30] and references therein. The papers deal with automata as abstractions of query languages and show that the problem is often undecidable or of non-polynomial complexity. They propose solutions employing various restrictions on schema languages or on classes of XML queries or transformations. In our work we are focused on the particular query language Xcerpt. In order to perform type operations efficiently we also propose a restriction on Type Definitions which re-
results in more efficient algorithms for type checking. The restricted class of Type Definitions is called proper and it corresponds to a single type tree grammar in the sense of [32, 21].

3.1 Type Definitions

This section introduces a formalism for specifying classes of decidable sets of data terms representing XML documents. First we specify a set of type names $T = C \cup S \cup V$ which consist of

- **type constants** from the alphabet $C$
- **enumeration type names** from the alphabet $S$
- **type variables** from the alphabet $V$

We associate each type name $T$ with a set $[T]$ (the type denoted by $T$) of data terms which are allowed values assigned to $T$. For $T$ being a type constant or an enumeration type name, the elements of $[T]$ are basic constants. Type constants correspond to XML schema language base types. The set of type constants is fixed and finite. In our examples we will use a type constant Text assuming that $[\text{Text}]$ is the set of non empty strings of characters. This is similar to #PCDATA in DTD. We also assume that Text is a union of all types represented by type constants and enumeration type names.

Each type variable $T$ is associated with a set of data terms $[T]$ which is specified in a way similar to that of [12] and described below. Similarly, each enumeration type name $T$ is associated with a finite set $[T]$ of basic constants.

First we introduce some auxiliary notions. The empty string will be denoted by $\epsilon$. A regular expression over an alphabet $\Sigma$ is $\epsilon, \phi, a \in \Sigma$ and any $r_1, r_2, r_1|r_2$ and $r_1^*$, where $r_1, r_2$ are regular expressions. A language $L(r)$ of strings over $\Sigma$ is assigned to each regular expression $r$ in the standard way (see e.g. [28]). In particular, $L(\phi) = \emptyset$, $L(\epsilon) = \{\epsilon\}$ and $L(r_1|r_2) = L(r_1) \cup L(r_2)$.

**Definition 17.** A regular type expression is a regular expression over the alphabet of type names $T$. We abbreviate a regular expression $r^n|r^{n+1}|\cdots|r^m$, where $n \leq m$, as $r^{[n:m]}$, $r^n r^* \equiv r^{(n:\infty)}$, $rr^* \equiv r^+$, and $r^{(0:1)} \equiv r^?$. A regular type expression of the form

$$r_1 \cdots r_k$$

where $k \geq 0$, each $r_i$ is $T_i^{(n_i:1:n_i:2)}$, $0 \leq n_i:1 \leq n_i:2 \leq \infty$ for $i = 1, \ldots, k$, and $T_1, \ldots, T_k$ are distinct type names, will be called a multiplicity list.

Multiplicity lists will be used to represent multisets of type names. The union of two multiplicity lists $r_1$ and $r_2$ can be obtained by concatenation of $r_1, r_2$ and replacing each pair of expressions of the form $T^{(i:u)}$ and $T^{(i':u')}$ by the expression $T^{(i+i':u+u')}$.
Definition 18. A Type Definition is a set \( D \) of rules of the form

\[
T \rightarrow l[r], \quad T \rightarrow l\{s\}, \quad \text{or} \quad T' \rightarrow c_1 \ldots | c_n,
\]

where \( T \) is a type variable, \( T' \) an enumeration type name, \( l \) a label, \( r \) a regular type expression, \( s \) a multiplicity list, and \( c_1, \ldots, c_n \) are basic constants. A rule \( U \rightarrow G \in D \) will be called a rule for \( U \) in \( D \). We require that for any type name \( U \in V \cup S \) occurring in \( D \) there is exactly one rule for \( U \) in \( D \).

If the rule for a type variable \( T \) in \( D \) is as above then \( l \) will be called the label of \( T \) (in \( D \)) and denoted \( \text{label}_D(T) = l \). For \( T \) being a type constant or an enumeration type name we define \( \text{label}_D(T) = \$ \). The regular expression in a rule for type variable \( T \) is called the content model of \( T \). If a rule for a type variable \( T \) in \( D \) is \( T \rightarrow l[r] \) (or \( T \rightarrow l\{r\} \)) then \( [ \] \) (or \( \{ \} \), respectively) are called the parentheses of \( T \).

The formalism of Type Definitions is a certain simplification of the formalism of [12]. The difference concerns type constants which in our approach are assumed to have labels. This makes it possible to treat them in a simpler way.

Example 8. Consider a Type Definition \( D \):

\[
\begin{align*}
Cd & \rightarrow cd[\text{Title Artist}^+ \text{ Category}^2] \\
Title & \rightarrow \text{title[Text Subtitle}^2] \\
Subtitle & \rightarrow \text{subtitle[Text]} \\
Artist & \rightarrow \text{artist[Text]} \\
Category & \rightarrow "\text{pop}\} | "\text{rock}\} | "\text{classic}\}
\end{align*}
\]

\( D \) contains a rule for each of type variables: \( Cd \), \( Title \), \( Subtitle \), \( Artist \) and a rule for enumeration type name \( Category \). Labels occurring in \( D \) are: \( \text{cd} \), \( \text{title} \), \( \text{subtitle} \), \( \text{artist} \), \( \text{pop} \), \( \text{rock} \), \( \text{classic} \) are basic constants.

Type Definitions are a kind of grammars, they define sets by means of derivations, where a type variable \( T \) is replaced by the right hand side of the rule for \( T \) and a regular expression \( r \) is replaced by a string from \( L(r) \); if \( T \) is a type constant or an enumeration type name then it is replaced by a basic constant from respectively \( \text{[}[T]\text{]} \), or from the rule for \( T \). This can be concisely formalized as follows (treating type definitions similarly to tree automata).

Definition 19. Let \( D \) be a Type Definition. We will say that a data term \( t \) is derived in \( D \) from a type name \( T \), iff there exists a mapping \( \nu \) from the subterms of \( t \) to type names such that \( \nu(t) = T \) and for each subterm \( u \) of \( t \)

- if \( u \) is a basic constant then \( \nu(u) \in C \) and \( u \in L(\nu(u)) \) or \( \nu(u) \in S \) and there exists a rule \( \nu(u) \rightarrow \ldots | u \ldots \) in \( D \).
- otherwise \( \nu(u) = U \in V \) and
  - there is a rule \( U \leftarrow l[r] \in D, u = l[t_1, \ldots, t_n], \) and \( \nu(t_1) \ldots \nu(t_n) \in L(r) \).
CHAPTER 3. TYPE SPECIFICATION

The set of the data terms derived in $D$ from a type name $T$ will be denoted by $[T]_D$.

Example 9. For the Type Definition $D$ from the previous example, we have that the data term

$$t = cd[title]["Stop"], artist["Sam Brown"], "pop"]$$

is derived from the type variable $Cd$. The type names assigned to the three arguments of $cd$ are, respectively, Title, Artist, Category, and the type constant Text is assigned to the constants "Stop", and "Sam Brown".

Notice that if $T$ is a type constant then $[T]_D = [T]$. If it is clear from the context which type definition is considered, we will often omit the subscript in the notation $[T]_D$ and similar ones. For $U$ being a set of type names $\{T_1, \ldots, T_n\}$, we define a set of data terms $[U] = [T_1] \cup \ldots \cup [T_n]$. For a regular type expression $r$ we define $[r] = \{d_1, \ldots, d_n \mid d_1 \in [T_1], \ldots, d_n \in [T_n] \}$ for some $T_1, \ldots, T_n \in L(r)$. Notice that if $D \subseteq D'$ are Type Definitions then $[T]_D = [T]_{D'}$ for any type name $T$ occurring in $D$. We use $\text{types}_D(r)$ to denote the set of all type names occurring in the regular expression $r$. We define a set of type names with a given label $l$ occurring in a regular expression $r$ as $\text{types}_D(l, r) = \{T \mid T \in \text{types}_D(r) \text{ and } label(T) = l\}$. Observe that if $d \in [T]$ then $\text{root}(d) = label(T)$.

3.1.1 Proper Type Definitions

For our analysis of Xcerpt rules we need algorithms computing intersection of sets defined by Type Definitions, and performing emptiness and inclusion checks for such sets. To obtain efficient algorithms we impose certain restrictions on Type Definitions. They are discussed in this section.

Consider a Type Definition $D$ and a content model $r$ occurring in $D$. We say that the content model $r$ is proper, if it does not contain two distinct type names having the same label. We call the content model $r$ weakly proper, if the following property holds. If two distinct type names occurring in $r$ have the same label then they are type variables and they have different kind of parentheses.

We say that a Type Definition $D$ is proper, if all contents model occurring in $D$ are proper. Thus given a term $l[c_1 \ldots c_n]$ and a rule $T \rightarrow l[r] \in D$ or a term $l[c_1 \ldots c_n]$ and a rule $T \rightarrow l[r] \in D$ for each $c_i$ determines at most one type name $S$ such that $S$ occurs in $r$ and $label_D(S) = root(c_i) = l_i$. For any proper content model $r \in D$ and arbitrary label $l$ a set $\text{types}_D(l, r)$ contains at most one element. Notice that, for a proper Type Definition $D$, at most one type constant or enumeration type name occurs in any regular expression of $D$ since all type constants and enumeration type names have the same label $\$. 
We say that a Type Definition $D$ is \textbf{weakly proper}, if all contents model occurring in $D$ are weakly proper.

The class of proper Type Definitions, when restricted to ordered terms (i.e. without $\{}$), is essentially the same as single-type tree grammars of \cite{33}. Restriction to proper Type Definitions results in simpler and more efficient algorithms although it imposes some limitations. We will state explicitly if we require a Type Definition to be proper (or weakly proper).

**Example 10.** Type Definition $D_1 = \{ A \rightarrow a[A | B | C], B \rightarrow b[D], C \rightarrow b[Text], D \rightarrow c[Text] \}$ is not proper because type names $B, C$ have the same label $b$ and occur in one regular expression. In contrast, $D_2 = \{ A \rightarrow a[A | B | D], B \rightarrow b[CD], C \rightarrow b[Text], D \rightarrow c[Text] \}$ is proper and e.g. $\text{types}_{D_2}(A | B | D) = \{ A, B, D \}$, $\text{types}_{D_2}(b, A | B | D) = \{ B \}$ and $\text{types}_{D_2}(b, CD) = \{ C \}$.

### 3.2 Operations on Types

In this section we describe algorithms for basic operations on types: check for emptiness, computing intersection, and check for inclusion. The algorithms for latter two operations employ some standard operations on languages described by regular expressions like inclusion and equality checks, computing intersection of such languages. This can be done by transforming regular expressions to deterministic finite automata (DFA’s) and using standard efficient algorithms for DFA’s.

In the general case the number of states in a DFA may be exponentially greater than the length of the corresponding regular expression \cite{28}. Notice that the XML definition \cite{24} requires (Section 3.2.1) that content models specified by regular expressions in element type declarations of a DTD are \textit{deterministic} in the sense of Appendix E of \cite{24}. The formal meaning of this requirement is that the regular type expressions are 1-unambiguous in a sense of \cite{8}. For such regular expressions a corresponding DFA can be constructed in linear time.

#### 3.2.1 Emptiness Check

We show how to check if a type defined by a type definition is empty. In what follows we assume that the regular expressions in Type Definitions do not have useless symbols. A type name $T$ is \textit{useless} in a regular expression $r$ if no string in $L(r)$ contains $T$. (If $r$ contains a useless symbol then the regular expression $\phi$ occurs in $r$.)

A type name $T$ in a Type Definition $D$ will be called \textbf{nullable} if no data terms can be derived from $T$. In other words, $[T]_D = \emptyset$ iff $T$ is nullable in $D$.

To find nullable symbols in a Type Definition $D$ we mark type names in $D$ in the following way. First we mark all type constants and all enumeration type names (that do not denote $\emptyset$). Then we mark each unmarked type variable $T_i$ in $D$ with the rule for $T_i$ of the form $T_i \rightarrow l[r_i]$ or of the form $T_i \rightarrow l\{r_i\}$ such that there exists a sequence of marked type names $S_1 \ldots S_m \in L(r_i)$ ($m \geq 0$).
We repeat the second step until an iteration which does not change anything. The type names which are unmarked in $D$ are nullable.

Here, we explain how to check whether there exists a sequence of marked type names $S_1\ldots S_m \in L(r)$ ($m \geq 0$). Let $\lambda$ be a parse tree of $r$ (e.g. a parse tree for a regular expression $((T_1^*|T_2|T_3)^*|T_1^*|T_2^*|T_3^*)$ represented as a term is $\text{or(then(or(star(T_1),T_2),T_1),T_3)}$). We walk on the tree starting from its root. For each visited node we do the following:

- if a node is an unmarked type name we replace it by $\phi$ (the node is a leaf)
- if a node is $\text{star}$ we replace it by $\epsilon$ and remove its child (the node becomes a leaf)
- if a node is $\text{or}$ we visit its children. If both of them were replaced by $\phi$ we replace the node by $\phi$; otherwise we replace it by a child which was not replaced by $\phi$
- if a node is $\text{then}$ we visit its children.

If the result tree does not have any $\phi$ node, there exists a sequence of marked type names which belongs to $L(r)$. Otherwise, such a sequence does not exist. Assuming that the tree $\lambda$ has $n$ nodes the time complexity of the operation is $O(n)$.

If the number of types defined by $D$ is $m$ the check must be done at most $m^2$ times. Thus, the worst case time complexity of the type emptiness checking is $O(m^2 n)$.

Example 11. Let us use the algorithm to find nullable type names in a Type Definition $D = \{ A \rightarrow a[AB], B \rightarrow b[B^*] \}$. The initial step does not mark any type names. In the second step we mark $B$ because $\epsilon \in L(B^*)$. In the next iteration we cannot mark any other type names and the algorithm stops. Since $A$ is unmarked, it is nullable.

### 3.2.2 Intersection of Types

Here we explain a way of obtaining the intersection of two types $T$ and $U$ defined by a Type Definition $D$. We do not require that $D$ is proper. A simpler algorithm for type intersection of types defined by a proper Type Definition was presented in [43]. The algorithm we present can be applied for types defined by a non proper Type Definition $D$ but in general it may produce results which are approximations. It produces exact results i.e. $[T \cap U] = [T] \cap [U]$ if a Type Definition $D$ does not contain non weakly proper multiplicity lists. In general case the type $T \cap U$ may be an approximation of the intersection of the types $U$ and $T$ (i.e. a superset of $[T] \cap [U]$). Such approximation is necessary if there is a need to intersect types whose content model is not a weakly proper multiplicity list. This is because the intersection of non weakly proper multiplicity lists may be unable to be expressed by a multiplicity list. A method for obtaining a
weakly proper approximation of a multiplicity list is presented later on in this section.

Now, we are ready to present an algorithm for obtaining the intersection of types \( T, U \). We assume that the types \( T, U \) are defined by a Type Definition \( D \) which does not contain non weakly proper multiplicity lists. In the following algorithm we do not distinguish type names \( D \) of \( T, U \) is also proper.

3.2. OPERATIONS ON TYPES

If the Type Definition \( D \) contains the rule \( T \rightarrow c_1 \ldots c_n \), where \( [T] \cap [U] = \{c_1, \ldots, c_n\} \).

If one of the type names \( T, U \) is a type variable and the other is a constant type then their intersection is empty.

If \( T \) and \( U \) are type variables and \( \text{label}_D(T) \neq \text{label}_D(U) \) then their intersection is empty.

If \( T \) and \( U \) are type variables and \( D \) contain rules of the form \( T \rightarrow l \{r_1\} \) and \( U \rightarrow l \{r_2\} \) or of the form \( T \rightarrow l \{r_1\} \) and \( U \rightarrow l \{r_2\} \) then the intersection of \( T, U \) is empty.

If \( D \) contains rules of the form \( T \rightarrow l \{r_1\} \) and \( U \rightarrow l \{r_2\} \) then \( [T] \cap [U] \) is defined by a rule \( T \cap U \rightarrow l \{r_1\} \), where \( L(r) = L(r_1') \cap L(r_2') \) and \( r_1', r_2' \) are obtained in the following way. For each \( T_i \) occurring in \( r_1 \) let \( S_i = \{U_{i1}, \ldots, U_{ik_i}\} \) be the set of type names occurring in \( r_2 \) such that all type names from each \( S_i \) have the same label as \( T_i \) and if \( T_i \) is a type variable they have also the same kind of parentheses. \( r_1' \) is \( r_1 \), where each symbol \( T_i \) is replaced by a regular expression \( T_i \cap U_{1i} \ldots U_{ik_i} \) (or \( \phi \), if \( k_i = 0 \)) and \( T_i \cap U_{ij} \) are new type names. \( r_2' \) is obtained analogically to \( r_1' \) (new type names are \( T_{ij} \cap U_{ij} \)). For each new type name \( T' \cap U' \) occurring in \( r \) a rule defining the type \( [T' \cap U'] = [T'] \cap [U'] \) must be defined.

If \( D \) contains rules of the form \( T \rightarrow l \{r_1\} \) and \( U \rightarrow l \{r_2\} \) (where \( r_1, r_2 \) are weakly proper) then \( [T] \cap [U] \) is defined by a rule \( T \cap U \rightarrow l \{r\} \), where \( r \) is obtained in following way. Let \( r_1' \) be \( r_1 \), where each \( T_i \) is replaced by a type name \( T_i \cap U_i \) and \( U_i \) is a type name occurring in \( r_2 \) such that \( \text{label}_D(U_i) = \text{label}_D(T_i) \) and if \( T_i \) is a type variable they have also the same kind of parentheses. If there is no such a type name \( U_i \) in \( r_2 \) then \( T_i \) is replaced by \( \phi \). Let \( r_2' \) be obtained from \( r_2 \) in the same way as \( r_1' \) is obtained from \( r_1 \). Let \( r_2' \) be of the form \( s_{1}^{(l_1' ; u_1')} \ldots s_{n}^{(l_n' ; u_n')} \) and let \( r_2' \) be \( s_{1}^{(l_1' ; u_1')} \ldots s_{n}^{(l_n' ; u_n')} \). The multiplicity list \( r \) is \( s_{1}^{(l_1; u_1)} \ldots s_{n}^{(l_n; u_n)} \) where each \( l_i' = \max(l_i, l_i') \) and each \( u_i' = \min(u_i, u_i') \). For each new type name \( T' \cap U' \) occurring in \( r \) a rule defining the type \( [T' \cap U'] = [T'] \cap [U'] \) must be defined.

If the Type Definition \( D \) is proper the new Type Definition defining the type \( T \cap U \) is also proper.
Example 12. Consider a Type Definition \( D = \{ A \rightarrow l[B|C], B \rightarrow l[A^+], C \rightarrow l[], A' \rightarrow l[A'^+|C'], C' \rightarrow l[C'^+] \} \). We construct a type definition \( D' \) which defines a type \( A \cap A' \) being the intersection of types \( A \) and \( A' \). \( \{ A \cap A' \}_{D'} = [A]_{D} \cap [A']_{D} \). 
\( D' = \{ A \cap A' \rightarrow l[B \cap A'|C \cap C'], B \cap A' \rightarrow l[(A \cap A')^+] \} \). Example 14 will show that \( [A]_{D} \subseteq [A']_{D} \) and that is why \( [A \cap A']_{D'} = [A]_{D} \).

The presented algorithm employs an operation of intersection of two regular languages \( L(r_1) \) and \( L(r_2) \). To intersect \( L(r_1) \) and \( L(r_2) \) we need to build automata representing both languages and then build the product automaton. If the regular expressions are 1-unambiguous \( \mathcal{S} \), the automata representing the languages can be built in linear time and building the product automaton requires polynomial time. Otherwise the complexity of intersection of \( L(r_1) \) and \( L(r_2) \) is exponential.

Assume that \( D \) contains \( m_1 \) rules, there is \( m_2 \) type constants and \( m = m_1 + m_2 \). To intersect two types \( T_1, T_2 \) defined in \( D \), in the worst case we may need to compute intersection of regular languages \( m^2 \) times. Thus, if \( D \) does not contain regular expressions which are not 1-unambiguous then the complexity of type intersection algorithm is polynomial. Otherwise, it is exponential.

Now, we present a way a non weakly proper multiplicity list \( r \) can be approximated by a weakly proper multiplicity list \( r' \). Let \( S_C, S_1, \ldots, S_n \) be disjoint sets of type names such that \( S_C \cup S_1 \ldots \cup S_n = \text{typesD(r)} \) and \( S_C \) is a set of type constants or enumeration type names and each \( S_i \) is a set of all type variables (occurring in \( r \)) with the same label and the same kind of parentheses. For the sets of type names \( S_C, S_1, \ldots, S_n \) we construct corresponding types \( T_C, T_1, \ldots, T_n \) representing unions of types from each set:

- If \( S_C \) contains only enumeration type names \( T_C \) is an enumeration type name. Otherwise, \( T_C \) is Text (as Text is a union of all types represented by type constants or enumeration type names).
- For a set \( S_i \) of type names defined by rules of the form \( T_{ij} \rightarrow l_i[r_{ij}] \) the corresponding type \( T_i \) is defined as \( T_i \rightarrow l_i[r_{i1} | \cdots | r_{ik}] \).
- For a set \( S_i \) of type names defined by rules of the form \( T_{ij} \rightarrow l_i\{r_{ij}\} \) the corresponding type \( T_i \) is defined as \( T_i \rightarrow l_i\{u_i\} \), where \( u_i \) is a multiplicity list obtained in the following way. Let \( u'_i \) be a multiplicity list being a union of \( r_{i1}, \ldots, r_{ik} \). If \( u'_i \) is weakly proper \( u_i = u'_i \). Otherwise, \( u'_i \) must be approximated by a weakly proper multiplicity list \( u_i \) (using the same algorithm).

Finally, \( r' \) is \( r \) where every type name \( T \) is replaced by the type name \( T_s \in \{ T_C, T_1, \ldots, T_n \} \) such that \( T \in S_s \) (where \( S_s \in \{ S_C, S_1, \ldots, S_n \} \)). In order for \( r' \) to be a multiplicity list all expressions of the form \( T_S^{(l_1, u_1)}, \ldots, T_S^{(l_k, u_k)} \) must be replaced by en expression \( T_S^{(l_1 + \cdots + l_k, u_1 + \cdots + u_k)} \).

Example 13. The example shows how to obtain the intersection of types \( T_1, T_2 \), where \( T_1 \) is defined with non weakly proper multiplicity list. Consider a Type
3.2. OPERATIONS ON TYPES

Definition $D = \{ T_1 \rightarrow l\{A_1^1, A_2^2\}, T_2 \rightarrow l\{A^+\}, A \rightarrow a[C^*], A_1 \rightarrow a[\cdot], A_2 \rightarrow a[CC], C \rightarrow c[\cdot]\}$. As $A_1 \cap A = A_1$ and $A_2 \cap A = A_2$ the intersection of the types $T_1$ and $T_2$ would be expressed as $T_1 \cap T_2 \rightarrow l\{ A_1 | A_2 | (A_1 A_2) \}$. This is however not allowed as the content model of $T_1 \cap T_2$ is not a multiplicity list (and it cannot be represented as a multiplicity list). Thus, before intersecting $T_1, T_2$ we approximate the multiplicity list $A_1^1 A_2^2$ by a weakly proper one. As the types $A_1, A_2$ have the same label and the same kind of parentheses they can be approximated as a type $A'$ being their union and defined as $A' \rightarrow a[CC]^\ast$. Thus the multiplicity list $A_1^1 A_2^2$ can be approximated as $A'^0 A'^0$ which is equal to a weakly proper multiplicity list $A'^{(0:2)}$. The type $T_1$ from $D$ can be approximated as a type $T_1'$ with a weakly proper multiplicity list $T_1' \rightarrow \{A'^{(0:2)}\}$. The intersection of types obtained using the presented algorithm is $T_1' \cap T_2 \rightarrow l\{A'^{(1:2)}\}$. The type $[T_1' \cap T_2]$ is an approximation of the type $[T_1' \cap T_2]$.

3.2.3 Type Inclusion

The algorithm presented here is based on the approach taken in [12].

Let $T_1, T_2$ be type names defined in Type Definitions $D_1, D_2$, respectively. $T_1$ is an inclusion subtype of $T_2$ iff $[T_1]_{D_1} \subseteq [T_2]_{D_2}$. We present an algorithm which checks this fact. It is required that $D_2$ is proper.

The first part of the algorithm constructs a set $C(T_1, T_2)$ of pairs of types to be compared. It is the smallest set such that

- if $\text{label}(T_1) = \text{label}(T_2)$ then $(T_1, T_2) \in C(T_1, T_2),$

- if

  - $(T_1', T_2') \in C(T_1, T_2),$
  
  - $D_1, D_2$ contain, respectively, rules $T_1' \rightarrow l\{r_1\}$ and $T_2' \rightarrow l\{r_2\}$, or $T_1' \rightarrow l\{r_1\}$ and $T_2' \rightarrow l\{r_2\}$ (with the same label $l$), and

  - type names $T_1''$, $T_2''$ occur respectively in $r_1, r_2$, and $\text{label}_{D_1}(T_1'') = \text{label}_{D_2}(T_2'')$

then $(T_1'', T_2'') \in C(T_1, T_2)$. As $D_2$ is proper, for every $T_1''$ in $r_1$, there exists at most one $T_2''$ in $r_2$ satisfying this condition.

The second part of the algorithm checks whether $[T_1'] \subseteq [T_2']$ for each $(T_1', T_2') \in C(T_1, T_2)$:
CHAPTER 3. TYPE SPECIFICATION

34

IF \( C(T_1, T_2) = \emptyset \) THEN return false
ELSE for each \((T'_1, T'_2) \in C(T_1, T_2)\) do the following:
   IF \( T'_1, T'_2 \) are enumeration type names or type constants
   THEN check whether \([T'_1] \subseteq [T'_2]\) and return the result
   Let \( T'_1 \rightarrow l[r_1] \) and \( T'_2 \rightarrow l[r_2] \), or \( T'_1 \rightarrow l\{r_1\} \) and \( T'_2 \rightarrow l\{r_2\} \)
   be rules of \( D_1, D_2 \), respectively
   Let \( s_1 \) and \( s_2 \) be the regular expressions over labels
   corresponding to \( r_1 \) and \( r_2 \)
   Check whether \( L(s_1) \subseteq L(s_2) \)
   IF for all pairs from \( C(T_1, T_2) \) the answer is true THEN return true
   ELSE return false

The algorithm employs a check if \([T'_1] \subseteq [T'_2]\), where each of \( T'_1, T'_2 \) is either an enumeration type name or a type constant. This check is based on recorded information about inclusion of the sets defined by type constants and about which constants are members of these sets.

If the algorithm returns true then \([T_1]_{D_1} \subseteq [T_2]_{D_2}\). If it returns false and \( D_1 \) has no nullable symbols (i.e. \([T]_{D_1} \neq \emptyset\) for each type name \( T \) in \( D_1 \)) then \([T_1]_{D_1} \not\subseteq [T_2]_{D_2}\).

**Example 14.** Consider the Type Definitions from the Example 12:

\[ D = \{ A \rightarrow l[B[C]], B \rightarrow l[A'^+], C \rightarrow m[l]\} \]

and \( D' = \{ A' \rightarrow l[A'^+][C'], C' \rightarrow m[l'^+]\} \). To check whether \([A]_{D} \subseteq [A']_{D'}\), first we construct set \( C(A, A'), (B, A'), (C, C')\). Then the second part of the algorithm checks if \( L(l|m) \subseteq L(l'|m)\), \( L(l'^+|m) \subseteq L(l^+|m)\) and \( L(e) \subseteq L(m'^+)\). Since all the checks give positive results, we conclude that \([A]_{D} \subseteq [A']_{D'}\).

Notice that for a proper \( D_2 \) and 1-unambiguous regular expressions \([8]\) in \( D_1, D_2 \) the algorithm is polynomial. In the general case a polynomial algorithm does not exist, as inclusion for a less general formalism of tree automata is \( \text{EXPTIME-complete} \).[18]

### 3.3 Type Definitions and XML Schema Languages

For defining sets of XML documents we have introduced a simple and concise formalism of Type Definitions. This section discusses what features of particular XML schema languages are expressible by the Type Definitions and which are not.

The main task of schema languages is to describe XML documents. However different approaches to that task provide a wide range of functionality. What is common for most XML schema languages is that the schemata defined by them are transformations which given an instance document can produce a PSVI (Post Schema Validation Infoset) that besides the information from the original document includes default values, types, etc. In the thesis we focus on one aspect of XML schema languages, namely defining classes of documents (types). This implies that we neglect the other aspects like e.g. an ability to
describe default attribute values or to specify processing instructions (notations in DTD).

Our formalism of Type Definitions is focused on defining possible tree structure of XML documents and leaves out the aspect related to defining specific types of text nodes. Thus we do not discuss here in detail the simple types which are available in XML schema languages. We believe that our type system is flexible enough so that simple types can be implemented based on type constants. In the current version of the type system there is one constant type defined, namely Text, and it corresponds to a set of all strings (text values). However it is possible to define other type constants corresponding to simple types from DTD, XML Schema or Relax NG like e.g. NMTOKEN, ID, integer, etc. In this case some additional mechanism must be developed for validation according to these types or for ensuring uniqueness of the corresponding values. This is however out of scope of this work.

### 3.3.1 DTD

Any set of documents which can be defined by DTD can be also defined by a proper Type Definition. From the view point of formal language theory DTD is a local tree grammar (in the sense of [32]). DTDs are less expressive than Type Definitions as they cannot define two different sets of elements with the same label e.g. one set containing elements with a label title as a title of a book and another set of elements with a label title as a title of a chapter.

A Type Definition representing a DTD contains a definition of a type for each element declared in the DTD. The type names are the same as the corresponding element names in the DTD. Declarations of entities and notations in the DTD are neglected. As the DTD cannot define two different sets of elements with the same label the corresponding Type Definition is proper.

**Example 15.** This is an example of a DTD:

```xml
<!ELEMENT bib (book*)>
<!ELEMENT book (title, (author+ | editor+ ))>
<!ATTLIST book year CDATA #REQUIRED >
<!ATTLIST book isbn CDATA #IMPLIED >
<!ATTLIST book language (en | sw | pl) >
<!ELEMENT author (last, first )>
<!ELEMENT editor (last, first )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
```

and a corresponding Type Definition:
3.3.2 XML Schema

Although XML Schema in most cases corresponds to a single type tree grammar \cite{32}, sometimes it fails to satisfy its requirements e.g. because of the xsi:type mechanism (which is explained later on). Thus generally XML Schema can be represented as a proper Type Definitions but it has some features which cause problems. In this subsection we discuss the details of XML schema whose transformation to type definitions may be not clear, problematic or impossible.

We start with a simple example of an XML Schema with typical constructs.

Example 16. This is a fragment of XML Schema defining a set of elements named book:

```xml
<element name="book">
    <complexType>
        <sequence>
            <element name="title" type="string"/>
            <choice minOccurs="1" maxOccurs="unbounded">
                <element name="author" type="string">
                    <element name="editor" type="string">
                </choice>
            </sequence>
        </complexType>
    </element>
</element>
```

and a proper Type Definition defining a corresponding type Book:

\[
\begin{align*}
    \text{Book} & \rightarrow \text{book} \left[ \text{Book_attr Title (Author | Editor)}^+ \right] \\
    \text{Book_attr} & \rightarrow \text{attr} \left[ \text{Book_isbn} \right] \\
    \text{Book_isbn} & \rightarrow \text{isbn} \left[ \text{Text} \right] \\
    \text{Author} & \rightarrow \text{author} \left[ \text{Text} \right] \\
    \text{Editor} & \rightarrow \text{editor} \left[ \text{Text} \right] \\
    \text{Title} & \rightarrow \text{title} \left[ \text{Text} \right]
\end{align*}
\]
3.3. TYPE DEFINITIONS AND XML SCHEMA LANGUAGES

Below we present XML Schema features whose representation by Type Definition is not obvious.

1. For defining types being sets of elements XML Schema uses element definitions. Additionally XML schema allows to use type definitions to define sets of sequences of elements and attributes. Such types defined outside element definitions can be later used in different element definitions or as a basis for type derivation. For example, we can define a type \textit{Book} and then use it in definition of an element \textit{book}:

   \begin{verbatim}
   <complexType ="Book">
   <sequence>
     <element name="title" type="string"/>
     <choice minOccurs="1" maxOccurs="unbounded">
       <element name="author" type="string"/>
       <element name="editor" type="string"/>
     </choice>
   </sequence>
   <attribute name="isbn" type="string"/>
   </complexType>

   <element name="book" type="Book"/>
   \end{verbatim}

In the formalism of Type Definitions the rules defining types correspond to element definitions in XML Schema. Thus, when we transform such a schema into a Type Definition we ignore definitions of types and consider only definitions of elements. Before transforming a schema into a Type Definition we perform a kind of normalization on the schema i.e. we replace the references to types in element definitions by corresponding definitions of types and then remove the definitions of types which are not parts of element definitions. Such operation applied to the schema above results in the schema and the Type Definition from the Example 16.

2. XML Schema provides a type called \textit{anyType} which is the most general type from which all simple and complex types are derived. \textit{anyType} can be seen as a set of all XML documents. It is possible to use \textit{anyType} like other types. We do not provide such a type within Type Definition formalism. However we present a way of extending Type Definition with a type \textit{Top} which is an equivalent of \textit{anyType}.

3. A construct \textit{all} in XML Schema is used to specify the set of children of an element when their order is irrelevant. More precisely, all permutations of child elements are valid, as in XML child elements always occur in some order. Representation of such content models by regular type expressions often requires to list explicitly all the possible permutations. Although the number of all such permutations is finite, it may be so big that listing all the possibilities may be unfeasible. Note, that we cannot define such
CHAPTER 3. TYPE SPECIFICATION

a content model with a multiplicity list as multiplicity lists are used to specify unordered data terms e.g. $a[\{b\}, c\}]$ cannot be of the type defined by a rule $A \rightarrow a\{B, C\}$.

4. Type derivation by restriction of a complex type is a declaration that the derived type is a subset of the base type. When a new type is derived, its full content model must be specified such that the new type is a logical restriction of the base type. Although such a declaration of type inclusion is useful for some applications it has no practical meaning for a type system where type inclusion checking is performed based on a content model (and not on a type inclusion declaration).

5. Type derivation by extension of a complex type is a way to define a new type based on a type already defined. The content model of the new type is a sequence with the content model of the base type followed by the new content model. This is virtually equivalent to defining a new type from scratch by just explicit declaration of the whole content model. Again, although type extension mechanism may provide important information for some applications, from the point of view of our type system it can be seen just as syntactic sugar.

6. Element and attribute groups are containers in which sets of elements and attributes may be embedded and then they can be referenced in multiple definitions and declarations. Again, this can be seen as a syntactic sugar as any schema with element and attribute groups may be easily rewritten as an equivalent schema (defining the same class of XML documents) without them.

7. Substitution group is a mechanism that allows elements to be substituted for other elements. More precisely, elements can be assigned to a special group of elements that are said to be substitutable for a particular named element called the head element. Elements in a substitution group must have the same type as the head element, or they can have a type that has been derived from the head element’s type. For instance, consider a definition of an element article containing elements title, author, and comment. The element comment is defined as a head element of a substitution group and elements authorComment and reviewerComment are assigned to the substitution group.

```xml
<element name="article">
  <complexType>
    <sequence>
      <element name="title" type="string"/>
      <element name="author" type="string"/>
      <element ref="comment"/>
    </sequence>
  </complexType>
</element>
```
<element/>
<element name="comment" type="string"/>
<element name="authorComment" type="string">
    substitutionGroup="comment"/
</element>
<element name="reviewerComment" type="string">
    substitutionGroup="comment"/
</element>

The declaration implies that elements authorComment and reviewerComment can be substituted for an element comment in the instance document. Such a declaration can be expressed by the following Type Definition:

\[
\begin{align*}
    Article & \rightarrow \text{article} [ \text{Title} \ \text{Author} \ (\text{Comment} \ | \\
    & \quad | \ \text{aComment} \ | \ \text{rComment})] \\
    \text{Comment} & \rightarrow \text{comment} [ \text{Text} ] \\
    \text{aComment} & \rightarrow \text{authorComment} [ \text{Text} ] \\
    \text{rComment} & \rightarrow \text{reviewerComment} [ \text{Text} ]
\end{align*}
\]

8. Abstract elements and xsi:type. XML Schema provides a mechanism to force substitution for a particular element or type. When an element is declared to be abstract, it cannot be used in an instance document and only a member of the element’s substitution group can appear in the instance document. When an element’s type is declared as abstract, all instances of that element must contain the attribute xsi:type indicating a derived type that is not abstract. Because of the mechanism called xsi:type an XML Schema may not satisfy the constraints of single type tree grammar. For instance, consider a schema defining an abstract type Book, which is then used to derive by restriction the new types Book1 and Book2. Furthermore, the schema contains a definition of an element book whose content is of the abstract type Book:

<complexType name="Book" abstract="true">
    <sequence>
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="author" type="string"/>
            <element name="editor" type="string"/>
        </choice>
        <element name="title" type="string"/>
    </sequence>
</complexType>

<complexType name="Book1">
    <complexContent>
        <restriction base="Book">
            <sequence>
            </sequence>
        </restriction>
    </complexContent>
</complexType>
An instance document of this schema contains elements book with a content matching the content model of Book1 or Book2. Additionally, it is required that every instance element book contains information about the type of its content e.g.

```
<library>
  <book xsi:type="Book1">...</book>
  <book xsi:type="Book2">...</book>
  ...
</library>
```

The abovementioned schema can be expressed by the following non proper
Type Definition:

\[
\begin{align*}
Library & \rightarrow \text{book} [(\text{Book}_1 | \text{Book}_2)^*] \\
\text{Book}_1 & \rightarrow \text{book} [\text{Book}_1\_\text{attr} \ \text{Author}^* \ \text{Title}] \\
\text{Book}_2 & \rightarrow \text{book} [\text{Book}_2\_\text{attr} \ \text{Editor}^* \ \text{Title}] \\
\text{Book}_1\_\text{attr} & \rightarrow \text{attr} \{\ \text{Type}_1\} \\
\text{Book}_2\_\text{attr} & \rightarrow \text{attr} \{\ \text{Type}_2\} \\
\text{Author} & \rightarrow \text{author} [\ \text{Text}] \\
\text{Editor} & \rightarrow \text{editor} [\ \text{Text}] \\
\text{Type}_1 & \rightarrow \text{xsi}:\text{type} \{\ \text{Text}_1\} \\
\text{Type}_2 & \rightarrow \text{xsi}:\text{type} \{\ \text{Text}_2\} \\
\text{Text}_1 & \rightarrow "\text{Book}_1" \\
\text{Text}_2 & \rightarrow "\text{Book}_2"
\end{align*}
\]

Such a type specification can be approximated by a proper Type Definition which allows the document to contain elements of the abstract type \text{Book}. Then, the rule for \text{Library} could be:

\[
\begin{align*}
\text{Library} & \rightarrow \text{book} [\ \text{Book}^*] \\
\text{Book} & \rightarrow \text{book} [\ \text{Book}\_\text{attr} \ (\text{Author} | \text{Editor})^* \ \text{Title}] \\
\text{Book}\_\text{attr} & \rightarrow \text{attr} \{\ \text{Type}\} \\
\text{Type} & \rightarrow \text{xsi}:\text{type} \{\ \text{Text}_1 | \text{Text}_2\} \\
\text{Text}_1 & \rightarrow "\text{Book}_1" \\
\text{Text}_2 & \rightarrow "\text{Book}_2" \\
\text{Author} & \rightarrow \text{author} [\ \text{Text}] \\
\text{Editor} & \rightarrow \text{editor} [\ \text{Text}]
\end{align*}
\]

Note that the type \text{Library} defined in this way is a superset of data terms defined by the schema as it allows the library to contain books with both authors and editors.

### 3.3.3 Relax NG

The formalism of Type Definitions and Relax NG schema language are close to each other as both are based on the production rules from the regular tree grammars. However Relax NG has some significant extensions comparing to proper Type Definitions. We will discuss the most important ones.

Before the presentation of more advanced features of Relax NG consider a simple Relax NG schema

\[
\begin{align*}
\text{First} & = \text{element first } \{\ \text{text} \} \\
\text{Second} & = \text{element second } \{\ \text{text} \} \\
\text{Author} & = \text{element author } \{\ \text{First}, \ \text{Second} \} \\
\text{Editor} & = \text{element editor } \{\ \text{First}, \ \text{Second} \}
\end{align*}
\]

\[
\text{Book} = \text{element book } \{} 
\]
attribute isbn { text },
  element title { text },
(Author+ | Editor+)
}

The schema can be expressed by the following proper Type Definition.

\[
\begin{align*}
\text{Book} & \rightarrow \text{book} [ \text{Book_attr Title (Author+ | Editor+)} ] \\
\text{Book_attr} & \rightarrow \text{attr} \{ \text{Book_isbn} \} \\
\text{Book_isbn} & \rightarrow \text{isbn} [ \text{Text} ] \\
\text{Author} & \rightarrow \text{author} [ \text{First Second} ] \\
\text{Editor} & \rightarrow \text{editor} [ \text{First Second} ] \\
\text{Title} & \rightarrow \text{title} [ \text{Text} ] \\
\text{First} & \rightarrow \text{first} [ \text{Text} ] \\
\text{Second} & \rightarrow \text{second} [ \text{Text} ]
\end{align*}
\]

A content model of Relax NG can constrain both attributes and elements. It is illustrated by the next example. It presents a modified definition of the element \text{book} containing an attribute \text{isbn} and a list of authors or an attribute \text{publisher} and a list of editors:

\[
\begin{align*}
\text{Book} & = \text{element book} \{ \\
& (\text{attribute isbn} \{ \text{text} \}, \\
& \text{element title} \{ \text{text} \}, \\
& \text{Author}+) \}
\end{align*}
\]

\[
\begin{align*}
& | \\
& (\text{attribute publisher} \{ \text{text} \}, \\
& \text{element title} \{ \text{text} \}, \\
& \text{Editor}+) \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{First} & = \text{element first} \{ \text{text} \} \\
\text{Second} & = \text{element second} \{ \text{text} \} \\
\text{Author} & = \text{element author} \{ \text{First, Second} \} \\
\text{Editor} & = \text{element editor} \{ \text{First, Second} \}
\end{align*}
\]

Such schema can be represented by a non proper Type Definition:

\[
\begin{align*}
\text{Book} & \rightarrow \text{book} [(\text{Book_attr1 Title Author+}) | \\
& (\text{Book_attr2 Title Editor+})] \\
\text{Book_attr1} & \rightarrow \text{attr} \{ \text{Book_isbn} \} \\
\text{Book_isbn} & \rightarrow \text{isbn} [ \text{Text} ] \\
\text{Book_attr2} & \rightarrow \text{attr} \{ \text{Book_publisher} \} \\
\text{Book_publisher} & \rightarrow \text{publisher} [ \text{Text} ] \\
\text{Author} & \rightarrow \text{author} [ \text{First Second} ] \\
\text{Editor} & \rightarrow \text{editor} [ \text{First Second} ] \\
\text{Title} & \rightarrow \text{title} [ \text{Text} ] \\
\text{First} & \rightarrow \text{first} [ \text{Text} ] \\
\text{Second} & \rightarrow \text{second} [ \text{Text} ]
\end{align*}
\]
Another interesting feature of Relax NG are co-occurrence constraints. They allow to choose different content model depending on a value of an element or attribute. For example, we can define bibliography containing entries which have different structure depending on the value of the attribute type:

\[
\text{bibliography} = \text{element bibliography}\{
  \text{element entry}\{
    \text{attribute type } \{"article"\}
    \text{element author } \{\text{text}\}
    \text{element title } \{\text{text}\}
    \text{element journal } \{\text{text}\}
  \},*,
  \text{element entry}\{
    \text{attribute type } \{"inProceedings"\}
    \text{element author } \{\text{text}\}
    \text{element title } \{\text{text}\}
    \text{element bookTitle } \{\text{text}\}
  \}*
}\}
\]

Such a schema can be represented by a non proper Type Definition:

\[
\begin{align*}
\text{Bibliography} & \rightarrow \text{bibliography} \left[ \text{Entry}_1^* \text{Entry}_2^* \right] \\
\text{Entry}_1 & \rightarrow \text{entry} \left[ \text{Entry}_1\_\text{attr} \text{Author Title Journal} \right] \\
\text{Entry}_1\_\text{attr} & \rightarrow \text{attr} \left[ \text{Entry}_1\_\text{type} \right] \\
\text{Entry}_1\_\text{type} & \rightarrow \text{type} \left[ \text{Article} \right] \\
\text{Article} & \rightarrow \"\text{article}\" \\
\text{Entry}_2 & \rightarrow \text{entry} \left[ \text{Entry}_2\_\text{attr} \text{Author Title BookTitle} \right] \\
\text{Entry}_2\_\text{attr} & \rightarrow \text{attr} \left[ \text{Entry}_2\_\text{type} \right] \\
\text{Entry}_2\_\text{type} & \rightarrow \text{type} \left[ \text{Proceedings} \right] \\
\text{Proceedings} & \rightarrow \"\text{inProceedings}\" \\
\text{Title} & \rightarrow \text{title} \left[ \text{Text} \right] \\
\text{Author} & \rightarrow \text{author} \left[ \text{Text} \right] \\
\text{BookTitle} & \rightarrow \text{bookTitle} \left[ \text{Text} \right]
\end{align*}
\]

The above Relax NG schema can be expressed by an equivalent schema with only one definition for the element entry. A new content model for the element entry would be a disjunction of the content models from the two definitions for entry. However, a corresponding Type Definition would again be non proper.

Another thing that cannot be used in straightforward way in Type Definitions is the \text{interleave} operator &. The operator can be used in a content model additionally to the standard operators from regular expressions. The operator is used for unordered patterns. The following example defines a book element containing elements title, author, publisher which may appear in any order. It can also contain an element subtitle, which can occur only after the element title.
Title = element title { text }
Subtitle = element subtitle { text }
Author = element author { text }
Publisher = element publisher { text }
Book = element book {
    (Title, Subtitle?) &
    Author &
    Publisher}

We may say that the content model of the book element is partially ordered. In literature about regular languages the operator & is known as shuffle operator. According to [29] the operation shuffle applied to two regular languages is a regular language. Thus, a content model with the operator & can be expressed by a regular expression without &. A regular language \( L \) which is a result of shuffling two other regular languages \( L_1, L_2 \) can be obtained in the following way. Let \( A_1, A_2 \) be DFA representing \( L_1, L_2 \), respectively. The language \( L \) is represented by an automaton \( A \), which for each pair of states \( s_i, s'_j \) from \( A_1, A_2 \), respectively, has a corresponding state \( s_{ij} \). Provided that \( s_0 \) and \( s'_0 \) are the initial states of \( A_1, A_2 \), respectively, the initial state of \( A \) is \( s_{00} \). For each pair of final states: \( s_i \) in \( A_1 \) and \( s'_j \) in \( A_2 \), the final state in \( A \) is \( s_{ij} \). \( A \) has a transition with a label \( l \) from a state \( s_{ij} \) to a state \( s_{i'j} \) only if there is a transition in \( A_1 \) with a label \( l \) from a state \( s_i \) to \( s_{i'} \). Similarly, \( A \) has a transition in with a label \( l \) from a state \( s_{ij} \) to a state \( s_{ij'} \) only if there is a transition in \( A_2 \) with a label \( l \) from a state \( s'_j \) to \( s'_{j'} \).

Relax NG allows to specify a content model of an element without providing its name. For example, the schema

\[
\text{Book} = \text{element book} \{
\quad \text{element title} \{ \text{text} \},
\quad \text{element *} \{ \text{text} \}
\}
\]

defines an element \textit{book} containing a \textit{title} and an arbitrary element with a text content. Such a schema cannot be expressed by a Type Definition.
Chapter 4

Reasoning about Types of Xcerpt Query Results

This chapter presents a mechanism of type inference for results of non recursive Xcerpt programs. First, the inference mechanism is presented on an abstract level using typing rules. The typing rules are introduced inductively based on the syntax of Xcerpt. Based on the typing rules we present a formal proof of soundness of the type system. Afterwards a practical algorithm for type inference is presented and a method of checking correctness of Xcerpt programs is described.

4.1 Motivation

This section presents purposes the type system may serve. The main goal of the type system is to infer a type of a program results given types of databases being queried. Then, if the specification of the expected type of program results is given, it can be automatically checked whether the inferred type conforms to the specified type. Thus, the main two purposes the type system can serve are:

- **Type inference.** An approximation of a program result type can be computed given a type of a database. This information can be used as follows:
  - The user can check manually if the inferred result type conforms to his/her expectations. As a part of type inference for a program, types of variables occurring in query rules are computed. The information about types of variables can help to find errors in the program as the variables may be not of the types intended by a user.
  - If the inferred result type is empty it means that the program will never give any results. This may be a suggestion of an error caused by fact that the query terms in a body of a query rule do not match
the type of the database i.e. the query terms do not match any data term of that type. An algorithm for checking type emptiness was presented in Section 3.2.1.

- The inferred program result type can be used for documentation of the program.

**Checking type correctness.** Given a specification of the result type for a program it can be checked whether the inferred result type is included in the specified one. If such inclusion check succeeds the user can be sure that the program is correct with respect to the result type specification. As the presented type system does not infer precise types for Xcerpt programs results (but only approximations) the fact that the inferred result type is not included in the specified one is not a proof of a type error. In the case of type inclusion check failure a user can be only informed about possibility of a type error. However, for some restricted form of Xcerpt programs and Type Definitions the inferred result type is precise enough for a typechecking failure to be a proof of an unquestionable type error.

### 4.2 Variable-type Mappings

This section presents auxiliary definitions used later on in this chapter. In what follows we assume a fixed Type Definition \( D \) (describing the type of the database).

To represent a set of answers (for a query and a set of data terms) we will use a mapping \( \Gamma: V \rightarrow \mathcal{E} \) (called a variable-type mapping), where \( V \) is a set of variables (sometimes denoted \( \text{dom}(\Gamma) \)) and \( \mathcal{E} \) is a set of expressions. \( \mathcal{E} \) contains \( 0, 1 \), the type names from \( D \), and expressions of the form \( T_1 \cap T_2 \), where \( T_1, T_2 \in \mathcal{E} \). Each expression \( E \) from \( \mathcal{E} \) denotes a set \( [E] \) of data terms. \( [1] \) denotes the set of all data terms, \( [0] = \emptyset \), \( [T] = [T]_D \) for any type name \( T \), and \( [T_1 \cap T_2] = [T_1] \cap [T_2] \). The set of answer substitutions corresponding to a mapping \( \Gamma: V \rightarrow \mathcal{E} \) is

\[
\text{substitutions}_D(\Gamma) = \{ \theta \mid \forall X \in V \theta X \in [\Gamma(X)] \}.
\]

(According to our convention, we will often skip the index \( D \).) Notice that if \( \theta \in \text{substitutions}(\Gamma) \) then \( V \subseteq \text{dom}(\theta) \) and if \( \theta \subseteq \theta' \) then \( \theta' \in \text{substitutions}(\Gamma) \). For a set \( \Psi \) of variable-type mappings we define \( \text{substitutions}(\Psi) = \bigcup_{\Gamma \in \Psi} \text{substitutions}(\Gamma) \).

We define \( \bot, \top : V \rightarrow \mathcal{E} \) by \( \bot(X) = 0 \) and \( \top(X) = 1 \) for every \( X \in V \). For \( Y_1, \ldots, Y_k \in V, T_1, \ldots, T_k \in \mathcal{E} \), mapping \( [Y_1 \mapsto T_1, \ldots, Y_k \mapsto T_k] : V \rightarrow \mathcal{E} \) is defined as

\[
[Y_1 \mapsto T_1, \ldots, Y_k \mapsto T_k](X) = \begin{cases} T_i & \text{if } X = Y_i \\ 1 & \text{otherwise.} \end{cases}
\]

We will not distinguish between expressions \( T \cap 1 \) and \( T \), and between \( T \cap 0 \) and \( 0 \) (where \( T \in \mathcal{E} \)).
For any $\Gamma_1, \Gamma_2 : V \rightarrow \mathcal{E}$ we introduce $\Gamma_1 \cap \Gamma_2 : V \rightarrow \mathcal{E}$ such that 

$$(\Gamma_1 \cap \Gamma_2)(X) = \Gamma_1(X) \cap \Gamma_2(X).$$

Notice that $\Gamma \cap \bot = \bot$ and $\Gamma \cap \top = \Gamma$ for any $\Gamma : V \rightarrow \mathcal{E}$.

Inclusion of types induces a pre-order $\sqsubseteq$ on the mappings from $V \rightarrow \mathcal{E}$, as follows. If $\Gamma$ and $\Gamma'$ are such mappings then $\Gamma \sqsubseteq \Gamma'$ iff $[[\Gamma(X)]] \subseteq [[\Gamma'(X)]]$ for each variable $X \in V$. Notice that $\Gamma \sqsubseteq \Gamma'$ is equivalent to $\text{substitutions}(\Gamma) \subseteq \text{substitutions}(\Gamma')$, provided that $[[\Gamma'(X)]] \neq \emptyset$ for each $X \in V$.

For a particular query there may be many possible assignments of types for variables. That is why we will use sets of mappings from $V \rightarrow \mathcal{E}$. For such sets $\Psi_1$ and $\Psi_2$ we define:

$$\Psi_1 \cap \Psi_2 = \{\Gamma_1 \cap \Gamma_2 \mid \Gamma_1 \in \Psi_1, \Gamma_2 \in \Psi_2\},$$

$$\Psi_1 \cup \Psi_2 = \Psi_1 \cup \Psi_2,$$

Hence $\Psi \cap \{\bot\} = \{\bot\}$, $\Psi \cap \{\top\} = \Psi$, for any set of mappings $\Psi$. We will not distinguish between $\Psi \cup \{\bot\}$ and $\Psi$, and between $\Psi \cup \{\top\}$ and $\{\top\}$.

### 4.3 Typing Rules for Xcerpt

The rules presented in this section provide a descriptive type system for Xcerpt: the typing of a program is an approximation of its semantics. Based on the assumption that a type of each database queried by the program is given, the typing rules provide a way to infer a program result type. An algorithm computing a type of results for a given non recursive Xcerpt program can be easily derived from the presented rules as they can be seen as an abstract version of the algorithm. Below we present inductively typing rules for the syntactic constructs of Xcerpt: query terms, queries, construct terms, query rules and programs.

The typing rules presented here (except the rule (PROGRAM)) were introduced earlier in a joint work [5]. However, there are some minor changes in the rules presented here with respect to the rules in [5].

#### 4.3.1 Query terms

The rules in this subsection provide a way to derive variable-type mappings for a query term given a type of database to which the query term is applied. They can be used to derive facts of the form $D \vdash q : T \vdash \Gamma$, where $D$ is a Type Definition, $q$ a query term, $T$ a type name, and $\Gamma$ a variable-type mapping. The intention is that if $q$ is applied to a data term $d \in [T]$ then the resulting answer substitution is in $\text{substitutions}(\Gamma)$ for some $\Gamma$ such that $D \vdash q : T \vdash \Gamma$ can be derived.

$$b \in [T]$$

$$\frac{D \vdash b : T \vdash \Gamma}{D \vdash b : T \vdash \Gamma} \quad \text{(CONST)}$$
where \( b \) is a basic constant.

Thus, for a query term being a basic constant any variable-type mapping can be derived.

\[
\Gamma \subseteq [X \mapsto T] \\
D \vdash X : T \triangleright \Gamma
\]  

(\text{VAR})

Thus, application of a query term being a variable \( X \) to a type \( T \) results in a variable-type mapping which binds \( X \) to some \( T' \) such that \([T']_D \subseteq [T]_D\).

\[
D \vdash q : T \triangleright \Gamma \\
D \vdash X \mapsto q : T \triangleright \Gamma
\]  

(\text{AS})

\[
D \vdash q : T \triangleright \Gamma \\
D \vdash \text{desc } q : T \triangleright \Gamma
\]  

(\text{DESCENDANT})

\[
D \vdash \text{desc } q : T' \triangleright \Gamma \\
D \vdash \text{desc } q : T \triangleright \Gamma
\]  

(\text{DESCENDANT REC})

where \( T' \in \text{types}(r) \) and \( r \) is the content model of \( T \).

\[
D \vdash q_1 : T_1 \triangleright \Gamma , \ldots , D \vdash q_n : T_n \triangleright \Gamma \\
D \vdash l \alpha q_1, \ldots , q_n \beta : T \triangleright \Gamma
\]  

(PATTERN)

where \( l \) is the rule for \( T \) in \( D \) is of the form \( T \rightarrow l[r] \)

or it is of the form \( T \rightarrow l\{r\} \) and \( (\alpha \beta = \{\} \text{ or } \alpha \beta = \{\{\})\),

\( s \) is \( r \) with every type name \( U \) replaced by \( U|\epsilon \),

\( T_1 \cdots T_n \in L(r) \) if \( \alpha \beta = \{\} \),

\( T_1 \cdots T_n \in L(s) \) if \( \alpha \beta = \{\{\})\),

\( T_1 \cdots T_n \in \text{perm}(L(r)) \) if \( \alpha \beta = \{\} \),

\( T_1 \cdots T_n \in \text{perm}(L(s)) \) if \( \alpha \beta = \{\{\})\).

Here \( \text{perm}(L) \) stands for the language of permutations of the strings from a language \( L \).

We explain the fact that given a query term \( q = l o q_1, \ldots , q_n \beta \), the typing rule (\text{PATTERN}) produces the same variable-type mapping \( \Gamma \) as the mappings produced by typing rules applied to the query terms \( q_1, \ldots , q_n \). (A similar fact will hold for the typing rule (\text{AND QUERY}) in the next subsection.) Obviously, typing rules may produce different \( \Gamma_i \) for each \( q_i \) \( (i = 1, \ldots , n) \). However, (due to the rules (\text{VAR}) and (\text{AS})) they can produce also any "smaller" mapping \( \Gamma'_i \) for each \( q_i \) i.e. \( \Gamma'_i \subseteq \Gamma_i \). In particular, each \( \Gamma'_i \) may be \( \Gamma'_i = \Gamma_1 \cap \ldots \cap \Gamma_n = \Gamma \).
4.3.2 Queries

The rules in this subsection provide a way to derive variable-type mappings for a query given types of data terms to which the query is applied. In general, a query may be applied to data terms produced by query rules of an Xcerpt program. As their results may be of different types, we consider here a set of type names \( U \) instead of a single type \( T \). We assume that the types of external databases being queried by \( Q \) are given by a mapping \( \text{type}(db) \). The mapping associates each database \( db \) occurring in \( Q \) with a type \( T \) defined by the Type Definition \( D \). If \( \text{type}(db) = T \) then \( d(db) \in \{ [T] \} \).

From the rules below one can derive facts of the form \( D \vdash Q : U \bowtie \Gamma \), where \( Q \) is a query, \( U \) a finite set of type names and \( \Gamma \) a variable-type mapping. If \( \theta \) is an answer substitution for \( Q \) and a data term from \( \{ [U] \} \) then \( \theta \in \text{substitutions}(\Gamma) \) for some \( \Gamma \) such that \( D \vdash q : T \bowtie \Gamma \) can be derived.

\[
\begin{align*}
D \vdash q : T \bowtie \Gamma & \quad T \in U \\
\frac{D \vdash q : U \bowtie \Gamma}{D \vdash \text{in}(db, q) : U \bowtie \Gamma} & \quad (\text{QUERY TERM})
\end{align*}
\]

where \( \text{type}(db) = T \)

\[
\begin{align*}
D \vdash Q_1 : U \bowtie \Gamma & \quad \cdots \quad D \vdash Q_n : U \bowtie \Gamma \\
D \vdash \text{and}(Q_1, \ldots, Q_n) : U \bowtie \Gamma & \quad (\text{AND QUERY})
\end{align*}
\]

\[
\begin{align*}
D \vdash Q : U \bowtie \Gamma \\
D \vdash \text{or}(\ldots, Q, \ldots) : U \bowtie \Gamma & \quad (\text{OR QUERY})
\end{align*}
\]

4.3.3 Construct terms

The rules for construct terms use the variable-type mappings inferred by the rules for queries to define the result type of a query rule. To formulate typing rules for construct terms we need an equivalence relation on mappings:

**Definition 20.** Given a Type Definition \( D \), a set of variable-type mappings \( \Psi \) and a set \( V \) of variables, such that \( V \subseteq \text{dom}(\Gamma) \) and \( \text{substitutions}(\Gamma) \neq \emptyset \) for each \( \Gamma \in \Psi \), the relation \( \sim_V \subseteq \Psi \times \Psi \) is defined as: \( \Gamma_1 \sim_V \Gamma_2 \) iff \( \Gamma_1(X) \cap \Gamma_2(X) \neq \emptyset \) for all \( X \in V \). The set of equivalence classes of the transitive closure \( \hat{\sim}_V \) of \( \sim_V \) is denoted by \( \Psi/\hat{\sim}_V \).

The following rules allow to derive facts of the form \( D \vdash c : \Psi \bowtie s \), where \( c \) is a construct term, \( \Psi \) is a set of variable-type mappings (for which the types are defined by \( D \)) and \( s \) is a regular type expression. The intention is that if applying a substitution set \( \Theta \) to \( c \) results in a data term sequence \( \Theta(c) = d_1, \ldots, d_n \) and \( \Theta \subseteq \text{substitutions}(\Psi) \) then \( D \vdash c : \Psi \bowtie s \) can be derived such that each \( d_i \in \{ [T_i] \} \).
and \(T_1 \cdots T_n \in L(s)\). To derive \(D \vdash c : \Psi \triangleright s\) it is necessary that \(\Gamma(X) \neq 1\) for any \(\Gamma \in \Psi\) and any variable \(X\) occurring in \(c\). For correctness of the rules it is required that for any \(\Gamma \in \Psi\), substitutions(\(\Gamma\)) \(\neq \emptyset\) and for any \(\Gamma_1, \Gamma_2 \in \Psi\), \(\Gamma_1 \sim_{FV(c)} \Gamma_2\).

\[
\frac{(T_b \to b) \in D}{D \vdash b : \Psi \triangleright T_b} \quad \text{(CONST)}
\]

where \(b\) is a basic constant.

\[
\frac{[T_1] = [\Gamma_1(X)] \cdots [T_n] = [\Gamma_n(X)]}{D \vdash X : \{\Gamma_1, \ldots, \Gamma_n\} \triangleright T_1 \cdots \triangleright T_n} \quad \text{(VAR)}
\]

where \(D\) is weakly proper, \(T_1, \ldots, T_n\) are type names

Notice, that the rule (VAR) requires that a Type Definition \(D\) defines types \(T_1, \ldots, T_n\) such that \([T_i] = [\Gamma_i(X)]\). In particular, this means that if \(\Gamma_i(X)\) is not a type name i.e. it is an expression of the form \(A_1 \cap \ldots A_{ik_i}\), \(T_i\) is a type name representing an intersection of types \(A_1, \ldots, A_{ik_i}\). However, if \(D\) is not weakly proper, it may be impossible to define a type being the intersection of given types. For such cases an application of the rule (VAR) is impossible and some approximations must be done. This is expressed by the typing rule (VAR APPROX).

\[
\frac{[T_1] \supseteq [\Gamma_1(X)] \cdots [T_n] \supseteq [\Gamma_n(X)]}{D \vdash X : \{\Gamma_1, \ldots, \Gamma_n\} \triangleright T_1 \cdots \triangleright T_n} \quad \text{(VAR APPROX)}
\]

where \(D\) is not weakly proper, \(T_1, \ldots, T_n\) are type names

\[
\frac{D \vdash c_1 : \Psi \triangleright s_1 \quad \ldots \quad D \vdash c_n : \Psi \triangleright s_n \quad (T_c \to l \alpha s_1 \cdots s_n \beta) \in D}{D \vdash l \alpha c_1, \ldots, c_n \beta : \Psi \triangleright T_c} \quad \text{(PATTERN)}
\]

\[
D \vdash c : \Psi_1 \triangleright s_1 \quad \ldots \quad D \vdash c : \Psi_n \triangleright s_n \quad \{\Psi_1, \ldots, \Psi_n\} = \Psi / z_{FV(c)} \quad \text{(ALL)}
\]

\[
\frac{D \vdash c : \Psi_1 \triangleright s_1 \quad \ldots \quad D \vdash c : \Psi_n \triangleright s_n \quad \{\Psi_1, \ldots, \Psi_n\} = \Psi / z_{FV(c)} \quad \text{(SOME)}}
\]

Note, that for construct terms not being of the form \text{some} \(k\) \(c\) and \text{all} \(c\) the derived facts are of the form \(D \vdash c_n : \Psi \triangleright T_1 \cdots \triangleright T_n\), where \(T_1, \ldots, T_n\) are type names.
4.3. TYPING RULES FOR XCEP RT

We need to explain the fact that all the presented typing rules assume the same Type Definition \( D \) to be given. The typing rules are to be used to infer a result type for query rules which is not yet known and it cannot be defined by \( D \) which is assumed to be known from the very beginning. Thus, in practical usage of the typing rules the Type Definition \( D \) must be constantly updated by adding definitions of newly constructed types. It will result in a new Type Definition \( D' \supseteq D \). The intention is that the facts derived by the typing rules will hold for the extended Type Definition \( D' \).

4.3.4 Xcerpt query rules

For a given Type Definition \( D \), query \( Q \) and a set \( U \) of types names, the rules introduced above nondeterministically generate variable-type mappings. Now we describe which sets of generated mappings are sufficient for the purpose of approximating the semantics of query rules.

**Definition 21.** Let \( D \) be a Type Definition. Let \( Q \) be a query term and \( W \) a type name, or \( Q \) a query and \( W \) a set of type names. A set \( \{ \Gamma_1, \ldots, \Gamma_n \} \) of variable-type mappings is complete for \( Q \) and \( W \) wrt. \( D \) if

- \( D \vdash Q : W \triangleright \Gamma_i \) for \( i = 1, \ldots, n \), and
- if \( D \vdash Q : W \triangleright \Gamma \) and \( \text{substitutions}(\Gamma) \neq \emptyset \), then there exists \( i \in \{1, \ldots, n\} \) such that \( \Gamma \sqsubseteq \Gamma_i \).

Let \( Q \) be a query and \( W \) a set of type names or \( Q \) a query term and \( W \) a type name from \( D \). Here we explain a way a complete set of variable-type mappings for \( Q \) and \( W \) can be obtained.

Consider a derivation tree \([34]\) for a fact \( D \vdash Q : W \triangleright \Gamma \). The non leaf nodes of the tree are labelled by quadruplets \( D \vdash Q' : W' \triangleright \Gamma \), where \( Q' \) is a query and \( W' \) is a set of type names or \( Q' \) is a query term and \( W' \) is a type name from \( D \). Leafs of the tree can be labelled by expressions of the form \( \Gamma \sqsubseteq [X \mapsto T] \). The derivation tree for \( D \vdash Q : W \triangleright \Gamma \) has for each subquery \( Q' \) of \( Q \):

- exactly one node labelled \( D \vdash Q' : W' \triangleright \Gamma \) (for some \( W' \)), if \( Q' \) is not of the form \( \text{desc} \ q \)
- at least one node labelled \( D \vdash Q' : W' \triangleright \Gamma \) (for some \( W' \)), if \( Q' \) is of the form \( \text{desc} \ q \)

Let us construct a derivation tree for \( D \vdash Q : W \triangleright \Gamma \). As \( \Gamma \) will be computed at the end we start the construction from a root labelled \( D \vdash Q : W \triangleright \Gamma \). From the conditions in the typing rules it follows that for each newly constructed node labelled \( D \vdash Q' : W' \triangleright \Gamma \) there is a finite number of possibilities of choosing its children. We require that any label cannot occur twice on a path of a tree. (Otherwise, a label \( D \vdash \text{desc} \ q : W' \triangleright \Gamma \) (for some \( q \) and \( W' \)) could occur more than once on a path.) In this way we discard loops which are unproductive. When the tree is constructed we compute \( \Gamma \) as follows. Let \( \Gamma = [X_1 \mapsto S_1, \ldots, X_n \mapsto S_n] \),
where \( X_1, \ldots, X_n \) are the variables occurring in the leafs of the tree, and each \( S_i \) is of the form \( T_{i_1} \cap \cdots \cap T_{i_m} \), and \( T_{ij} \) occurs in \( S_i \) iff the condition \( \Gamma \subseteq [X_i \mapsto T_{ij}] \) occurs in the tree. Let \( n_Q \) be the number of subqueries of \( Q \), \( n_{desc} \) be the number of the subqueries of the form \( \text{desc} q \), and \( n_T \) be the number of type names defined by \( D \). A number of non leaf nodes of a tree constructed in this way is not greater then \( n_Q + n_{desc} * n_T \). As there is a finite number of possibilities of choosing the set of children of each node, the set \( \Lambda \) of trees which can be constructed in this way is finite (for given \( Q \) and \( W \)).

Consider an arbitrary derivation tree \( \lambda \) for \( D \vdash Q : W \triangleright \Gamma \). If we remove from it iteratively parts of paths of the form \( D \vdash \text{desc} q : W' \triangleright \Gamma, \ldots, D \vdash \text{desc} q : W' \triangleright \Gamma \) we will obtain a tree which is isomorphic to some tree \( \lambda' \in \Lambda \). Moreover, for each node in \( \lambda \) labelled \( D \vdash Q' : W' \triangleright \Gamma' \) the corresponding node in \( \lambda' \) is labelled \( D \vdash Q' : W' \triangleright \Gamma' \). Additionally, \( \Gamma \subseteq \Gamma' \), as \( \Gamma' \) is the most general variable type mapping satisfying the conditions of the tree \( \lambda \). Thus, the set of variable-type mappings corresponding to the trees from \( \Lambda \) is complete for \( Q \) and \( W \). The following rule will be used to infer a type of query rule results. It allows to derive facts of the form \( \Gamma \vdash c \leftarrow Q : U \triangleright s_1 | \cdots | s_n \) where \( c \leftarrow Q \) is a query rule, \( U \) is a finite set of type names and \( s_i \) are regular type expressions. The intention is that if we apply a query rule \( c \leftarrow Q \) to a database of a type \( [U] \) then we obtain results belonging to the set \( [s_1 | \cdots | s_n] \).

\[
\frac{D \vdash c : \Psi_1 \triangleright s_1 \cdots D \vdash c : \Psi_n \triangleright s_n \quad \{\Psi_1, \ldots, \Psi_n\} = \Psi/_{Z_{FV}(c)}}{D \vdash (c \leftarrow Q) : U \triangleright s_1 | \cdots | s_n}
\]

(QURY RULE)

where \( \Psi \) is complete for \( Q \) and \( U \) wrt. \( D \) for each \( \Gamma \in \Psi \), substitutions(\( \Gamma \)) \( \neq \emptyset \).

Note, that as a construct term \( c \) cannot be of the form \textit{some} \( k \) \( c' \) and \textit{all} \( c' \) the derived facts are of the form \( D \vdash (c \leftarrow Q) : U \triangleright T_1 | \cdots | T_m \), where \( T_1, \ldots, T_m \) are type names.

**Example 17.** Consider a Type Definition \( D = \{ T \rightarrow l[A^* B C], A \rightarrow "a" , B \rightarrow "b" , C \rightarrow "c", R_1 \rightarrow a|A^+ A], R_2 \rightarrow a|A^+ B], R_3 \rightarrow a|(A|B)^+ C] \} \) and the query rule

\[
a[\text{all} X, Y] \leftarrow l[[X, Y]]
\]

abbreviated as \( c_0 \leftarrow g \). We apply the query rule to a set of types \( U = \{T, A, B, C\} \). First we need to find a complete set of mappings \( \Psi_0 \) for \( q \) and \( U \). If we apply the query term \( q \) to the type \( T \) using the rules for query terms we can derive facts \( D \vdash q : T \triangleright \Gamma_i \) for \( i = 1, \ldots, 4 \), where \( \Gamma_1 = [X \mapsto A, Y \mapsto A], \Gamma_2 = [X \mapsto A, Y \mapsto B], \Gamma_3 = [X \mapsto A, Y \mapsto C] \) and \( \Gamma_4 = [X \mapsto B, Y \mapsto C] \). If we apply the query term \( q \) to the type \( A, B \) or \( C \) we cannot derive anything using the rules. Hence, the rules for queries allow us to derive \( D \vdash q : U \triangleright \Gamma_i \)
for $i = 1, \ldots, 4$. The set $\Psi_0 = \{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4\}$ is complete for $q$ and $U$. Since $FV(c_0) = \{Y\}$, $\Psi_0 \setminus FV(c_0) = \{\Psi_1, \Psi_2, \Psi_3\}$, where $\Psi_1 = \{\Gamma_1\}$, $\Psi_2 = \{\Gamma_2\}$, $\Psi_3 = \{\Gamma_3, \Gamma_4\}$. Now we apply each of $\Psi_i$ to the construct term $c_0$. Using the rules for construct terms we can derive the following facts: $D \vdash c_0 : \Psi_1 \triangleright R_1$, $D \vdash c_0 : \Psi_2 \triangleright R_2$ and $D \vdash c_0 : \Psi_3 \triangleright R_3$. Using the rule (QUERY RULE) we can derive $D \vdash c_0 \leftarrow q : U \triangleright R_1 | R_2 | R_3$. It means that if the rule $c_0 \leftarrow q$ is applied to a set of data terms from $[U]$ all the obtained results are in the set $[R_1 | R_2 | R_3]$.

The following theorem expresses the correctness of the typing rules wrt. the semantics given in section 2.1. More precisely, it expresses the existence of a typing derivation for a rule whenever it has a result for some set of data terms $Z$ of the type denoted by a set $U$ of type names. It also expresses that any type derived for a query rule $(c \leftarrow Q)$ wrt. a set of type names $U$ is a correct approximation of the set of results for $(c \leftarrow Q)$ and any set of data terms of the type denoted by $U$.

Theorem 1. Let $D$ be a Type Definition and $(c \leftarrow Q)$ be a query rule, where for each targeted query term $\exists n(db, q)$ in $Q$ there is a type name $T = \text{type}(db)$ defined in $D$. Let $U$ be a set of type names and $Z$ a set of data terms such that $Z \subseteq [U]$.

If a result for $(c \leftarrow Q)$ and $Z$ exists then there exist $s$ and $D'$ such that $D' \supseteq D$ and $D' \vdash (c \leftarrow Q) : U \triangleright s$.

If $D \vdash (c \leftarrow Q) : U \triangleright s$ and $d'$ is a result for $(c \leftarrow Q)$ and $Z$, then $d' \in [s]$.

Proof. See the Appendix.

4.3.5 Xcerpt programs

Now we present a rule allowing to infer a result type of a query rule in the context of a program. From the following rule one can derive facts of the form $P, D \vdash p \triangleright S$ where $P = (P, G)$ is a non recursive Xcerpt program, $p \in P$ is a query rule and $s$ is a regular type expression. The intention is that given a program $P$ the query rule $p$ produces results belonging to the set $[[s]]$.

$$P, D \vdash p \triangleright (T_{11} \mid \ldots \mid T_{1k_1}) \ldots P, D \vdash p_n \triangleright (T_{n1} \mid \ldots \mid T_{nk_n}) D \vdash p : U \triangleright s$$

$$P, D \vdash p \triangleright s \quad \text{(PROGRAM)}$$

where $p_1, \ldots, p_n$, ($n \geq 0$) are all the query rules on which the query rule $p$ directly depends,

$U = \{T_{11}, \ldots, T_{1k_1}, \ldots, T_{n1}, \ldots, T_{nk_n}\}$.

Let $G = \{p_1, \ldots, p_n\}$. A type of results of a program $P = (P, G)$ is a set $[[s_1] \mid \ldots \mid [s_n]]_D$ such that $P, D \vdash p_i \triangleright s_i$ for each $p_i \in G$.

The following theorem expresses the correctness of the typing rules wrt. the semantics given in section 2.1 in a context of a program. More precisely, it
expresses the existence of a typing derivation for a program whenever it has a result. It also expresses that the union of the derived types for the goals of a program is a correct approximation of the set of its results.

**Theorem 2.** Let \( \mathcal{P} = (P, G) \) be a non recursive Xcerpt program and \( D \) a Type Definition, where for each targeted query term \( \text{in}(db, q) \) in \( P \) there is a type name \( T = \text{type}(db) \) defined in \( D \).

If a result of \( \mathcal{P} \) exists then there exist \( p \in G, s \) and \( D' \) such that \( D' \supseteq D \), \( \mathcal{P}, D' \vdash p \triangleright s \).

If \( d \) is a result of \( \mathcal{P} \) then there exist \( p \in G \) such that if \( \mathcal{P}, D \vdash p \triangleright s \) then \( d \in \llbracket s \rrbracket_D \).

**Proof.** See the Appendix.

### 4.3.6 Exactness of the Typing Rules

#### Query terms

The set of variable type mappings \( \Psi \) produced by the typing rules for query terms expresses a superset \( \Theta \) of the set of possible answers for a query term \( q \). If \( q \) does not contain restricted variables (i.e. a construct \( \sim \)) then the set \( \Theta \) is the exact set of answers.

**Proposition 3.** Let \( D \) be a Type Definition without nullable type names, and whose content models do not contain useless symbols. Let \( q \) be a query term, \( T \) a type name from \( D \), and \( \Theta = \{ \theta \mid D \vdash q : T \triangleright \Gamma, \theta \in \text{substitutions}_D(\Gamma) \} \). If \( q \) does not contain \( \sim \) then each \( \theta \in \Theta \) is an answer for \( q \) and some \( d \in \llbracket T \rrbracket_D \).

**Proof.** See the Appendix.

**Example 18.** Consider a Type Definition \( D = \{ T \rightarrow l[A], A \rightarrow a[B^*], B \rightarrow "b" \} \) and a query term \( q_1 = l[X \sim a[]] \). The typing rules for query terms allow to infer the fact \( D \vdash q_1 : T \triangleright \Gamma \), where \( \Gamma = [X \mapsto A] \). The set \( \text{substitutions}_D(\Gamma) \) is not the exact set of answers for the query term \( q_1 \) and a database of type \( T \) i.e. it contains substitutions which cannot be answers for \( q_1 \) e.g. \( \theta = \{X/a["b"]\} \). In contrast, the same mapping \( \Gamma \) can be inferred for the query term \( q_2 = l[X] \) and \( \text{substitutions}_D(\Gamma) \) is the exact set of answers for \( q_2 \) and a database of type \( T \).

#### Queries

The set of variable type mappings \( \Psi \) produced by the typing rules for queries expresses a superset \( \Theta \) of the set of possible answers for a query \( Q \). If \( Q \) does not contain restricted variables (i.e. a construct \( \sim \)) and a construct and\((...)\) then the set \( \Theta \) is the exact set of answers.

**Proposition 4.** Let \( D \) be a Type Definition without nullable type names, and whose content models do not contain useless symbols. Let \( U \) be a set of type names from \( D \), \( Q \) be a query and \( \Theta = \{ \theta \mid D \vdash q : T \triangleright \Gamma, \theta \in \text{substitutions}_D(\Gamma) \} \).
Let $T_1, \ldots, T_n$ be type names in $D$ such that $\text{type}(db_i) = T_i$ for each targeted query term $\text{in}(db_i, q_i)$ in $Q$ ($i = 1, \ldots, n$). If $Q$ does not contain $\rightarrow$ and a construct $\text{and}(\ldots)$ then for each $\theta \in \Theta$ there exist

- data terms $d_1, \ldots, d_n$ of types $T_1, \ldots, T_n$, respectively,
- a data term $d \in [U]_D$ (if $U \neq \emptyset$)

such that $\theta$ is an answer for $Q$ and $d$ (or, if $U = \emptyset$, for $Q'$ and no data term), where $Q'$ is $Q$ with each targeted query term $\text{in}(db_i, q_i)$ replaced by a targeted query term $\text{in}(db'_i, q_i)$, such that $d(db'_i) = d_i$.

Proof. See the Appendix. \qed

Example 19. Consider a Type Definition $D = \{ T \rightarrow [T^2], A_1 \rightarrow a_1[C], A_2 \rightarrow a_2[C], B \rightarrow b[C \text{ Text}], C \rightarrow c[.], \}$, queries $Q_1 = \text{and}(l[X], X)$, $Q_2 = \text{and}(a_1[X], a_2[Y])$, $Q_3 = \text{and}(b[X, "a"], b[Y, "b"])$ and sets of type names $U_1 = \{ T \}$, $U_2 = \{ A_1, A_2 \}$, $U_3 = \{ B \}$. The typing rules allow as to derive the following mappings:

- $\Gamma_1 = [X \mapsto T]$ for $Q_1$ and $U_1$,
- $\Gamma_2 = [X \mapsto C, Y \mapsto C]$ for $Q_2$ and $U_2$,
- $\Gamma_3 = [X \mapsto C, Y \mapsto C]$ for $Q_3$ and $U_3$.

However, there is no answer substitution for each $Q_i$ and any data term $d_i \in U_i$. Thus, each $\Gamma_i$ is an approximation of the set of possible answers.

It is easy to eliminate one source of approximation of a derived variable type mapping for a query containing a construct $\text{and}(\ldots)$. We need to make a change in the typing rules for queries. Instead of deriving a mapping for a query $Q$ and a set of type names $U$ we can derive a mapping for $Q$ and a single type name $T \in U$. Such rules would be applied to each type name from $U$. Then, all the mappings for a query $Q$ and the set $U$ would be collected by the rule (QUERY RULE).

Construct Terms and Query Rules

This section presents sources of approximations which are related to construct terms. Then it summarizes all conditions for a query rule result type to be exact.

First, we define what we mean by an exact result type of a query rule. Let $D$ be a type definition, $U$ be a set of type names from $D$, $Q$ be a query, $T_1, \ldots, T_n$ be type names in $D$ such that $\text{type}(db_i) = T_i$ for each targeted query term $\text{in}(db_i, q_i)$ in $Q$. A type denoted $[R]_D$ is an exact result type for $c \leftarrow Q$ with respect to $U$ and $T_1, \ldots, T_n$ iff the following two conditions hold:

- any data term $d_r$, which is a result for $c \leftarrow Q$ and some set of data terms $Z \subseteq [U]$ belongs to $[R]_D$ i.e. $d_r \in [R]_D$
CHAPTER 4. REASONING ABOUT TYPES

• for each data term $d \in \{R\}_D$ there exist
  – data terms $d_1, \ldots, d_n$ of types $T_1, \ldots, T_n$, respectively,
  – a set of data terms $Z \subseteq \{U\}$

such that $d$ is an answer for $Q'$ and $Z$, where $Q'$ is $Q$ with each targeted
query term $\text{in}(d_b, q_i)$ replaced by a targeted query term $\text{in}(d'_b, q_i)$, such
that $d(d'_b) = d_i$.

The typing rules for construct terms introduce an approximation which is
related to the constructs all and some. Because of the constructs the typing
rules for construct terms cannot consider each variable type mapping separately.
The mappings must be grouped into equivalence classes which make a result type
of query rules not exact even if the query rule is without all and some.

Example 20. Let $T_1, T_2$ be types such that $\{T_1\} \cap \{T_2\} \neq \emptyset$. Let $\Gamma_1 = [X \mapsto T_1, Y \mapsto T_2], \Gamma_2 = [X \mapsto T_2, Y \mapsto T_1]$ and let $\Psi = \{\Gamma_1, \Gamma_2\}$ be a complete set of variable type mappings for some query $Q$ and some set of types $U$. Additionally, we assume that the set substitutions $D(\Psi)$ is an exact set of answers for $Q$.

Consider a query rule $c[X, Y] \leftarrow Q$. As $\Gamma_1, \Gamma_2$ belong to one equivalence class with respect to the free variables of $c[X, Y]$ the result type for $c[X, Y] \leftarrow Q$ produced by the typing rule (Query Rule) is $R$, which is defined as $R \rightarrow c[(T_1|T_2)(T_2|T_1)]$. This type is not exact set of possible results as it contains data terms which cannot be results for the query rule $c[X, Y] \leftarrow Q$ e.g. $c[t_1, t_2]$, where $t_1 \in \{T_1\}$, $t_2 \in \{T_1\} \backslash \{T_2\}$.

However, equivalence classes of variable-type mappings do not introduce any
approximations if for any two types defined by a Type Definition their intersec-
tion is empty. In such case, two mappings would be in the same equivalence class
only if the free variables are bound to the same types by each of the mappings.

Abandoning the constructs all and some would make it possible to create
a much simpler typing rule for query rules. Such a simpler typing rule does not
introduce any approximations:

\[
D \vdash c : \Psi \triangleright s \quad D \vdash Q : U \triangleright \Gamma \quad \Psi = \{\Gamma\}
\]

\[
D \vdash (c \leftarrow Q) : U \triangleright s
\]

(Query Rule 2)

where $c$ does not contain constructs all and some

Note, that the typing rule does not require a complete set of variable-type
mappings.

Consider the following example.

Example 21. Let $T_1, T_2, \Gamma_1, \Gamma_2, \Psi$ be defined as in the previous example. We
consider the same query rule $c[X, Y] \leftarrow Q$. To obtain the result type of the
query rule we apply the typing rule (Query Rule 2) twice: once for $\Gamma_1$ and
once for $\Gamma_2$. The first application of (Query Rule 2) results in the result type
4.3. TYING RULES FOR XCEPR

$R_1$ defined as $R_1 \rightarrow c[T_1T_2]$ and the second application results in the result type $R_2$ defined as $R_2 \rightarrow c[T_2T_1]$. The type of results $[R_1|R_2]$ is exact as it does not contain data terms which cannot be results of the query rule.

The second approach giving more precise results cannot be used for query rules containing constructs all or some. However, one may consider a combination of the two approaches and use typing rule (QUERY RULE2) for query rules not containing all and some and the typing rule (QUERY RULE) for query rules with all and some. This would result in more precise result types in some cases.

We have already mentioned another reason of inexactness of result types for query rules. The result type of a query rule $p$ may be inexact if a construct term in $p$ contains multiple occurrences of the same variable. In such case each occurrence of a the same variable in a construct term must be replaced with the same value. It is not sufficient for the values to be of the same type. Such a set of results may be not regular and may be unable to be represented by our formalism.

**Example 22.** Consider a query rule $c[X,X] \leftarrow l[X]$ which is applied to a data term of type $T$ defined by a Type Definition $D = \{ T \rightarrow l[A], A \rightarrow a[Text] \}$. The inferred result type for the query rule would be $R$ defined as $R \rightarrow c[Text Text]$. Although a data term $d = c["text1","text2"]$ belongs to $[R]$, $d$ cannot be a result for the query rule as it contains two different subterms.

Another issue that may lead to inexactness of an inferred result type is a non weakly proper Type Definition. For non weakly proper Type Definitions we use the construct term typing rule (VAR APPROX) which may cause the inferred result type to be not exact.

Now, we sum up the conditions needed for an exact type of query rule results. Let $D$ be a Type Definition, $U$ a set of type names, $c \leftarrow Q$ a query rule, and $T_1, \ldots, T_n$ types of databases queried by $Q$. Let $[R]_D$ be a type of results inferred with the typing rules (of the previous sections) for $c \leftarrow Q$ and types $U, T_1, \ldots, T_n$. The type $[R]_D$ is an exact type of results of $c \leftarrow Q$ wrt. $U, T_1, \ldots, T_n$ if

- the Type Definition $D$
  - is weakly proper,
  - does not contain content models with useless symbols or nullable type names,
  - does not define any two types with a non empty intersection,

- the query $Q$
  - does not contain a construct $\rightarrow$,
  - does not contain a construct and(...),
– does not contain constructs all and some,
– does not contain multiple occurrences of a variable.

In general an inferred result type for a program consisting of more than one query rule is not exact. It is so because we cannot guarantee that an inferred result type for a query rule is described by a weakly proper Type Definition.

4.4 Type Inference Algorithm

This section presents a practical algorithm for type inference for results of non-recursive Xcerpt programs. The algorithm is based on the typing rules presented in the previous section. An input to the algorithm is an Xcerpt program, a Type Definition specifying the types of resources (databases) being queried by the program and a mapping type(db) which associates each resource in the program with a type defined by the Type Definition. We start with a presentation of type inference algorithm for a single query rule and then we generalize it to programs with multiple query rules.

4.4.1 Type Inference for a Query Rule

Here we present a type inference algorithm for a single query rule, that is an implementation of the typing rules up to the rule (QUERY RULE). The algorithm computes the type of results of the query rule \( c \leftarrow Q \) which is applied to data terms of type \([U]\) (i.e. data terms produced by other query rules) and data terms from the resources specified in the query rule \( Q \) (i.e. from external databases). We assume that a set of types \( U \) is given as an input to the algorithm together with the types of resources occurring in \( Q \) (given by the mapping type(db) for each resource \( db_i \)). All these types are defined by a given Type Definition \( D \). A concrete algorithm consists of two main steps. First, a complete set of variable-type mappings \( \Psi \) for \( Q \) and \( U \) must be found. Then, based on \( \Psi \) types of query rule results are built.

**Computing a complete set of variable-type mappings**

Here we describe a method to compute a complete set of variable type mappings. The method, which is based on typing rules for queries, is implemented as a procedure mappingSet and presented later on in this section. First, we present a procedure match which describes a way of typing query terms, which are parts of queries. The procedure computes a complete set of variable type mappings for a given query term \( q \) and a given type \( T \).

For a type name \( T \) we define a set of reachable type names reachable\((T)\) in the following way. If \( T \) is not a type variable then reachable\((T) = \emptyset \). Let \( r \) be a content model of \( T \). A type name \( T' \in \text{reachable}(T) \) iff

- \( T' \in \text{types}(r) \), or
4.4. TYPE INFEERENCE ALGORITHM

- $T' \in \text{types}(r'')$, where $T'' \in \text{reachable}(T)$ is a type variable and $r''$ is a content model of $T''$.

Now we are ready to present the procedure \textit{match}.

\begin{verbatim}
match(q, T) :
  IF q is a variable X THEN
    return \{X ↦ T\}
  IF q is of the form X ∼ q' THEN
    return \{X ↦ T\} ∩ match(q', T)
  IF q is of the form desc q' THEN
    return match(q', T) \uplus \bigcup_{T' \in \text{reachable}(T)} \text{match}(q', T')
    (Now q is a rooted query term or a basic constant).
  IF root(q) \neq label(T) THEN return \emptyset
  IF T is a type constant or a special type name THEN
    IF q is a basic constant in \{T\} THEN return \{\top\} ELSE return \emptyset
  let q = q_1 \cdots q_n \beta (n ≥ 0),
  IF \{\} are the parentheses for T and (αβ = [ ] or αβ = [[]]) THEN
    return \emptyset
  let r be the regular type expression in the rule for T in D
  let s be r with every type name U replaced by U|ε
  let $L' = \begin{cases}
    L(r), & \text{if } αβ = [ ] \\
    L(s), & \text{if } αβ = \{\} \\
    \text{perm}(L(r)), & \text{if } αβ = [[]] \\
    \text{perm}(L(s)), & \text{if } αβ = \{\{\} \}
  \end{cases}$
  return \{Γ_1 ∩ \cdots ∩ Γ_n | T_1 \cdots T_n \in L',
      Γ_1 \in \text{match}(q_1, T_1), \ldots, Γ_n \in \text{match}(q_n, T_n) \}
\end{verbatim}

Here \text{perm}(L) stands for the language of permutations of the strings from a language $L$.

The procedure \textit{match} is inefficient in general. The crucial operation which determines its complexity is finding sequences of type names $T_1, \ldots, T_n$ belonging to a permutation of a regular language $L(r)$. In the worst case, there may be $m^n$ sequences of $T_1 \cdots T_n \in L(r)$ (where $m$ is a number of distinct type names occurring in $r$). In such a worst case the set of $m^n$ sequences already contains all the permutations of each sequence belonging to the set. Hence, the worst case time complexity for the operation is $O(m^n)$. Thus, for a practical usage of the algorithm some optimizations must be provided. Notice, that the elements of the sequence $T_1, \ldots, T_n$ are used to be matched\footnote{The expression 'a query term matches a type' is used informally here. It means that the query term matches some data term of the type.} with respective subterms of the query term $q = lαq_1 \cdots q_n \beta$. If a subterm $q_i$ is a rooted query term it can only match a type whose label is the same as the root of $q_i$. Usually, there will be at most one type name occurring in $r$ with a given label. Thus, it is possible to fix (or constrain) some type names in sequences $T_1 \cdots T_n \in L(r)$. This will decrease the number of cases to be considered. This optimization will not be
very helpful for query terms with a big number of unrooted query terms (e.g. variables) among \(q_1, \ldots, q_n\). In this case the practical usage of the algorithm may be impossible. However, it seems that for cases occurring in practice the optimized algorithm can be used effectively.

Note, that the complexity of \(\text{match}(q, T)\) increases also when the number of the children of \(q\) or the number of different types occurring in the content model of \(T\) grows.

**Example 23.** Consider a Type Definition \(D = \{T \rightarrow \text{\{Text\}}, T \rightarrow \text{\{\}\]}\) and a query term \(q = \text{\{X\} \rightarrow \text{\{\}\]}\). We execute \(\text{match}(q, T)\). In the first run of the procedure we obtain \(L' = \text{perm}(T_i, T_2)\). Thus the sequences of type names of the length two belonging to \(L'\) are \(T_1T_2, T_2T_1, T_2T_2\). Then for each such a sequence of type names we call \(\text{match}\) for relevant query terms and types:

- \(\text{match}(X \rightarrow \text{\{\}\]}, T_1)\) and receive \(\emptyset\)
- \(\text{match}(Y, T_1)\) and obtain \([Y \rightarrow T_1]\)
- \(\text{match}(X \rightarrow \text{\{\}\]}, T_2)\) and obtain \([X \rightarrow T_2]\)
- \(\text{match}(Y, T_2)\) and obtain \([Y \rightarrow T_2]\)

Now we consider only two sequences, namely \(T_2T_1\) and \(T_2T_2\), for which we get not empty sets of mappings. Thus as a result for \(\text{match}(q, T)\) we get a set of mappings \([X \rightarrow T_2, Y \rightarrow T_1], [X \rightarrow T_2, Y \rightarrow T_2]\). The received result is not exact. The mappings show that \(X\) may be bound to data terms of type \(T_2\). In fact \(X\) can be bound only to such data terms of \(T_2\) which have "s" inside.

Finally, we are ready to present the procedure \(\text{mappingSet}(Q, U)\) that returns a set of variable type mappings for a query \(Q\) a a set of type names \(U\). The procedure is an implementation of typing rules for queries. Moreover it expresses the way of derivation of complete set of variable-type mappings described in Section [4.3.4]. Thus, the set \(\Psi = \text{mappingSet}(Q, U)\) is a complete set of variable type mappings for \(Q\) and \(U\).

We assume that the types from \(U\) are defined by a Type Definition \(D\) as well as the types of resources occurring in \(Q\) which are given by a mapping \(\text{type}(db)\).

\[
\text{mappingSet}(Q, U) : \\
\text{IF } Q \text{ is of the form or}(Q_1, \ldots, Q_n) \text{ then return } \text{mappingSet}(Q_1, U) \cup \ldots \cup \text{mappingSet}(Q_n, U) \\
\text{IF } Q \text{ is of the form and}(Q_1, \ldots, Q_n) \text{ then return } \text{mappingSet}(Q_1, U) \cap \ldots \cap \text{mappingSet}(Q_n, U) \\
\text{IF } Q \text{ is of the form in}(db, q) \text{ then return } \text{match}(q, \text{type}(db)) \\
\text{IF } Q \text{ is a query term } q \text{ then return } \bigcup_{T \in U} \text{match}(q, T)
\]

The values of the mappings from \(\Psi = \text{mappingSet}(Q, U)\) may be expressions of the form \(T_1 \cap \ldots \cap T_n\), where each \(T_i\) is a type name. Consider the set \(W_\Psi\)
4.4. TYPE INFERENCE ALGORITHM

of all such expressions

\[ W_\Psi = \left\{ T_1 \cap \ldots \cap T_n \mid T_1 \cap \ldots \cap T_n = \Gamma(X), \ \Gamma \in \Psi, \ X \in V \right\}_{n > 1}, \]  
where each \( T_i \) is a type name.

For any expression \( E \in W_\Psi \), \([E]\) is the intersection of types defined by \( D \). Provided that \( D \) does not contain non weakly content models, using the algorithm from the Section 3.2.2 we can construct a Type Definition \( D_\Psi \) such that for each \( E \in W_M \) there exists a type variable \( T_E \) for which \([E]\n \in \Psi \). Moreover, \([T]_{D_\Psi} = [T]_D \) for all type variables occurring in \( D \) (hence for those occurring in \( \Psi \)). If \( D \) is proper then \( D_\Psi \) is proper. If \( D \) contains non weakly proper content models, first it must be approximated by a Type Definition without such models. In consequence the obtained set of variable-type mappings will approximate the previous set of mappings.

From the obtained set of mappings \( \Psi \) we remove all the mappings which bind variables to empty types. Such empty types may be results of type intersections. To determine if a type is empty we can use the algorithm from the Section 3.2.1. The set of mappings \( \Psi' \) obtained in this way is still a complete set of variable type mappings. Moreover, for each \( \Gamma \in \Psi' \), \( \text{substitutions}(\Gamma) \neq \emptyset \).

### Computing Type of Query Rule Results

Here we present a second step of type inference for a query rule \( c \leftarrow Q \) and a set of types \( U \). We assume that a complete set \( \Psi \) of variable type mappings for \( Q \) and \( U \) is given. Moreover, all variable type mappings from \( \Psi \) are of the form \( [X_1 \mapsto T_1, \ldots, X_n \mapsto T_n] \) where \( T_1, \ldots, T_n \) are not nullable type names defined in the Type Definition \( D \).

First we present a way to obtain a set of equivalence classes \( \Psi/_{\sim_V} \) given a set of variable type mappings \( \Psi = \{\Gamma_1, \ldots, \Gamma_n\} \) and a set of variables \( V \) (such that \( V \subseteq \text{dom}(\Gamma) \) for each \( \Gamma \in \Psi \)). We divide the set of mappings \( \Psi \) into one element sets \( \Psi_1, \ldots, \Psi_n \) such that each \( \Psi_i = \{\Gamma_i\} \). Now we join the sets of mappings under the following condition. Two sets \( \Psi_i \) and \( \Psi_j \) can be joined if there exists \( \Gamma' \in \Psi_i \) and \( \Gamma'' \in \Psi_j \) such that \( \Gamma' \sim_V \Gamma'' \) i.e. for each variable \( X \in V \), \( [\Gamma'(X)] \cap [\Gamma''(X)] \neq \emptyset \). To decide if an intersection of two types is empty algorithms for type intersection and type emptiness can be employed. We continue joining the sets of mappings inductively until no joint is possible. The obtained in this way set of sets of variable type mappings is the set of equivalence classes \( \Psi/_{\sim_V} \). Assume that the method is implemented as a function \( \text{eqClasses}(\Psi, V) \) returning the set \( \Psi/_{\sim_V} \). The complexity of the presented procedure is polynomial provided that checking whether the intersection of two types is empty, is also polynomial. This is not the case if the content models of the types to be intersected, are not 1-unambiguous regular expressions. In this case the complexity of the presented procedure is exponential.

The following procedure \( \text{buildType}(c, \Psi_i) \) returns a regular type expression \( r_{\Psi_i} \). The arguments of the function are a construct term \( c \), a set of variable type mappings \( \Psi_i \in \Psi/_{\sim_{FV(c)}} \) (where \( FV(c) \) stands for the set of the free variables...
CHAPTER 4. REASONING ABOUT TYPES

of c) and a Type Definition D. Let Ψ₁, ..., Ψₙ be the equivalence classes of Ψ. The function buildType(c, Ψᵢ) is called for each Ψᵢ (i = 1, ..., n). The union of the types produced by all the calls [rᵢ₁ | ··· | rᵢₙ] is a superset of the set of results of the query rule c ← Q.

During the execution of the procedure new types are being created and the type definition D is being extended with rules defining the new types. We assume that a procedure define(N → ...) adds a rule N → ... to the Type Definition, and that N is a new type name, not occurring elsewhere. In this way a Type Definition DΨᵢ is constructed such that D ⊆ DΨᵢ.

buildType(c, Ψ') :
IF c is a basic constant THEN
define(T_c → c)
return T_c
IF c is a variable THEN
let {Γ₁, ..., Γₙ} = Ψ'
let Tᵢ = Γᵢ(c) for i = 1, ..., n
return (T₁ | ··· | Tₙ)
IF c is of the form lα₁, ..., cₙβ THEN
let rᵢ = buildType(cᵢ, Ψ') for i = 1, ..., n
define(T_c → lα₁ r₁ ... rₙβ)
return T_c
IF c is of the form all c THEN
let {Ψ₁, ..., Ψₙ} = eqClasses(Ψ', FV(c))
let rᵢ = buildType(c, Ψᵢ) for i = 1, ..., n
return (r₁ | ··· | rₙ)
IF c is of the form some k c THEN
let {Ψ₁, ..., Ψₙ} = eqClasses(Ψ', FV(c))
let rᵢ = buildType(c, Ψᵢ) for i = 1, ..., n
return (r₁ | ··· | rₙ)⁺

Given a set of types U, a Type Definition D and a complete set of variable-type mappings Ψ, a set of results for the query rule c ← Q is a subset of

\[ R = \bigcup_{\Psi \in \text{eqClasses}(\Psi', FV(c))} \text{buildType}(c, \Psi) \]

The conditions for the set R to be the exact set of results are specified in Section 4.3.6.

For a construct term c being the head of a query rule the function buildType(c, Ψᵢ) returns a regular type expression of the form Tᵢ₁ | ··· | Tᵢₖ. The set of results R is the union of types T₁₁, ..., T₁ₖ₁, ..., Tₙ₁, ..., Tₙₖₙ produced by the calls buildType(c, Ψᵢ) (where \{Ψ₁, ..., Ψₙ\} = Ψ/ ≤ FV(c)). Each of the types Tᵢj is defined by a Type Definition DΨᵢ. However we can assume that the types Tᵢj are defined by one Type Definition DΨ which is a union of Type Definitions DΨᵢ for i = 1, ..., n (type name conflicts should be resolved by renaming type names). Thus, assuming that \( M = \{T_{11}, ..., T_{1k_1}, ..., T_{n1}, ..., T_{nk_n}\} \) the set of results R can be expressed as \([M]_{D\Psi}\).
In general, the newly constructed Type Definition $D_{\Psi}$ is not weakly proper. It may be impossible to describe $R$ by a weakly proper Type Definition. Instead one may consider constructing a weakly proper Type Definition defining a superset of the given set. This could be done by approximating non weakly proper content models with weakly proper ones. A description of a method of such approximation was presented in Section 3.2.2.

Example 24. Consider the query rule:

\[
\text{result}[\text{name}[\text{TITLE}], \text{author}[\text{ARTIST}]] \leftarrow \\
\text{in}("http://www.example.com/cds.xml", \\
\text{catalogue}[[\text{cd}[\text{title}[\text{TITLE}], \text{artist}[\text{ARTIST}]]]])
\]

We will abbreviate the query rule by $c \leftarrow Q$, and $Q$ by $\text{in}(\text{url}, q)$.

Assume that $\text{type}(\text{url}) = \text{Catalogue}$ defined by a Type Definition $D$:

\[
\begin{align*}
\text{Catalogue} & \rightarrow \text{catalogue}[\text{Cd}^*] \\
\text{Cd} & \rightarrow \text{cd}[\text{Title} \\text{Artist}^+] \\
\text{Title} & \rightarrow \text{title}[\text{Text}] \\
\text{Artist} & \rightarrow \text{artist}[\text{Text}]
\end{align*}
\]

We want to use the presented algorithm to infer the type of the results for the query rule. First, we call $\text{mappingSet}(Q, \emptyset)$ to obtain a complete set of variable-type mappings for $Q$ and $\emptyset$ (as we assume that the query rule queries only the external database). It executes $\text{match}(q, \text{Catalogue})$. As a result $\text{mappingSet}$ returns a set of mappings $\Psi = \{\Gamma_1, \Gamma_2\}$ where $\Gamma_1 = [\text{TITLE} \rightarrow \text{Artist}, \text{ARTIST} \rightarrow \text{Artist}], \Gamma_2 = [\text{TITLE} \rightarrow \text{Title}, \text{ARTIST} \rightarrow \text{Artist}]$. The set of equivalence classes of $\Psi$ is $\{\{\Gamma_1\}, \{\Gamma_2\}\}$. A call of $\text{buildType}(c, \{\Gamma_1\})$ results in extending the Type Definition $D$ with the following definitions of types:

\[
\begin{align*}
\text{Result}_1 & \rightarrow \text{result}[\text{Name}_1 \\text{Author}_1] \\
\text{Name}_1 & \rightarrow \text{name}[\text{Artist}] \\
\text{Author}_1 & \rightarrow \text{author}[\text{Artist}]
\end{align*}
\]

and a call of $\text{buildType}(c, \{\Gamma_2\})$ extends $D$ with

\[
\begin{align*}
\text{Result}_2 & \rightarrow \text{result}[\text{Name}_2 \\text{Author}_2] \\
\text{Name}_2 & \rightarrow \text{name}[\text{Title}] \\
\text{Author}_2 & \rightarrow \text{author}[\text{Artist}]
\end{align*}
\]

Thus, finally a result type of the query rule is $[\text{Result}_1 \mid \text{Result}_2]_{D'}$, where $D'$ is:

\[
\begin{align*}
\text{Result}_1 & \rightarrow \text{result}[\text{Name}_1 \\text{Author}] \\
\text{Result}_2 & \rightarrow \text{result}[\text{Name}_2 \\text{Author}] \\
\text{Name}_1 & \rightarrow \text{name}[\text{Artist}] \\
\text{Name}_2 & \rightarrow \text{name}[\text{Title}] \\
\text{Author} & \rightarrow \text{author}[\text{Artist}] \\
\text{Title} & \rightarrow \text{title}[\text{Text}] \\
\text{Artist} & \rightarrow \text{artist}[\text{Text}]
\end{align*}
\]
4.4.2 Type Inference for a Multiple Query Rule Program

Let $\mathcal{P} = (P, G)$ be a non recursive Xcerpt program. Let $\mathcal{P}_1, \ldots, \mathcal{P}_n$ be programs extracted from $\mathcal{P}$ and containing exactly one goal i.e. $\mathcal{P}_i = (P \setminus G \cup \{g_i\}, \{g_i\})$ for each $g_i \in G$. Then each $\mathcal{P}_i$ can be considered separately.

Let $H_i$ be a tree with the query rules from $\mathcal{P}_i$ being its nodes. Let $g_i$ be a root of $H_i$ and children of each node $p$ in $H_i$ be the query rules on which $p$ directly depends. Our objective is to infer the result type for the goal $g_i$. To do that we need to infer result types for the query rules starting from the bottom of the tree. To infer a result type of a rule we need a set of types $U$ as an input.

We start from the query rules being leafs of a tree and take an empty set $U$ as an input. Then we infer result types of query rules whose children’ result types were already inferred taking the sets of inferred children’ types as inputs. We perform this procedure until the result type $[r_i]$ of the goal $g_i$ is obtained. To obtain a type of results for a given program $\mathcal{P}$ we perform the procedure for each $\mathcal{P}_i$ where $i = 1, \ldots, n$. The result type of $\mathcal{P}$ can be represented as $[r_1] \cdots [r_n]$.

The presented method for type inference cannot be applied to recursive programs $(P, G)$. Direct application is not possible as a set of type names $U$ to which a query rule $p$ can be applied may be unknown. This is because $p$ may depend on itself and $U$ should already contain result type for $p$ which we want to infer.

One may think about the following modification of the method. Consider a program $(P, G)$. As the goals of the program do not depend on themselves first we consider only query rules which are not goals. Let $\{p_1, \ldots, p_n\} = P \setminus G$. The goal of the modified method would be to find a type $[U]$ such that each query rule $p_i$ applied to the type $[U]$ returns results of the type $[U]$ i.e $D \vdash p_i : U \rightarrow s_i$, such that $[s_i] \subseteq [U]$. We may try to build such set $[U]$ in the following way. We apply the typing rules for each $p_i$ ($i = 1, \ldots, n$) and an empty set $U_0$ (i.e $[U_0] = \emptyset$) to infer a result type $[s_{0i}]$. Based on the inferred types we construct a new set $[U_1] = [U_0] \cup [s_{10}] \cup \ldots \cup [s_{0n}]$. Then again we use typing rules to infer a result type for each $p_i$ and the type $[U_1]$. In general case $[U_{k+1}] = [U_k] \cup [s_{1k}] \cup \ldots \cup [s_{nk}]$. If $[U_{k+1}] = [U_k]$ for some $k$, $[U_k]$ is the result type of the rules $p_1, \ldots, p_n$ and it can be used to infer a result type for goals from $G$. In general case the condition $[U_{k+1}] = [U_k]$ may be never satisfied so the computation is infinite.

**Example 25.** Consider query rules $p = a[b[X]] \leftarrow \text{or}(a[X], \text{in}(db, a[X]))$ and $g = r[X] \leftarrow a[X]$. Let $\mathcal{P} = \{(p, g), \{g\}\}$ be a program and $\text{type}(db) = T$, where $T$ is defined as $T \rightarrow a[\text{Text}]$. To infer a program result type, first, we need to infer a result type for $p$. We apply the rule $p$ to an empty set of type names $U_0$ and obtain a result type $T_1$ defined as $T_1 \rightarrow a[B_1], B_1 \rightarrow b[\text{Text}]$. Thus $U_1 = \{T_1\}$. Then we apply the rule $p$ to $U_1$ and besides the type $T_1$ we obtain a type $T_2$ defined as $T_2 \rightarrow a[B_2], B_2 \rightarrow b[B_1]$. Thus $U_2 = \{T_1, T_2\}$. The operation is repeated until the inferred result type is the same as the result type obtained in the previous iteration. However, in this case, this condition will be never satisfied. The result type for $p$ can be expressed by an infinite sequence of type names $[T_0] \cdots [T_k] \cdots$, where $T_k$ is defined as $T_k \rightarrow a[B_k], B_k \rightarrow b[B_{k-1}]$. 
4.5 Type Correctness of Programs

Given a type of resources queried by a non-recursive program $P$, a type of its results $R = [T_1 | \cdots | T_n]$ can be inferred using the type inference algorithm. This type is given as a union of types $T_1, \cdots, T_n$. Given a specification of an expected program result type $T_E$ it can be checked whether the inferred type is included in $T_E$. The relation of type inclusion holds if each of the types $T_1, \ldots, T_n$ is included in $T_E$. This checking can be done using the algorithm for Type Inclusion. The algorithm requires that the type $T_E$ is specified by a proper Type Definition.

A positive result of such typechecking (i.e. checking whether each $T_i \subseteq T_E$) is a proof of program correctness with regard to the type specification. On the other hand a negative result can be a hint that the program results may not be of the intended type. Generally, a typechecking failure is not a proof of a type error because the inferred program result type is only an approximation of the set of program results. However, for some restricted form of Xcerpt programs and Type Definitions (described in Section 4.3.6), the inferred result type is the exact set of program results. In such case a typechecking failure is a proof of an unquestionable type error.

In practice very often the specification of the result type will be given by an XML schema which will need to be transformed into a Type Definition. As discussed in the Section 3.3 in most cases such a transformation results in a proper Type Definition. Otherwise the obtained Type Definition must be approximated as a proper Type Definition.

4.6 Possible Extensions

4.6.1 Type Correctness of Recursive Programs

Here we present a method for checking type correctness of recursive programs. The approach can be used to check type correctness of recursive programs with regard to type specification. Consider an Xcerpt program $P = (P, G)$ and a specification $S$ describing a set $[U]$ of allowed database terms and allowed results for any query rule $p \in P$. A sufficient condition for correctness of $P$ with regard to $S$ is that each rule of $P$ applied to data terms from $[U]$ produces results from the set $[U]$. This is an inductive proof method, similar to those used for partial correctness of programs. (For such a method for logic programs see [23] and references therein.) The sufficient condition can be checked by means of the type inference algorithm described earlier. For each rule of $P$ one can compute the type of its results. If the computed type of results is included in the type $[U]$ the program is correct with respect to the specification $S$. Otherwise, the program may be incorrect wrt. $S$. 
4.6.2 The Most General Type

A useful extension to the presented type system is introducing a type name \( \text{Top} \) for which \( \llbracket \text{Top} \rrbracket \) is the set of all data terms. Such extension would allow to infer types of program results without any information about types of databases being queried assuming that the databases are of the type \( \text{Top} \). This would also help to discover some errors even without knowledge about the types of the queried databases. For instance, it can be discovered that a query rule does not match the results produced by some other query rule (when such matching is expected).

Introducing the type \( \text{Top} \) requires modifications in some definitions related to Type Definitions. Furthermore it requires some simple extensions in the algorithms for basic operations on types.

As we cannot associate a label with the type \( \text{Top} \) we need to redefine proper content model. We say that a content model \( r \) is proper if the only type name occurring in \( r \) is \( \text{Top} \) or it does not contain \( \text{Top} \) and any two distinct type names having the same label.

In order to be able to express an intersection of types with a Type Definition we need a requirement that any multiplicity list containing the type name \( \text{Top} \) is \( \text{Top}^* \). It is not sufficient that a multiplicity list is of the form \( \text{Top}^{(l:m)} \).

For instance, a result of the intersection of a multiplicity list \( A \{1:2\} B \) with a multiplicity list \( \text{Top}^{(2:2)} \) cannot be expressed by a multiplicity list (it can be expressed by a regular expression \( AA | AB \)).

4.6.3 Correctness Checking Based on Types of Variables

There is another interesting way of checking correctness of \text{Xcerpt} query rules. Instead of inferring a type of results of an \text{Xcerpt} query rule we only infer types of variables i.e. variable-type mappings. Additionally, based on the specification of a type of expected results for the query rule we compute expected types of variables in a construct term which is the head of the rule. This can be done by treating a construct term as a query and using the algorithm for computation of a set o variable-type mappings for it. The algorithm must be however extended to be able to handle the constructs \text{all} and \text{some}. Finally, we check whether the inferred types of variables from the query are included in those computed expected types of variables from the construct term. Failure of such typechecking would suggest a type error. This method of checking type correctness of \text{Xcerpt} programs allows more precise location of errors as we know exactly which variable is not of the intended type. In some cases the method allows also to prove correctness of a program wrt. a specification of its result type even if such a proof is impossible for the general method. Consider the following example.

Example 26. Consider a Type Definition \( D = \{ R \rightarrow r[(AA)\{(BB)\}], T \rightarrow l[AB], A \rightarrow a[], B \rightarrow b[], \} \). Let the query rule \( r[XX] \leftarrow l[X] \) be applied to a database of the type \( \llbracket T \rrbracket_D \). Assume that a specified result type is \( \llbracket R \rrbracket_D \). First we infer variable type mappings for the query; they are \( \Gamma_1 = [X \mapsto A] \) and \( \Gamma_2 = [X \mapsto B] \).
Using the general method for type inference we infer a result type for the query rule which is \( R' \) defined as \( R' \rightarrow r[(A|B)(A|B)] \). As \([R'] \not\subseteq [R] \) there is a possibility of query rule incorrectness. Thus its correctness is not proved.

According to the method presented here (i.e. in the current section), besides the mappings \( \Gamma_1, \Gamma_2 \) we compute the expected types of variables based of the result type specification. They are given by the variable-type mappings: \( \Gamma'_1 = [X \mapsto A] \), \( \Gamma'_2 = [X \mapsto B] \). For each inferred variable type mapping \( \Gamma_i \) there is a variable type mapping \( \Gamma'_j \) obtained from the result type specification such that the types to which the variables are bound in \( \Gamma_i \) are included in the types to which the variables are bound in \( \Gamma'_j \) i.e. \( \text{substitutions}(\Gamma_i) \subseteq \text{substitutions}(\Gamma'_j) \). Thus, this is a proof of query rule correctness.
Chapter 5

The Prototype

This chapter presents a prototype of the type system (typechecker) implemented as a part of this thesis. Similarly to the prototypical runtime system for Xcerpt, it has been implemented in the functional language Haskell. It has been attached as a module to the Xcerpt prototype. The current version of the typechecker supports type specifications given with the formalisms of Type Definitions or DTDs.

The prototype is restricted to the fragment of Xcerpt covered by our type system (described in the Section 4.3). Moreover, it is restricted to Xcerpt programs consisting only of one query rule. It is still under development and the goal is to extend it towards the full Xcerpt.

The prototype of the type system together with the Xcerpt runtime system can be accessed online via the link [http://ida.liu.se/~artwi/XcerptT](http://ida.liu.se/~artwi/XcerptT). However the user interface for the online access to the prototype is still under development.

5.1 Usage of the Prototype

This section uses the notation where square brackets [ ] and the elements denoted by triangle parentheses <...> belong to a metalanguage.\[1\]

The type system is called like the standard Xcerpt runtime system (i.e. executing xcerpt or xcerpt.exe). To perform type checking (or type inference) of a program a parameter \(-t\) is used:

```
xcerpt -t <program file> [<type specification>]
```

The typing mechanism can also be called using the interactive Xcerpt command mode with the command:

```
:type <program file> [<type specification>]
```

\[1\] [ ] represents optional part and <...> is a nonterminal which can be replaced with a text without spaces.
In the abovementioned commands \textit{<program file>} is a restricted \textit{Xcerpt} program and \textit{<type specification>} is a text file specifying the types of resources\textsuperscript{2} which are queried and the types of expected results. \textit{<type specification>} file may contain:

- a Type Definition i.e. rules defining types,
- one or more input type specifications,
- one output type specification.

The input type specification has the syntax:

\begin{verbatim}
Input::
[ resource = <resource URI> ]
[ typedef = <typedef location> ]
type = <type name>
\end{verbatim}

and the output type specification has the syntax:

\begin{verbatim}
Output::
[ typedef = <typedef location> ]
type = <type name>
\end{verbatim}

where

- \textit{<resource URI>} is an URI of the resource being queried whose type we specify. If the parameter \texttt{resource} is omitted the input type specification specifies a type of every resource occurring in the \textit{<program file>} whose type was not specified (overridden) by other input type specification. A \textit{<type specification>} can contain at most one input type specification without the parameter \texttt{resource}.

- \textit{<typedef location>} is a URI of an external file containing a Type Definition (or DTD). If the \texttt{typedef} is omitted the input or output type specification refers to the local Type Definition i.e. specified in the current \textit{<type specification>} file.

- \textit{<type name>}, if used in an input type specification, is a type name specifying the type of the resource the specification refers to. If it is used in an output type specification it is a type name specifying the result type of the program. It can be the most general type \texttt{Top} or a type name which is defined in the Type Definition or the DTD the input or output type specification refers to. If the specification refers to a DTD then a type name can be one of the element names declared in the DTD.

A syntax of a Type Definition for the typechecker is slightly different than the one used in renaming part of this thesis. One difference is that we do not use here quotation marks in rules defining enumeration type names. The second difference is that we use a symbol \# instead of Text.

\textsuperscript{2}A resource corresponds to a database \texttt{db} in a targeted query term \texttt{in(db,q)}.  

Example 27. This is an example of a <type specification> file books.xts:

Books -> books[Book*]
Book -> book[Title Author+ Editor+]
Title -> title[#]
Author -> author[P]
Editor -> editor[P’]
P -> person[S]
P’ -> person[F? S?]
Person -> person[F+ S]
F -> firstname[#]
S -> surname[#]
Result -> result[Person+]

Input::
resource = file:books.xml
typename = Books

Output::
typpename = Result

Invoking the typing mechanism (e.g., with the command \texttt{xcerpt -t program file} <type specification>) starts the process of type inference for the query rule. The type inference is done using the knowledge of types of resources given by input type specifications. If the type of a resource is not specified by any input type specification it is assumed to be the most general type \texttt{Top} (which can be seen as a default type of a resource). After the type of results for the query rule has been inferred it is checked whether it is included in the corresponding output type (specified by the output type specification). If the corresponding output type is \texttt{Top} type checking is needless i.e. type inference is only performed. If the output type specification is missing the specified output type is assumed to be \texttt{Top}. Invoking the typing mechanism without <type specification> parameter has the same effect as invoking it with an empty <type specification> file.

As a result of typing an Xcerpt program we get a printout containing:

- information whether type checking or only type inference was performed with the result of type checking (if it was performed),
- the inferred result type,
- variable type mappings for variables occurring in the query rule,
- a Type Definition defining the inferred type and types of resources

For the types being intersection of other types their content model is provided by a DFA instead of a regular type expression. A DFA is presented by description.

\footnote{For any regular expression a DFA representing the same language may be constructed.}
tions of all its states. Each such a description is of the form $S_i \Rightarrow a_1 > S_{k_1}$, $\ldots a_n > S_{k_n}$, where $S_i$ is the number of the state being described, $a_1, \ldots, a_n$ are the symbols of the alphabet on which the DFA is defined and each $S_{k_j}$ is the number of the state reached from the state $S_i$ by reading the symbol $a_j$. Additionally, the number of the state being described may be preceded by the character '>' which denotes the initial state or it may be followed by the character '!' which denotes a final state. This is an example of a DFA corresponding to the language defined by a regular expression $AF^*$:

$$
0 \Rightarrow A>0 F>0 \\
1 > \Rightarrow A>2 F>0 \\
2! \Rightarrow A>0 F>2
$$

A name given by the system for a type being the intersection of types $T_1, T_2$ is $T_1 \cap T_2$. The type checker also invents type names for the newly inferred types. The devised new type names are the labels of the corresponding construct terms occurring in heads of query rules. If there is a need to define a type with a type name which has already been used the new type name is augmented with an index i.e. a number added at the end of the type name (underscore separated). If a type name with a given index already exists the new type name has the index increased by 1.

**Example 28.** Here we present a printout being a result of typing the following Xcerpt program:

```xcerpt
CONSTRUCT
  result [ all var X ]
FROM
  in {
    resource { "file:books.xml" },
    books {{
      book {{
        title [ var Y ],
        author [ var X ],
        editor [ var X ]
      }
    }
  } END

A type specification for the program is given by books.xts file from the previous example. The obtained printout is:

```
Type checking ... FAILED
```

Result type: result (not a subset of Result)
5.2 OVERALL STRUCTURE OF THE SOURCE CODE

Variable-type mappings:

Y -> #, X -> P^P'

Type Definition:

result -> result[P^P'+]  
Books -> books[Book*]  
Book -> book[(Title (Author+ Editor+))]  
Title -> title[( (# ))]  
Author -> author[P]  
Editor -> editor[P']  
P -> person[S]  
P' -> person[(F? S?)]  
Person -> person[(F+ S)]  
F -> firstname[#]  
S -> surname[#]  
Result -> result[Person+]  
P^P' -> person[
  0 => S>0
  >1 => S>2
  2! => S>0
]

5.2 Overall Structure of the Source Code

The source code of the runtime Xcerpt system together with a type system structured using Haskell’s hierarchical module mechanism is shown in the Figure 5.1. Most of the modules shown there are discussed in the description of Xcerpt prototype in [37]. Here, we present a short description of the parts related to the type system. With respect to the Xcerpt prototype two new submodules have been added to the implementation:

- Xcerpt.Typing implements the core part of the type system. It contains the files:
  - Type.hs containing an implementation of the Type Inference algorithm
  - TypeIncl.hs containing an implementation of the Type Inclusion algorithm
Figure 5.1: Overall module and file structure of the Xcerpt runtime system together with the type system; modules denoted by rectangles, files by rounded rectangles, added modules related to the type system in grey, modified modules or files in light grey.

- **TypeInter.hs** containing an implementation of the Type Intersection algorithm.

- **Xcerpt.RegExp** implements regular expressions and automata using a Haskell library[^40]. The library has been modified to support lists of strings instead of lists of characters. Additionally, the submodule contains the file **ProductDfa.hs** used for construction of product automata.

Additionally, some files has been added to the modules already existing in the Xcerpt prototype:

- **Xcerpt/Parser** has been extended with two files **TD.hs** and **DTD.hs** which are used to parse Type Definitions and DTDs.

- **Xcerpt/Data** has been extended with the files
5.2. OVERALL STRUCTURE OF THE SOURCE CODE

- **TypeDef.hs** containing data structures and definitions of basic operations for Type Definitions.
- **Mapping.hs** containing data structures and definitions of basic operations for variable type mappings.

- Xcerpt has been extended with the file **Helper.hs** containing basic helper functions.

Furthermore, some of the files in the already existing Xcerpt prototype has been modified. The files **Xcerpt.hs**, **XcerptInteractive.hs**, **XcerptCGI.hs** have been extended with options supporting the type system.
Chapter 6

Use Cases

This chapter presents examples of simple scenarios showing the way the presented type system can be helpful for programmers using Xcerpt for querying Web data. The chapter is divided into sections corresponding to the purposes the type system can serve. The purposes were described in Section 4.1 and we shortly recall them here:

- **Type inference.** An approximation of a program result type can be computed given a type of a database. These are the ways it can be helpful for a user:
  - A programmer can check manually if the inferred result type conforms to his/her expectations. He/she may also check if the inferred types of variables are as expected.
  - Emptiness of the inferred result type of a program may suggest an error as the program will never give any results.
  - An inferred program result type can be used for documentation of the program.

- **Checking type correctness.** Given a specification of a result type for a program it can be checked whether the inferred result type is included in the specified one. A success of such inclusion check is a proof of type correctness of the program. In general, a typechecking failure suggests a possibility of a type error. However, such a failure is a proof of an unquestionable type error for the restricted form of Xcerpt programs and Type Definitions described in Section 4.5.

All one query rule examples have been checked by our prototype and printouts are presented in Appendix B. The example of Section 6.1.3 uses several query rules. Our prototype is not yet operational for such program examples.
6.1 Type Inference

This section presents examples of Xcerpt programs showing a usefulness of the type inference mechanism. We assume here that there is no type specification of the expected program results given.

6.1.1 Manual Type Checking

Consider a Type Definition:

\[
\begin{align*}
Cds & \rightarrow \bib \[ Cd^+ \] \\
Cd & \rightarrow \cd \[ Title \ Arti st^+ \ Category^? \] \\
Title & \rightarrow \title \[ Text \] \\
Artist & \rightarrow \artist \[ Text \] \\
Category & \rightarrow "pop" | "rock" | "classic"
\end{align*}
\]

The query rule below queries a document \textit{cds.xml} of the type \textit{Cds} defined above. The intention of the query rule is to collect artists together with all their titles of the CD’s of a category ‘pop’.

\textbf{CONSTRUCT}
\begin{verbatim}
result [ 
  all entry [ 
    var ARTIST, 
    all var TITLE 
  ] 
] FROM 
in { resource [ "file:cds.xml" ], 
  bib {{ 
    cd [[ var TITLE, var ARTIST, "pop" ]] 
  }} 
END
\end{verbatim}

Printout 6.1.1 in the appendix is a result of typing the query rule by the type-checker. We assume that an intention of an author of the query rule is that the variable TITLE will be bound to data terms \textit{title}[…] and the variable ARTIST will be bound to data terms \textit{artist}[…]. The type system infers the types of variables used in the query rule. They are given by the variable-type mappings: \textit{[TITLE↔Title, ARTIST↔Artist]}, \textit{[TITLE↔Artist, ARTIST↔Artist]}. As the variable TITLE is intended (by the programmer) to take values only of the type \textit{Title}, the inferred types for variables suggest that the query rule is incorrect with regard to the programmer’s expectations.

Based on the inferred types of variables the query rule result type is inferred.
6.1. TYPE INFERENCE

The inferred result type is \(\text{Result}'\) defined as:

\[
\text{Result}' \quad \to \quad \text{result} \left[ \text{Entry}^{+} \right] \\
\text{Entry}' \quad \to \quad \text{entry} \left[ \text{Artist} \left( \text{Title} \mid \text{Artist} \right)^{+} \right] \\
\text{Artist} \quad \to \quad \text{artist} \left[ \text{Text} \right] \\
\text{Title} \quad \to \quad \text{title} \left[ \text{Text} \right] 
\]

Looking at this specification of the inferred type of program results, a programmer can also realize that the results of the program can be different than expected ones (as an entry should not contain more than one artist).

6.1.2 Empty Result Type

Consider the following Type Definition:

\[
\begin{align*}
\text{Bibliography} & \quad \to \quad \text{bib} \left( \text{Book} \mid \text{Article} \mid \text{InProceedings} \right)^{*} \] \\
\text{Book} & \quad \to \quad \text{book} \left\{ \text{Title Authors Editors Publisher} \right\} \\
\text{Article} & \quad \to \quad \text{article} \left\{ \text{Title Authors Journal} \right\} \\
\text{InProceedings} & \quad \to \quad \text{inproc} \left\{ \text{Title Authors Book} \right\} \\
\text{Title} & \quad \to \quad \text{title} \left[ \text{Text} \right] \\
\text{Authors} & \quad \to \quad \text{authors} \left[ \text{Person}^{*} \right] \\
\text{Editors} & \quad \to \quad \text{editors} \left[ \text{Person}^{*} \right] \\
\text{Publisher} & \quad \to \quad \text{publisher} \left[ \text{Text} \right] \\
\text{Journal} & \quad \to \quad \text{journal} \left\{ \text{Title Editors} \right\} \\
\text{Person} & \quad \to \quad \text{person} \left[ \text{FirstName LastName} \right] \\
\text{FirstName} & \quad \to \quad \text{first} \left[ \text{Text} \right] \\
\text{LastName} & \quad \to \quad \text{last} \left[ \text{Text} \right] 
\end{align*}
\]

The following programs query a document \(\text{bibliography.xml}\) of the type \(\text{Bibliography}\) defined above. Printout 6.1.2 is a result returned from the typechecker for the tree following query rules.

First, consider a query rule:

```
CONSTRUCT
  result []
  all var AUTHOR,
  titles [ all var TITLE ]
]
FROM
in { resource [ "file:bibliography.xml" ],
  Bib {{
    Book {{ Author [ var AUTHOR ], Title [ var TITLE ] }}
  }}
}
END
```

The query rule returns no results when it is applied to a database of type \(\text{Bibliography}\) because of the labels’ mismatch. The labels occurring in the body
of the rule are written with capital letters while labels occurring in the Type Definition are written with lower case letters. Thus, the query rule does not match the type of the database and the result type inferred for this query rule is empty.

This is another example of a query rule with an empty result type:

```construct
CONSTRUCT
results [
  all publisher [ var NAME , var URL ]
]
FROM
in { resource [ "file:bibliography.xml" ],
  bib {
    book {{ publisher [ name[ var NAME ], url[ var URL ] ]}}
  }
}
END
```

The inferred result type is empty due to the fact that the query term in the body of the query rule cannot be matched against data terms of type Bibliography. This is because the query looks for name[...], and url[...] as direct subterms of publisher[...] while data terms of type Publisher contain only text.

The next query rule does not match the database because of the square brackets used to match data terms book{...}. According to the type of database direct subterms of book{...} are unordered and cannot be matched with a query term being an ordered pattern.

```construct
CONSTRUCT
result [
  all var AUTHOR,
  titles [ all var TITLE ]
]
FROM
in { resource [ "file:bibliography.xml" ],
  bib [
    book [[ title [ var TITLE ], author [ var AUTHOR ] ]]
  ]
}
END
```

The inferred result type for the next query rule is also empty. This is caused by the wrong usage of the variable PERSON. Its first occurrence will be bound to data terms of the type Person while its second occurrence will be bound to direct subterms of a data term person[...] which can be of the type either FirstName or LastName. As the intersection of type Person with each of latter types is empty the inferred query result type is also empty.
6.1. TYPE INFERENCE

```sql
CONSTRUCT
  result [ all var AUTHOR,
    titles [ all var TITLE ]
  ] FROM
  in { resource ["file:bibliography.xml" ],
    bib {{
      book {{
        editors {{ var PERSON }}
      }},
      book {{
        authors {{
          person {{ var PERSON }}
        }}
      }}
    }}
  } END
```

6.1.3 Program Documentation

Here we present an example of an Xcerpt program being one of the use cases for Xcerpt which were presented in [37]. As no result type specification is given for the program the type system is only able to perform type inference. This results in the specification of the inferred program result type. Such a type specification provided by the inference mechanism can be used for documentation purposes. Additionally, it can be used by a programmer to check manually if the inferred result type conforms to his/her expectations.

The scenario of the program is in analogy to a use case in the XQuery Use Cases (XMP-Q5 in [15]). The program queries two online bookstores and provides a summary over the prices for books in both book stores. The summary is given using two representations: HTML representation and a representation suitable for mobile devices, in the WML format (wireless markup language\(^1\)). The program uses rule chaining to separate the query part from the presentation part and creates an intermediate representation for the data (in the example below: for each book, a `book-with-prices[...]` data term containing `title[...]`, `price-a[...]` and `price-b[...]` subterms for the price in the first bookstore and the price in the second bookstore). This 'simpler' representation is then queried by the two rules that create HTML and WML representations.

The schemata defining the structure of databases for the two bookstores are given in [37] using the Relax NG notation and can be expressed by the following Type Definition:

\(^1\)
[^http://www.wapforum.org/DTD/wml_1.1.xml]: http://www.wapforum.org/DTD/wml_1.1.xml
The type of the document bib.xml is Bib and the type of the document reviews.xml is Reviews. This is the Xcerpt program:

```
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 {
```
6.1. TYPE INFERENCE

```
resource [ "file:prices.wml" , "xml" ],
  wml [ all card [ "Title: " , var Title , "Price A: " , var PriceA, "Price B: " , var PriceB ] ]
} FROM
END

CONSTRUCT
```
As the program contains more than one query rule it has not been checked by our prototype. However, we present results, which would be obtained from an improved version of the prototype.

Since no output type specification is given the type system may infer a type of the program results and check if it is not empty.

First the result type for the third query rule is inferred which is \textit{Books-with-prices}. Then the result types for the first and the second goal are inferred which are respectively \textit{Html} and \textit{Wml}. These types are defined by the following Type Definition:

\begin{align*}
\text{Books-with-prices} & \rightarrow \text{books-with-prices[ Book-with-prices$^+$ ]} \\
\text{Book-with-prices} & \rightarrow \text{book-with-prices[Title Price-a Price-b]}
\end{align*}

\begin{align*}
\text{Price-a} & \rightarrow \text{price-a[ Text]}
\text{Price-b} & \rightarrow \text{price-b[ Text]}
\text{Title} & \rightarrow \text{title[ Text]}
\text{Html} & \rightarrow \text{html[ Head Body]}
\text{Head} & \rightarrow \text{head[ TitleH]}
\text{TitleH} & \rightarrow \text{title[ Text1]}
\text{Text1} & \rightarrow \text{"Price Overview"}
\text{Body} & \rightarrow \text{body[ Table]}
\text{Table} & \rightarrow \text{table[ TrH Tr$^+$ ]}
\text{TrH} & \rightarrow \text{tr[ TdH0 TdH1 TdH2]}
\text{TdH0} & \rightarrow \text{td[ Text2]}
\text{Text2} & \rightarrow \text{"Title"}
\text{TdH1} & \rightarrow \text{td[ Text3]}
\text{Text3} & \rightarrow \text{"Price at A"}
\text{TdH2} & \rightarrow \text{td[ Text4]}
\text{Text4} & \rightarrow \text{"Price at B"}
\text{Tr} & \rightarrow \text{tr[ Td Td Td]}
\text{Td} & \rightarrow \text{td[ Text]}
\text{Wml} & \rightarrow \text{wml[ Card$^+$ ]}
\text{Card} & \rightarrow \text{card[ Text5 Text6 Text7 Text]}
\text{Text5} & \rightarrow \text{"Title : "}
\text{Text6} & \rightarrow \text{"PriceA : "}
\text{Text7} & \rightarrow \text{"PriceB : "}
\end{align*}

The result types inferred for the query rules are not empty as the query terms in the third query rule match the data terms of the types given by specification and the query terms in both goals match the data terms produced by the third query rule.
6.2 Result Type Correctness

This section presents examples showing how the type system can help in checking Xcerpt programs correctness.

Consider the query rule from the Section 6.1.1:

```
CONSTRUCT
  result [
    all entry [
      var ARTIST,
      all var TITLE
    ]
  ]
FROM
  in { resource [ "file:cds.xml" ],
    bib {
      cd [[ var TITLE, var ARTIST, "pop" ]]
    }
  }
END
```

The type of the resource cds.xml is Cds as defined in the Section 6.1.1. This time, additionally, a type specification for the query rule results is given. The specified result type is \textit{Result} as defined below:

\[
\begin{align*}
\text{Result} & \rightarrow \text{result} [ \text{Entry}^* ] \\
\text{Entry} & \rightarrow \text{entry} [ \text{Artist Title}^+ ] \\
\text{Artist} & \rightarrow \text{artist} [ \text{Text} ] \\
\text{Title} & \rightarrow \text{title} [ \text{Text} ]
\end{align*}
\]

The result returned by the typechecker for the program is Printout 6.2.a. The type system infers the types of variables used in the query rule which are given by variable type mappings: \([\text{TITLE} \rightarrow \text{Title}, \text{ARTIST} \rightarrow \text{Artist}], [\text{TITLE} \rightarrow \text{Artist}, \text{ARTIST} \rightarrow \text{Artist}]\). Based on the type of resource being queried the query rule result type is inferred. The inferred result type is \textit{Result'} defined as

\[
\begin{align*}
\text{Result'} & \rightarrow \text{result} [ \text{Entry'}^+ ] \\
\text{Entry'} & \rightarrow \text{entry} [ \text{Artist (Title | Artist)}^+ ] \\
\text{Artist} & \rightarrow \text{artist} [ \text{Text} ] \\
\text{Title} & \rightarrow \text{title} [ \text{Text} ]
\end{align*}
\]

As the type \textit{Entry'} is not a subtype of the type \textit{Entry}, the inferred query result type \textit{Result'} is not included in the type \textit{Result}. This information may suggest a type error. However, a type inclusion check failure is not a proof of a type incorrectness of the program as the inferred result type \textit{Result'} is not exact\(^2\) (as the query rule uses the construct \texttt{all}). Nevertheless, there is a type

\(^2\) The inferred result type \textit{Result'} is a superset of the real result type which is \textit{Result''} defined as \[
\text{Entry''} \rightarrow \text{entry} (\text{Artist (Title | Artist)}^* \text{ Title (Title | Artist)}^*)
\]
error in the program. An intention for the query rule is to produce a result containing entries with one artist and all his/her titles (at least one). However, the query rule may produce a result with entries containing more than one artist e.g.

\[
\text{result} \left[ \text{entry} \left[ \text{artist} \left[ \text{"artist1"} \right], \text{title} \left[ \text{"title1"} \right] \right], \right.
\text{entry} \left[ \text{artist} \left[ \text{"artist2"} \right], \text{title} \left[ \text{"title1"} \right], \right.
\text{artist} \left[ \text{"artist1"} \right] \right] \right]
\]

The abovementioned result is obtained if the query rule is applied to the data term:

\[
\text{cd} \left[ \text{title} \left[ \text{"title1"} \right], \text{artist} \left[ \text{"artist1"} \right], \text{artist} \left[ \text{"artist2"} \right] \right]
\]

Consider the following Type Definition which specifies the result type of the query rules presented in the remaining part of this section. This type definition specifies documents describing books. The documents are of the format similar to HTML:

\[
\begin{align*}
\text{TextBook} & \rightarrow \text{book} \left[ \text{Cover Body} \right] \\
\text{Cover} & \rightarrow \text{cover} \left[ \text{Title Author* Publisher?} \right] \\
\text{Body} & \rightarrow \text{body} \left[ \text{Abstract? Chapter*} \right] \\
\text{Title} & \rightarrow \text{title} \left[ \text{InlineContent} \right] \\
\text{Author} & \rightarrow \text{author} \left[ \text{Text} \right] \\
\text{Publisher} & \rightarrow \text{publisher} \left[ \text{Text} \right] \\
\text{Abstract} & \rightarrow \text{abstract} \left[ \text{Text} \right] \\
\text{Chapter} & \rightarrow \text{chapter} \left[ \text{Title Section*} \right] \\
\text{InlineContent} & \rightarrow \text{inline} \left[ \text{Text | Bf | Em} \right] \\
\text{Section} & \rightarrow \text{section} \left[ \text{Title (Paragraph | Table | List)*} \right] \\
\text{Em} & \rightarrow \text{em} \left[ \text{InlineContent} \right] \\
\text{Bf} & \rightarrow \text{bf} \left[ \text{InlineContent} \right] \\
\text{Paragraph} & \rightarrow \text{p} \left[ \text{InlineContent*} \right] \\
\text{Table} & \rightarrow \text{table} \left[ \text{TableRow*} \right] \\
\text{List} & \rightarrow \text{list} \left[ \text{ListItem} \right] \\
\text{TableRow} & \rightarrow \text{tr} \left[ \text{TableCell*} \right] \\
\text{ListItem} & \rightarrow \text{item} \left[ \text{InlineContent*} \right] \\
\text{TableCell} & \rightarrow \text{td} \left[ \text{InlineContent*} \right]
\end{align*}
\]

The few following query rules query a document \textit{bibliography.xml} of the type \textit{Bibliography} defined in the Section \textbf{6.1.2} Consider the query rule:

\[
\text{CONSTRUCT} \\
\text{book} \left[ \\
\text{cover} \left[ \text{title} \left[ \text{"List_of_Books"} \right] \right], \\
\text{body} \left[ \\
\text{table} \left[
\right]
\right]
\right]
\]

\footnote{The Type Definition and the two following examples of query rules were devised by Sacha Berger.}
6.2. RESULT TYPE CORRECTNESS

The result returned by the typechecker for the program is Printout 6.2.b. As the inferred result type of the query rule is not included in the type `TextBook` a type error is possible. This is due to the structure of the construct term used as a head of the query rule. The construct term creates a data term `body[...]` with `table[...]` as a direct subterm. According to the type specification `body[...]` can not contain any `table[...]` direct subterms. Note that in this case the inferred types of variables do not matter. Whatever variable-type mappings we get from the body of the query rule the result type is still wrong due to the structure of the construct term which does not conform to the specified result type.

Here is another query rule with a possibility of a type error as the inferred result type is not included in the type `TextBook`. This time the structure of the head of the rule conforms to the specified result type. The typechecking failure is due to the variables which get wrong values i.e. not of the types required by the result type specification. The result returned by the typechecker for the following program is Printout 6.2.c.

```
CONSTRUCT
  book [
    cover [ title ["Books"] ],
    body [
      chapter [
        title ["List_of_Books_and_Authors"],
        table [
          all tr [
            td [ var TITLE],
            td [ all em [ var FIRST, var LAST ]]
        ]
      ]
    ]
  ]
FROM
  in{ resource ["file:bibliography.xml"],
    bib {
      book {
        title [ var TITLE ],
        authors [[
          person {
            first [ var FIRST ],
            last [ var LAST ]
          }]
        ]
      }
    }
  }
END
```

The result returned by the typechecker for the program is Printout 6.2.b. As the inferred result type of the query rule is not included in the type `TextBook` a type error is possible. This is due to the structure of the construct term used as a head of the query rule. The construct term creates a data term `body[...]` with `table[...]` as a direct subterm. According to the type specification `body[...]` can not contain any `table[...]` direct subterms. Note that in this case the inferred types of variables do not matter. Whatever variable-type mappings we get from the body of the query rule the result type is still wrong due to the structure of the construct term which does not conform to the specified result type.

Here is another query rule with a possibility of a type error as the inferred result type is not included in the type `TextBook`. This time the structure of the head of the rule conforms to the specified result type. The typechecking failure is due to the variables which get wrong values i.e. not of the types required by the result type specification. The result returned by the typechecker for the following program is Printout 6.2.c.
The variable NAME used in the body of the query rule is unrestricted and it can be bound to any data term which is a direct or an indirect subterm of \textit{book[...]} (except a data term \textit{title[...]}). Thus, the variable NAME may be mapped to the types: \textit{Authors, Editors, Publisher}, etc. In the construct term the variable NAME is used to build content of cells of a table and according to the type specification it should be of a one of the types allowed for subterms of \textit{inlineContent[...]} which are \textit{Text, Bf} and \textit{Em}. A type error is likely as the union of the inferred types for the variable NAME is not included in the union of the types \textit{Text, Bf} and \textit{Em}. 
Chapter 7

Conclusions

The thesis shows how the information about the structure of the documents in an XML database, specified usually in a schema language such as DTD, XML Schema, RELAX NG, can be used for reasoning about correctness of Xcerpt queries and programs. For this purpose we introduce a type system for (a substantial fragment) of Xcerpt. The system is formalized by means of typing rules. It is descriptive; this means that types assigned to a program approximate its semantics.

For specifying types we adopted the formalism of Type Definitions that abstracts from specific features of concrete schema languages, irrelevant for the kind of analysis we aim at. In practice an XML schema provided in some schema language has to be transformed into a Type Definition capturing or approximating the structure information expressed by the schema. Our prototype accepts Type Definitions and DTDs, but as discussed in Section 3.3 in some cases translation of specifications written in XML Schema or RELAX NG into Type Definitions may be difficult. This problem was not addressed in the thesis. The sets defined by Type Definitions are decidable. We employ a restriction on Type Definitions which allows efficient algorithms for primitive operations on types. The restricted formalism called proper Type Definition is closed under intersection.

As Xcerpt is a rule based declarative language the approach presented in this work is motivated by and related to the work on proving correctness and descriptive typing of logic programs (see [23, 22] and references therein). That work is not directly applicable to Xcerpt, due to semantic differences between logic programming and Xcerpt.

The reasoning techniques presented in this thesis make possible:

- Computing a type which contains all the results of a given query rule, under an assumption that the rule is applied to data from a given type. This is often called type inference. Our type inference approach also works for non-recursive Xcerpt programs. The algorithm for type inference is inefficient in general but we present possible optimizations. It seems that
CHAPTER 7. CONCLUSIONS

for cases occurring in practice such an optimized algorithm can be used effectively.

- Proving correctness of a query rule (or an Xcerpt program) wrt. a given type specification. This is often called type checking. Correctness means that whenever the rule (the program) is applied to data from a given type (of the database) the result is from a given type (of expected results). Our type checking approach also works for recursive programs. (In this case it is required that there is a specified type of results for each query rule in the program). Correctness can be proved by successful checking whether the inferred type of results is included in the specified one. Failure of such check suggests that the program may be incorrect. In Section 4.3.6 we studied conditions (on the program and on the type specification), under which the type inference is exact. In such cases failure of the inference check does imply program incorrectness. The typechecking can be done by means of the presented type inclusion algorithm. Such a check is of polynomial complexity provided that the result type specification is expressed by a proper Type Definition containing only 1-unambiguous content models.

Information provided by the type system can be useful for a programmer in a few ways. Having a specification of a result type the programmer can be warned about a possible error or assured about program correctness wrt. the specification. Without a result type specification given, the programmer may check the inferred result type manually to see if it is as intended. The types of variables which are also inferred can facilitate error discovery. It may happen that the inferred type of program results is empty which is also a suggestion of an error.

We demonstrated the usefulness of the proposed approach for indicating errors in Xcerpt programs by running example Xcerpt programs on the implemented prototype. Some of the techniques discussed in the thesis are not yet implemented. The next release of the prototype will make it possible to handle multiple rule programs and to check type correctness of recursive programs. It should be also able to handle the full Xcerpt by providing rough type approximations related to the Xcerpt constructs not covered by this thesis. Several improvements outlined in the thesis, for instance those allowing computing more precise types, will also be implemented.

Further research on the following issues not addressed in this thesis would be relevant for ongoing development of Xcerpt:

- Location of type errors. The presented type system is able to find out that an error is possible, but it does not locate the error in the program. If a type specification of results for each query rule in the program is given, the system is able to locate a (possibly) incorrect query rule. Then the programmer’s role is to locate the actual error. An important issue is what additional information should be presented to the programmer to assist him/her in this task. The current prototype provides inferred types
of variables. When extended to multiple rule programs, it will provide the result type for each rule of the program. Another idea is to provide an example of a query rule result which is not intended by a user i.e. an example of a result which is not of the specified result type. A related theoretical problem is more precise automatic error location (than that locating only a whole query rule). A necessary first step is defining formally what it means that a fragment of a query rule (like a construct term or a query) is erroneous.

- Query optimization. Knowledge about a type of a database can be used to simplify queries. For example, knowing from the type specification that a data term $t$ is a mandatory direct subterm of another data term we can remove the construct `optional` from the query term intended to match the data term $t$.

- Type systems for Semantic Web applications of Xcerpt. As shown in [26], Xcerpt can be applied to querying RDF data. A proposal for extension of Xcerpt for querying XML data with semantic annotations is discussed in [39]. An interesting issue is adjusting and extending the presented type system for this kind of applications.
Appendix A

Proofs

A.1 Type System Correctness

Theorem 1 and Proposition 5 are very similar to Theorem 20 and Proposition 23 (respectively) of [6]. Thus, their proofs are also very similar and to prove them we use here lemmas and propositions from [6]. However, the set of rules on which the cited lemmas and propositions are based, is slightly different in this thesis than in [6]. One difference is in the rule (TARGETED QUERY TERM), which instead of the condition \(d(db) \in [T]\) has a condition \(\text{type}(db) = T\). The other difference is in the construct term typing rule (VAR). In the current version, the rule requires that a Type Definition \(D\) is weakly proper which was not the case in [6]. Additionally, we provide another typing rule for construct terms, namely (VAR APPROX), with a condition that \(D\) is not weakly proper. The last difference concerns the rule (QUERY RULE) which now has additional condition saying that \(\text{substitutions}(\Gamma) \neq \emptyset\) for each \(\Gamma \in \Psi\). As the modifications of the typing rule set require only small and simple changes in lemmas, propositions and proofs recalled from [6], we do not describe them here.

First, we need to recall an auxiliary definition of \(\Gamma_\theta\) (defined earlier in [6]).

**Definition 22.** Given a Type Definition \(D\) and a substitution \(\theta\), the mapping \(\Gamma_\theta\) is defined as:

\[
\Gamma_\theta(X) = \begin{cases} 
T_1 \cap \ldots \cap T_n & \text{if } X \in \text{dom}(\theta) \text{ and } \{T_1, \ldots, T_n\} = \{T \mid X_\theta \in [T]_D\} \\
1 & \text{otherwise}.
\end{cases}
\]

By definition \(\theta \in \text{substitutions}(\Gamma_\theta)\).

**Proposition 5.** Let \(D\) be a Type Definition, \(U\) a set of type names and \(Q\) a query such that for each targeted query term in (db, q) in \(Q\) there is a type name \(T = \text{type}(db)\) defined in \(D\). Let \(d\) be a data term such that \(d \in [U]_D\), if \(U \neq \emptyset\). If \(\theta\) is an answer for \(Q\) and \(d\) (or for \(Q\) and no data term, if \(U = \emptyset\)), then \(D \vdash Q : U \triangleright \Gamma_\theta\).

**Proof.** By induction on the query \(Q\).
Let \( Q \) be a query term. If \( U = \emptyset \), the proposition is not applicable as there is no answer for a query term and no data term. So \( U \neq \emptyset \). \( \theta \) is an answer for \( Q \) and \( d \), with \( d \in [T_i] \) for some \( T_i \in U \). By Lemma 22 of [6], \( D \vdash Q : T_i \triangleright \Gamma_\theta \). By rule (QUERY TERM), we obtain \( D \vdash Q : U \triangleright \Gamma_\theta \).

If \( Q \) is a targeted query term \( \text{in}(db,q) \), \( \theta \) is an answer for \( q \) and \( d(db) \). Let \( T = \text{type}(db) \). By Lemma 22 of [6], \( D \vdash q : T \triangleright \Gamma_\theta \). By rule (TARGETED QUERY TERM), we obtain \( D \vdash Q : U \triangleright \Gamma_\theta \).

Let \( Q \) be of the form \( \text{and}(Q_1,\ldots,Q_p) \). If \( U = \emptyset \), by Proposition 2, \( \theta \) is an answer for \( Q \) and an arbitrary data term \( d_a \). For each \( i \in \{1,\ldots,p\} \), \( \theta \) is an answer for \( Q_i \) and \( d \) (or \( d_a \), if \( U = \emptyset \)). By induction, we obtain, for each \( i \in \{1,\ldots,p\} \), \( D \vdash Q_i : U \triangleright \Gamma_\theta \). Thus, by rule (AND QUERY), we have \( D \vdash Q : U \triangleright \Gamma_\theta \).

Let \( Q \) be of the form \( \text{or}(Q_1,\ldots,Q_p) \). If \( U = \emptyset \), by Proposition 2, \( \theta \) is an answer for \( Q \) and an arbitrary data term \( d_a \). For some \( i \in \{1,\ldots,p\} \), \( \theta \) is an answer for \( Q_i \) and \( d \) (or \( d_a \), if \( U = \emptyset \)). By induction, we obtain \( D \vdash Q_i : U \triangleright \Gamma_\theta \). Thus, by rule (OR QUERY), we have \( D \vdash Q : U \triangleright \Gamma_\theta \).

\[ \square \]

**Theorem 1.** Let \( D \) be a Type Definition and \( (c \leftarrow Q) \) be a query rule, where for each targeted query term \( \text{in}(db,q) \) in \( Q \) there is a type name \( T = \text{type}(db) \) defined in \( D \). Let \( U \) be a set of type names and \( Z \) a set of data terms such that \( Z \subseteq [U] \).

If a result for \( (c \leftarrow Q) \) and \( Z \) exists then there exist \( s \) and \( D' \) such that \( D' \supseteq D \) and \( D' \vdash (c \leftarrow Q) : U \triangleright s \).

If \( D \vdash (c \leftarrow Q) : U \triangleright s \) and \( d' \) is a result for \( (c \leftarrow Q) \) and \( Z \), then \( d' \in [s] \).

**Proof.** Assume that there exists a result for \( (c \leftarrow Q) \) and \( Z \). Let \( \Theta \) be the set of all answers for \( Q \) and \( d \in Z \) (or, for \( Q \) and no data term if \( Z = \emptyset \)). By Definition 10, \( \Theta \neq \emptyset \). Let \( \theta \in \Theta \). By Proposition 2, \( D \vdash Q : U \triangleright \Gamma_\theta \). Thus, by Lemma 26 of [6], there is a set \( \Psi \) of variable type mappings that \( \Psi \) is complete for \( Q \) and \( U \).

Let \( V_Q \) be the set of variables occurring in \( Q \) and \( V_c \) the set of variables occurring in \( c \). Let \( \text{or}(Q'_1,\ldots,Q'_n) \) be the disjunctive normal form of \( Q \). Since Xcerpt requires that, for \( i \in \{1,\ldots,n\} \), all variables in \( c \) must occur in \( Q'_i \), by Lemma 24 of [6], we have that for all variables \( X \in V_c \) and all mappings \( \Gamma \in \Psi \), \( \Gamma(X) \neq 1 \).

Let \( \{\Psi_1,\ldots,\Psi_n\} = \Psi/\sim_{FV(o)} \). By induction from Lemma 27 of [6], we obtain that there exist \( D_n \supseteq \cdots \supseteq D_1 \supseteq D \) and \( s_1,\ldots,s_n \) such that \( D_i \vdash c : \Psi_i \triangleright s_i \) (and by Lemma 28 of [6], \( D_n \vdash c : \Psi_i \triangleright s_i \) for each \( i \in \{1,\ldots,n\} \)). By Lemma 29 of [6], \( \Psi \) is also complete for \( Q \) and \( U \). By rule (QUERY RULE), we obtain \( D_n \vdash (c \leftarrow Q) : U \triangleright s_1 \mid \cdots \mid s_n \).

Now assume that there exists \( s \) and \( D \) such that \( D \vdash c \leftarrow Q \triangleright s \). Let \( d' \) be a result for \( (c \leftarrow Q) \) and \( Z \). By Definition 10 there exists \( \Theta \) which is a set of all
A.1. TYPE SYSTEM CORRECTNESS

answers for \( Q \) and \( d \in Z \) (or, for \( Q \) and no data term if \( Z = \emptyset \)). By Proposition \[ a \] \( D \vdash Q : U \vdash \Gamma_\theta \). Since \( \Psi \) used in the rule (QUERY RULE) is complete for \( Q \) and \( U \) wrt. \( D \), there exists \( \Gamma \in \Psi \) such that \( \Gamma_\theta \subseteq \Gamma \). Since \( \theta \in \text{substitutions}(\Gamma_\theta) \), we obtain \( \theta \in \text{substitutions}(\Psi) \).

We have \( d' = \Theta'(c) \) for some \( \Theta' \in \Theta/\sim_{FV(c)} \). By Lemma 30 of \[ 6 \], there exists \( \Psi' \in \Psi/\sim_{FV(c)} \) such that \( \Theta' \subseteq \text{substitutions}(\Psi') \). Since \( D \vdash (c \leftarrow Q) : U \vdash s_1 \mid \cdots \mid s_n \), then for some \( i \in \{1, \ldots, n\} \), we have \( \Psi_i = \Psi' \) and \( D \vdash c : \Psi_i \triangleright s_i \).

By Proposition 31 of \[ 6 \], \( d = \Theta'(c) \in [s_i] \subseteq [s_1 \mid \cdots \mid s_n] \).

\[ \Box \]

**Proposition 6.** Let \( \mathcal{P} = (P, G) \) be a non recursive Xcerpt program and \( D \) a Type Definition, where for each targeted query term \( \text{in}(db, q) \) in \( P \) there is a type name \( T = \text{type}(db) \) defined in \( D \).

If a result for \( p \) in \( P \) exists then there exist \( s \), and \( D' \) such that \( D' \supseteq D \) and \( \mathcal{P}, D' \vdash p \triangleright s \).

Let \( \mathcal{P}, D \vdash p \triangleright s \) for \( p \in P \). If \( d \) is a result for \( p \) in \( P \) then \( d \in [s]_D \).

**Proof.** Let \( P_d = \{p_1, \ldots, p_n\} \) be the set of query rules on which \( p \) directly depends. Let \( d \) be a result for \( p \) in \( P \). By induction on the dependency tree of the set of rules \( P \).

- Let \( P_d = \emptyset \). By Definition \[ 5 \] \( d \) is a result for \( p \) and \( \emptyset \).

  By Theorem \[ 1 \] there exist \( s \) and \( D' \) such that \( D' \supseteq D \) and \( D' \vdash (c \leftarrow Q) : \emptyset \triangleright s \). By the rule (PROGRAM), \( \mathcal{P}, D' \vdash p \triangleright s \).

  Now assume that there exists \( s \) and \( D \) such that \( \mathcal{P}, D \vdash p \triangleright s \). As \( P_d = \emptyset \), by the rule (PROGRAM), \( D \vdash p : \emptyset \triangleright s \). By Theorem \[ 1 \] \( d \in [s] \).

- Let \( P_d \neq \emptyset \). Let \( Z \) be the set of data terms being results for query rules from \( P_d \). By Definition \[ 5 \] \( d \) is a result for \( p \) and \( Z \). By induction hypothesis, for each \( p_i \in P_d \), there exist \( s_i \) and \( D_i \), such that \( D_n \supseteq \cdots \supseteq D_1 \supseteq D \) and \( \mathcal{P}, D_i \vdash p_i \triangleright s_i \). Also, by induction hypothesis, for each \( d' \in Z \), \( d' \in [s_i]_{D_i} \) for some \( i = 1, \ldots, n \). Each \( s_i \) is of the form \( T_{i_1} \cdots T_{i_{k_i}} \). Let \( U \) be the set of type names \( \{T_{i_1}, \ldots, T_{i_{k_1}}, \ldots, T_{i_1}, \ldots, T_{i_{k_n}}\} \). Thus, for each \( d' \in Z \), \( d' \in [U]_{D_n} \) and then \( Z \subseteq [U]_{D_n} \).

  By Theorem \[ 1 \] there exist \( s \) and \( D' \) such that \( D' \supseteq D_n \) and \( D' \vdash (c \leftarrow Q) : U \triangleright s \). By the rule (PROGRAM), \( \mathcal{P}, D' \vdash p \triangleright s \).

  Now assume that there exists \( s \) and \( D \) such that \( \mathcal{P}, D \vdash p \triangleright s \). By the rule (PROGRAM), \( D \vdash p : U \triangleright s \). By Theorem \[ 1 \] \( d \in [s] \).

\[ \Box \]

**Theorem 2.** Let \( \mathcal{P} = (P, G) \) be a non recursive Xcerpt program and \( D \) a Type Definition, where for each targeted query term \( \text{in}(db, q) \) in \( P \) there is a type name \( T = \text{type}(db) \) defined in \( D \).

If a result of \( \mathcal{P} \) exists then there exist \( p \in G \), \( s \) and \( D' \) such that \( D' \supseteq D \), \( \mathcal{P}, D' \vdash p \triangleright s \).
APPENDIX A. PROOFS

If \( d \) is a result of \( \mathcal{P} \) then there exist \( p \in G \) such that if \( \mathcal{P}, D \vdash p \triangleright s \) then \( d \in \llbracket s \rrbracket_D \).

Proof. Assume that \( d \) is a result of \( \mathcal{P} \). By Definition \ref{def:goalexists} there exists a goal \( p \in G \) such that \( d \) is a result for \( p \).

By Proposition \ref{prop:queryexists} there exist \( s \), and \( D' \) such that \( D' \supseteq D \) and \( \mathcal{P}, D' \vdash p \triangleright s \).

Assume that \( \mathcal{P}, D \vdash p \triangleright s \). By Proposition \ref{prop:queryexists} \( d \in \llbracket s \rrbracket_D \).

\( \square \)

A.2 Exactness of Inferred Type

**Proposition** \ref{prop:queryexists}. Let \( D \) be a Type Definition without nullable type names, and whose content models do not contain useless symbols. Let \( q \) be a query term, \( T \) a type name from \( D \), and \( \Theta = \{ \theta \mid D \vdash q : T \triangleright \Gamma, \theta \in \text{substitutions}_D(\Gamma) \} \). If \( q \) does not contain \( \rightsquigarrow \) then each \( \theta \in \Theta \) is an answer for \( q \) and some \( d \in \llbracket T \rrbracket_D \).

Proof. Notice, that for any type name \( T \) occurring in \( D \), \( \llbracket T \rrbracket_D \neq \emptyset \). Assume that \( D \vdash q : T \triangleright \Gamma \) and \( \theta \in \text{substitutions}(\Gamma) \). We will show that \( \theta \) is an answer for \( q \) and some \( d \in \llbracket T \rrbracket \).

- If \( q \) is a basic constant then an arbitrary substitution \( \theta \) is an answer substitution for \( q \) and an arbitrary data term \( d \).
- If \( q \) is a variable \( X \), then given \( D \vdash q : T \triangleright \Gamma \) by the rule (VAR) \( \llbracket \Gamma(X) \rrbracket \subseteq \llbracket T \rrbracket \). As \( \theta \in \text{substitutions}_D(\Gamma) \) we obtain \( X\theta \in \llbracket \Gamma(X) \rrbracket \). Hence \( X\theta \in \llbracket T \rrbracket \). Thus, \( \theta \) is an answer for \( X \) and \( d = X\theta \).
- Let \( q \) be of the form \( l[q_1, \ldots, q_n] \) and the rule for \( T \) in \( D \) be of the form \( T \rightarrow l[r] \). Given \( D \vdash q : T \triangleright \Gamma \) by the query term typing rule (PATTERN), \( D \vdash q_i : T_i \triangleright \Gamma \) for \( i = 1, \ldots, n \) and \( T_i \in \text{perm}(L(r)) \). By induction hypothesis there exist data terms \( d_i \in \llbracket T_i \rrbracket \) (\( i = 1, \ldots, n \)) such that \( \theta \) is an answer for each \( q_i \) and \( d_i \). By the Definition \ref{def:dataexists} \( \theta \) is an answer for \( q \) and \( l[d_1, \ldots, d_n] \in \llbracket T \rrbracket \).
- Let \( q \) be of the form \( l[q_1, \ldots, q_n] \) and the rule for \( T \) in \( D \) be of the form \( T \rightarrow l[r] \). Given \( D \vdash q : T \triangleright \Gamma \) by the query term typing rule (PATTERN), \( D \vdash q_i : T_i \triangleright \Gamma \) for \( i = 1, \ldots, n \) and \( T_i \in \text{perm}(L(r)) \). By induction hypothesis there exist data terms \( d_i \in \llbracket T_i \rrbracket \) (\( i = 1, \ldots, n \)) such that \( \theta \) is an answer for each \( q_i \) and \( d_i \). Let \( t_1, \ldots, t_n \) be a permutation of \( d_1, \ldots, d_n \) such that \( l[t_1, \ldots, t_m] \in \llbracket T \rrbracket \). By the Definition \ref{def:dataexists} \( \theta \) is an answer for \( q \) and \( l[t_1, \ldots, t_n] \in \llbracket T \rrbracket \).
- Let \( q \) be of the form \( l[[q_1, \ldots, q_n]] \) and the rule for \( T \) in \( D \) be of the form \( T \rightarrow l[r] \). Given \( D \vdash q : T \triangleright \Gamma \) by the query term typing rule (PATTERN), \( D \vdash q_i : T_i \triangleright \Gamma \) for \( i = 1, \ldots, n \) and \( T_i \in L(s) \), where \( s \) is \( r \) with every type name \( U \) replaced by \( U|\epsilon \). By induction hypothesis there exist data terms \( d_i \in \llbracket T_i \rrbracket \) (\( i = 1, \ldots, n \)) such that \( \theta \) is an answer for each \( q_i \) and \( d_i \). Let \( t_1, \ldots, t_m \) be a sequence of data terms containing subsequence
Let \( q \) be of the form \( l\{q_1, \ldots, q_n\} \) and the rule for \( T \) in \( D \) be of the form \( T \rightarrow l\{r\} \). Given \( D \vdash q : T \triangleright \Gamma \), by the query term typing rule (\textsc{Pattern}), \( D \vdash q_i : T_i \triangleright \Gamma \) for \( i = 1, \ldots, n \) and \( T_1 \cdots T_n \in \text{perm}(L(s)) \), where \( s \) is the content model of \( U \) replaced by \( U \triangleright \epsilon \). By induction hypothesis there exist data terms \( d_i \in [T_i] (i = 1, \ldots, n) \) such that \( \theta \) is an answer for each \( q_i \) and \( d_i \). Let \( t_1, \ldots, t_m \) be a sequence of data terms containing subsequence \( d_1, \ldots, d_n \) such that \( l[t_1, \ldots, t_m] \in [T] \). By the Definition 3, \( \theta \) is an answer for \( q \) and \( l[t_1, \ldots, t_m] \in [T] \).

Let \( q \) be of the form \( \text{desc} \ q' \). Given \( D \vdash q : T \triangleright \Gamma \), by the query term typing rule (\textsc{Pattern}), \( D \vdash q_i : T_i \triangleright \Gamma \) for \( i = 1, \ldots, n \) and \( T_1 \cdots T_n \in \text{perm}(L(s)) \), where \( s \) is the content model of \( U \) replaced by \( U \triangleright \epsilon \). By induction hypothesis there exist data terms \( d_i \in [T_i] (i = 1, \ldots, n) \) such that \( \theta \) is an answer for each \( q_i \) and \( d_i \). Let \( t_1, \ldots, t_m \) be a sequence of data terms containing subsequence \( d_1, \ldots, d_n \) such that \( l[t_1, \ldots, t_m] \in [T] \). By the Definition 3, \( \theta \) is an answer for \( q \) and \( l[t_1, \ldots, t_m] \in [T] \).

Let \( q \) be of the form \( \text{desc} \ q' \). Given \( D \vdash q : T \triangleright \Gamma \), by the query term typing rule (\textsc{Pattern}), \( D \vdash q_i : T_i \triangleright \Gamma \) for \( i = 1, \ldots, n \) and \( T_1 \cdots T_n \in \text{perm}(L(s)) \), where \( s \) is the content model of \( U \) replaced by \( U \triangleright \epsilon \). By induction hypothesis there exist data terms \( d_i \in [T_i] (i = 1, \ldots, n) \) such that \( \theta \) is an answer for each \( q_i \) and \( d_i \). Let \( t_1, \ldots, t_m \) be a sequence of data terms containing subsequence \( d_1, \ldots, d_n \) such that \( l[t_1, \ldots, t_m] \in [T] \). By the Definition 3, \( \theta \) is an answer for \( q \) and \( d' \).

\begin{proposition}
Let \( D \) be a Type Definition without nullable type names, and whose content models do not contain useless symbols. Let \( U \) be a set of type names from \( D \), \( Q \) be a query and \( \Theta = \{ \theta \mid D \vdash q : T \triangleright \Gamma, \theta \in \text{substitutions}_{D}(\Gamma) \} \). Let \( T_1, \ldots, T_n \) be type names in \( D \) such that \( \text{type}(d_i) = T_i \) for each targeted query term \( \text{in}(d_i, q) \) in \( Q \) (\( i = 1, \ldots, n \)). If \( Q \) does not contain \( \sim \) and a construct \text{and}(\ldots) then for each \( \theta \in \Theta \) there exist
\end{proposition}
Proof. We assume that $\theta$ is an answer for $Q'$ and $d$ (or, if $U = \emptyset$, for $Q'$ and no data term), where $Q'$ is $Q$ with each targeted query term $\text{in}(d_{b'_i}, q_i)$ replaced by a targeted query term $\text{in}(d_{b'_i}, q_i)$, such that $d(d_{b'_i}) = d_i$.

Proof. We assume that $D \vdash Q : U \triangleright \Gamma$ and $\theta \in \text{substitutions}_D(\Gamma)$. Let $T_1, \ldots, T_n$ be type names such that $\text{type}(d_{b_i}) = T_i$ for each targeted query term $\text{in}(d_{b_i}, q_i)$ in $Q$. By induction on the query $Q$:

- Let $Q$ be a query term. As there are no targeted query terms in $Q$, $Q' = Q$. By query typing rule (QUERY TERM), $D \vdash Q' : U \triangleright \Gamma$ implies $D \vdash Q' : T \triangleright \Gamma$ for some $T \in U$. Thus, by Proposition\textsuperscript{3} there exists a data term $d \in [T]$ such that $\theta$ is an answer substitution for $Q'$ and $d$. Hence, there exists $d \in [U]$ such that $\theta$ is an answer substitution for $Q'$ and $d$.

For any query term $Q$, $D \vdash Q : \emptyset \triangleright \Gamma$ does not hold, as the rule (QUERY TERM) requires $U$ to be not empty. Thus the proposition is not applicable for an empty set of type names $U$ and a query $Q$ which is a query term.

- Let $Q$ be a targeted query term $\text{in}(d_{b_i}, q_i)$. By the query typing rule (TARGETED QUERY TERM), $D \vdash Q : U \triangleright \Gamma$ implies $D \vdash q_i : T_i \triangleright \Gamma$ for $T_i = \text{type}(d_{b_i})$. As $D \vdash q_i : T_i \triangleright \Gamma$, by Proposition\textsuperscript{3} there exists $d_i \in [T_i]$ such that $\theta$ is an answer for $q_i$ and $d_i$. Let $Q'$ be $\text{in}(d_{b'_i}, q_i)$, where $d(d_{b'_i}) = d_i$.

Assume that $U \neq \emptyset$. Let $d \in [U]$ be some data term. By the definition of targeted query term, $\theta$ is an answer for $Q'$ and $d$.

Assume that $U = \emptyset$. By the definition of targeted query term, $\theta$ is an answer for $Q'$ and an arbitrary data term. By Definition\textsuperscript{5} $\theta$ is an answer for $Q'$ and no data term.

- Let $Q$ be of the form $\text{o}(Q_1, \ldots, Q_n)$. By induction hypothesis there exist
  - data terms $d_1, \ldots, d_n$ of types $T_1, \ldots, T_n$, respectively,
  - a data term $d \in [U]_D$ (if $U \neq \emptyset$)

such that $\theta$ is an answer for some $Q'_j$ and $d$ (or, if $U = \emptyset$, for $Q'_j$ and no data term), where $Q'_j$ is $Q_j$ with each targeted query term $\text{in}(d_{b'_p}, q_p)$ replaced by a targeted query term $\text{in}(d_{b'_p}, q_p)$, such that $d(d_{b'_p}) = d_p$. Let $Q' = \text{o}(Q_1, \ldots, Q'_j, \ldots, Q_n)$. If $U \neq \emptyset$, by Definition\textsuperscript{4} $\theta$ is an answer for $Q'$ and $d$. If $U = \emptyset$ then by Definitions\textsuperscript{4, 5} $\theta$ is an answer for $Q'$ and no data term (as there exist a disjunctive normal form $\text{o}(Q'_1, \ldots, Q'_m)$ of $Q'$ such that $\text{o}(Q'_k, \ldots, Q'_m)$ ($1 \leq k \leq l \leq m$) is a disjunctive normal form of $Q'_j$).
Appendix B

Typechecker Printouts

This chapter presents printouts from the typechecker prototype. The printouts are results of typing the program examples from Chapter 6.

Printout 6.1.1

Type inference ...
-------------------------------
Result type: result
-------------------------------
Variable-type mappings:
-------------------------------
TITLE->Artist, ARTIST->Artist
TITLE->Title, ARTIST->Artist

Type Definition:
-------------------------------
result -> result[entry+]
entry -> entry[(Artist (Artist|Title)+)]
Cds -> bib[(( Cd* ))]
Cd -> cd[(( Title ( (Artist+ ( (Category? ))) ))]]
Title -> title[#]
Artist -> artist[#]
Category -> pop | rock | classic
-------------------------------
Evaluation took 0,30s
Printout 6.1.2

==============================================
Type inference ...
==============================================
Result type: 0
==============================================
Variable-type mappings:
0

==============================================
Type Definition:
----------------------------------------------
Bibliography -> bib[ (((Book )|(( (Article )))|( InProceedings)))* ]
Book -> book{Title Authors Editors Publisher?}
Article -> article{Title Authors Journal?}
InProceedings -> inproc{Title Authors Book}
Title -> title[ ( # )]
Authors -> authors[ (Person* )]
Editors -> editors[ (Person* )]
Publisher -> publisher[ ( # )]
Journal -> journal{Title Editors }
Person -> person[(FirstName LastName)]
FirstName -> first[ ( # )]
LastName -> last[ ( # )]

Evaluation took 0,10s

Printout 6.2.a

==============================================
Type checking ... FAILED
==============================================
Result type: result (not a subset of Result)

==============================================
Variable-type mappings:
TITLE->Artist, ARTIST->Artist
TITLE->Title, ARTIST->Artist

==============================================
Type Definition:
----------------------------------------------
result -> result[entry*]
entry -> entry[ (Artist (Artist|Title)+)]
Cds -> bib[[ (Cd* )]]
Cd -> cd[( (Title ( (Artist+ ( (Category? ))) )))]
Evaluation took 0.30s

Printout 6.2 b

Type checking ... FAILED
Result type: book (not a subset of TextBook)

Variable-type mappings:
TITLE->#, FIRST->#, LAST->#

Type Definition:
book -> book[(cover body)]
body -> body[table]
table -> table[tr+]
tr -> tr[td td_1]
td_1 -> td[em+]
em -> em[# #]
td -> td[#]
cover -> cover[title]
title -> title[List_of_Books]
List_of_Books -> List_of_Books
TextBook -> book[(Cover (Body ))]
Cover -> cover[(Title (Author* Publisher?))]
Body -> body[(Abstract? (Chapter* ))]
Title -> title[(InlineContent )]
Author -> author[( # )]
Publisher -> publisher[( # )]
Abstract -> abstract[( # )]
Chapter -> chapter[(Title (Section* ))]
InlineContent -> inline[( ( # ))][ (Bf )][ (Em )]
Section -> section[(Title (Paragraph )][ (Table )][ (List)])* )]
Em -> em[(InlineContent )]
Bf -> bf[(InlineContent )]
Paragraph -> p[(InlineContent* )]
APPENDIX B. TYPECHECKER PRINTOUTS

Table -> table[((TableRow+ ))]
List -> list[((ListItem ))]
TableRow -> tr[((TableCell* ))]
ListItem -> item[((InlineContent* ))]
TableCell -> td[((InlineContent* ))]
Bibliography -> bib[((((Book )))|((Article ))|((InProceedings)))* )]]
Book -> book{Title1 Authors Editors Publisher?}
Article -> article{Title1 Authors Journal?}
InProceedings -> inproc{Title1 Authors Book}
Title1 -> title{(# )]
Authors -> authors{((Person* ))]
Editors -> editors{((Person* ))]
Journal -> journal{Title1 Editors}
Person -> person{FirstName LastName]
FirstName -> first{(# )]
LastName -> last{(# )]

Evaluation took 0,40s

Printout 6.2.c

Type checking ... FAILED

Result type: book (not a subset of TextBook)

Variable-type mappings:

TITLE->#, NAME->Publisher
TITLE->#, NAME->#
TITLE->#, NAME->Editors
TITLE->#, NAME->Person
TITLE->#, NAME->FirstName
TITLE->#, NAME->LastName
TITLE->#, NAME->Authors

Type Definition:

book -> book{cover body]
body -> body[chapter]
chapter -> chapter{(title_1 table]
table -> table{tr[tr_1|(tr_2|tr_3|tr_4|tr_5|tr_6))])]+
tr_6 -> tr[td_12 td_13]
td_13 -> td[inlineContent_13]
inlineContent_13 -> inlineContent[Authors]
td_12 -> td[inlineContent_12]
APPENDIX B. TYPECHECKER PRINTOUTS

List -> list(( ListItem ))
TableRow -> tr(( TableCell* ))
ListItem -> item(( InlineContent* ))
TableCell -> td(( InlineContent* ))
Bibibliography -> bib(( ((Book )|(( Article ))|( InProceedings)))* ))
Book -> book{Title1 Authors Editors Publisher?}
Article -> article{Title1 Authors Journal?}
InProceedings -> inproc{Title1 Authors Book}
Title1 -> title(( # ))
Authors -> authors(( Person* ))
Editors -> editors(( Person* ))
Journal -> journal{Title1 Editors}
Person -> person{(FirstName LastName)}
FirstName -> first(( # ))
LastName -> last(( # ))

Evaluation took 0,30s
Bibliography


105


The thesis presents a type system for a substantial fragment of XML query language Xcerpt. The system is descriptive; the types associated with Xcerpt constructs are sets of data terms and approximate the semantics of the constructs. A formalism of Type Definitions, related to XML schema languages, is adopted to specify such sets. The type system is presented as typing rules which provide a basis for type inference and type checking algorithms, used in a prototype implementation. Correctness of the type system wrt. the formal semantics of Xcerpt is proved and exactness of the result types inferred by the system is discussed. The usefulness of the approach is illustrated by example runs of the prototype on Xcerpt programs.

Given a non-recursive Xcerpt program and types of data to be queried, the type system is able to infer a type of results of the program. If additionally a type specification of program results is given, the system is able to prove type correctness of a (possibly recursive) program. Type correctness means that the program produces results of the given type whenever it is applied to data of the given type. Non existence of a correctness proof suggests that the program may be incorrect. Under certain conditions (on the program and on the type specification), the program is actually incorrect whenever the proof attempt fails.