Linköping Studies in Science and Technology

Dissertation No. 1156

Types for XML with Application to Xcerpt

by

Artur Wilk

CUGS
National Graduate School of Computer Science

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

Linköping 2008

Printed by LiU-Tryck, Linköping 2008

Abstract

XML data is often accompanied by type information, usually expressed by some schema language. Sometimes XML data can be related to ontologies defining classes of objects, such classes can also be interpreted as types. Type systems proved to be extremely useful in programming languages, for instance to automatically discover certain kinds of errors. This thesis deals with an XML query language Xcerpt, which originally has no underlying type system nor any provision for taking advantage of existing type information. We provide a type system for Xcerpt; it makes possible type inference and checking type correctness.

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* is adapted to specify such sets. The formalism may be seen as a simplification and abstraction of XML schema languages. The type inference method, which is the core of this work, may be seen as abstract interpretation. A non standard way of assuring termination of fixed point computations is proposed, as standard approaches are too inefficient. The method is proved correct wrt. the formal semantics of Xcerpt.

We also present a method for type checking of programs. A success of type checking implies that the program is correct wrt. its type specification. This means that the program produces results of the specified type whenever it is applied to data of the given type. On the other hand, a failure of type checking 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.

A prototype implementation of the type system has been developed and usefulness of the approach is illustrated on example programs.

In addition, the thesis outlines possibility of employing semantic types (ontologies) in Xcerpt. Introducing ontology classes into Type Definitions makes possible discovering some errors related to the semantics of data queried by Xcerpt. We also extend Xcerpt with a mechanism of combining XML queries with ontology queries. The approach employs an existing Xcerpt engine and an ontology reasoner; no modifications are required.

Acknowledgments

I would like to express my deepest gratitude to my advisor Włodek Drabent for his engagement in this work and for discussions concerning almost all issues of the presented thesis. This work would never have been accomplished without his support.

I am very grateful to my second advisor Jan Małuszyński whose guidance and constant encouragement throughout this work have been invaluable. His interesting ideas have been inspiration for this work.

I thank Emmanuel Coquery, and also Sacha Berger, 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 Tim Furche.

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

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

Artur Wilk
Linköping, February 2008

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. It was also supported by CUGS (The National Graduate School in Computer Science) and SWEBPROD (Semantic Web for Products) by Vinnova.

List of Papers

Parts of the thesis are based on the following papers.

- A. Wilk and W. Drabent. *On types for XML query language Xcerpt.* In Proceedings of the First Workshop on Principles and Practice of Semantic Web Reasoning. LNCS 2901, Mumbai, India, 2003, 128-145.
- S. Berger, E. Coquery, W. Drabent, and A. Wilk. *Descriptive typing rules for Xcerpt.* In Proceedings of the Third Workshop on Principles and Practice of Semantic Web Reasoning. Dagstuhl, Germany, 2005. LNCS 3703, 85-100.
- E. Svensson and A. Wilk. *XML Querying Using Ontological Information.* In Proceedings of the Fourth Workshop on Principles and Practice of Semantic Web Reasoning. Budva, Montenegro, 2006. LNCS 4187, 190-203.
- A. Wilk and W. Drabent. *A prototype of a descriptive type system for Xcerpt.* In Proceedings of the Fourth Workshop on Principles and Practice of Semantic Web Reasoning. Budva, Montenegro, 2006. LNCS 4187, 262-275.
- W. Drabent and A. Wilk. *Combining XML querying with ontology reasoning: Xcerpt and DIG.* Online Proceedings of RuleML Workshop: Ontology and Rule Integration, Athens, Georgia, USA, 2006, `http://2006.ruleml.org/group3.html#3`.
- W. Drabent and A. Wilk. *Extending XML Query Language Xcerpt by Ontology Queries.* In Proceedings of IEEE / WIC / ACM International Conference on Web Intelligence (WI 2007), Silicon Valley, USA, 2007. IEEE, WIC, ACM.

Contents

Chapter 1

Introduction

The Problem and the Motivation

The work presented in this thesis is related to XML which is a dominant standard on the Web used to encode data. In order to retrieve data from the Web, query languages are needed. A well known standard for querying XML data is the language XQuery where data 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, which is inspired by logic programming. Further development of Xcerpt is one of the objectives of the Network of Excellence REWERSE[1]. This thesis is a contribution to this effort.

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 programmers. On the other hand, experience with untyped programming languages, like Prolog, shows how lack of typing makes many simple errors difficult to discover. Type systems enforce 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 or by using specialized run-time data structures. 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 issue is an additional effort which usually must be put to annotate programs with type information. However, the price seems to be worth to pay.

[1] `http://rewerse.net`

Xcerpt has no underlying type system nor any provision for taking advantage of type information. On the other hand, type information about XML data is often available, expressed by some schema language like DTD, XML Schema or RELAX NG. 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 results of Xcerpt programs given a type of external data to be queried, (2) checking type correctness of programs. In this way we make possible discovery of type errors in Xcerpt programs. As the system can be also used for finding dependencies between rules in programs, another its application is employing it for improving efficiency of program evaluation. As the inferred types approximate the semantics of the program (and of particular rules in the program), they can be used for documentation purposes.

Development of the Web is going in the direction where data is enhanced with semantic information. That is why ontologies, which are used for this purpose, play more and more important role on the Web. As a result XML data on the Web is going to be often related to ontologies. All this implies a need to query XML data with respect to the ontological information. For instance, one may want to filter XML data returned by a structural query by reasoning on semantic annotations included therein. Thus the idea is to enhance structural querying of XML data with ontology reasoning.

The objective of the thesis is to present how types, both syntactic (XML schemata) and semantic (ontologies) can be employed in querying XML data, using Web query language Xcerpt.

The Approach

Syntactic Types. This thesis presents a type system for a substantial fragment of the Web and Semantic Web query language Xcerpt [17, 18, 16, 10]. The considered fragment includes basic and the most important constructions of Xcerpt. (A way how most of the neglected constructions from Xcerpt can be handled is presented less formally.) We provide a formal semantics of the fragment of Xcerpt we deal with. The semantics (partially presented earlier in [11, 32, 56]) is substantially simpler than that of a full Xcerpt [50] (as it does not use the notion of simulation unification), and may be of separate interest. Similarly to other work related to Xcerpt [51, 50] we use data terms as an abstraction of semi-structured data [4] 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. We do not deal with data terms representing graphs that are not trees.

In our approach types are sets of data terms. To specify them we use a formalism of Type Definitions [58, 15]. Type Definitions are similar to unranked tree automata [13] (and equivalent formalisms), but deal also with the case of unordered children of a tree node. We adapt a restriction on Type Definitions which allows efficient algorithms for primitive operations on types, such as checking type inclusion. Another restriction is introduced to make the formalism closed under intersection.

Type Definitions define sets 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 sets of XML documents. Thus, we neglect such features of schema languages as 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 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, 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 thesis deals with a static type system. Static typing means that type errors are detected before a program execution. 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)[2]. 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. The correctness means here that if the program is applied to data from the specified database type then its results are within the specified result type. 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 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 given one.

The main part of our type system – type inference for a single Xcerpt rule – is defined by means of derivation rules. The rules abstract from lower level details and may be seen as an abstraction of an algorithm for type inference. The rules are similar to proof rules of logic, rules used in operational semantics [47], and those used in prescriptive typing [20]. 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 with regard to the formal semantics of Xcerpt. A former version of the type inference rules for a single Xcerpt rule has been published in [11] and a former version of the soundness proof is presented in [12].

Type inference for single Xcerpt rules is extended to typing Xcerpt pro-

[2]In contrast to descriptive typing, prescriptive typing is related to a typed programming language for which types are important part of its semantics.

grams, including recursive ones. We prove soundness and termination of the method. The thesis also shows how the presented type inference method can be used for discovering type based dependencies between rules in Xcerpt programs. Computing rule dependencies is essential from a point of view of efficient evaluation of programs [35].

Our approach was inspired by the work [49, 28] 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. Regular term grammars were used as a specification formalism.

Semantic Types. Ontology classes, similarly as types, are sets of objects. However, in contrast to types used in the presented type system, ontologies classify objects wrt. their meanings and not wrt. the syntax of expressions representing them. Thus the classes defined by ontologies can be called semantic types (to distinguish them from syntactic types from the previous section).

We propose two ways of taking into account semantic types when querying XML data with Xcerpt. Both of them require a way of communication with an ontology reasoner. For this purpose we use DIG interface [8] which is an API for description logic systems. It is a standard interface to ontology reasoners, supported by e.g. RacerPro[3] and Pellet[4]. Using DIG, clients can communicate with a reasoner through the use of XML encoded messages which express queries to the reasoner and replies of the reasoner.

The type system from the previous section can be extended by semantic types. The extension should make possible not only checking correctness of Xcerpt rules wrt. syntactic types (defined e.g. by XML schemata) but also wrt. semantic types defined by ontologies. For example the system may find inconsistencies, like a requirement that a value must be both of class male and female. When some operations on semantic types are needed, an ontology reasoner can be employed, for example for computing the intersection of classes.

Another idea of using semantic types is to enhance structural querying of XML data with ontology reasoning. We propose augmenting Xcerpt with ontology querying. The communication between Xcerpt and an ontology reasoner is based on DIG interface. As XML is used to encode DIG messages, the messages can be handled by Xcerpt in a natural way, similarly to any other XML data. No restrictions are imposed on the Xcerpt language and ontology queries expressible in DIG. In particular, ontologies are queried with arbitrary, not only Boolean, queries. The extended language, called DigXcerpt, is easy to implement on the top of an Xcerpt implementation

[3]`http://www.racer-systems.com/`

[4]`http://www.mindswap.org/2003/pellet/`

and a reasoner with DIG interface, without any need of modifying them. This work and its former versions has been presented in [52, 29, 30, 31].

Main Results

The contributions of the thesis related to syntactic types are:

- A formal semantics of a fragment of Xcerpt. The semantics is substantially simpler than that of full Xcerpt [50], mainly because it does not use a sophisticated notion of simulation unification. Former versions of the semantics were introduced earlier in the joint papers [58, 11, 32].
- Slight generalization of the formalism of Type Definitions (introduced in [15]). We have presented efficient algorithms for performing basic operations on types. In particular the algorithm of type inclusion is adapted from [15], and the algorithms for checking type emptiness and computing type intersection are adapted from our former work [59, 56].
- A type inference method for Xcerpt programs. The method is formally presented and proved correct. This is separated into two stages: typing of a single Xcerpt rule and typing of a program. We also discuss exactness of type inference. In general the inferred type is a superset of the set of possible results (of the considered Xcerpt program or rule, applied to data of the specified type). We provide conditions implying that the inferred type is exactly the set of possible results. We also suggest how to generalize the type inference method for several Xcerpt constructs which are outside of the Xcerpt fragment formally dealt with in this work.
- A method for checking type correctness of Xcerpt programs, given a type of the database and a type of expected results. A successful check is a proof that the program is type correct. In cases where type inference is exact, a negative result of the check is equivalent to type incorrectness of the program. In a general case, a failed check is only a suggestion that the program may be incorrect. We distinguish a few kinds of errors, and discuss how they can be discovered with help of our typing approach.
- A method of using the inferred types of rules to approximate rule dependencies.
- An implementation of a type checker for Xcerpt [57].

The contributions related to semantic types are:

- DigXcerpt – an Xcerpt extension which in addition to querying XML data allows to query ontologies. We present syntax and semantics of the extended language, and a way of its implementation together with a soundness proof. A prototype implementation of the language DigXcerpt is under development.

- An idea of an extension of the static type system with semantic types.

The Structure of the Thesis

Chapter 2. Background. The chapter introduces the query language Xcerpt and presents formal semantics of the fragment of the language we deal with. It also presents a short introduction to the major XML schema languages (DTD, XML Schema, Relax NG) and a major XML query language XQuery together with its type system. An introduction to DIG interface is also presented.

Chapter 3. Type Specification. The chapter 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. Reasoning about Types. Here a descriptive type system for Xcerpt is described. First, type inference for a single Xcerpt rule is presented by means of syntax-driven typing rules. From this abstract form, a concrete algorithm is derived later on in Section 4.2.5. Section 4.2.6 suggests a generalization for some Xcerpt constructs not dealt with by the formal semantics used in the thesis.

Based on the method for single rules, a type inference method for Xcerpt programs is introduced. Then sufficient conditions for exactness of type inference are shown. It is also discussed how the types inferred by the type system can be used to approximate rule dependencies in programs, and how errors in a program are related to the results of type inference and type checking. Additionally, the chapter provides a comparison of the presented type system with a type system of XQuery.

Chapter 5. Type System Prototype. The chapter 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. Use Cases. The chapter illustrates the use of the type system on example Xcerpt programs.

Chapter 7. Semantic Types. The chapter presents DigXcerpt, an extension of Xcerpt allowing ontology queries. It introduces syntax and semantics of the extended language and provides examples of programs showing possible applications of the extended language. It also presents a way of implementation of DigXcerpt together with its soundness proof. Additionally, it briefly discusses adding semantic types to Type Definitions.

Chapter 8. Conclusions. The chapter provides summary of the work presented in this thesis.

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

Appendix B. Typechecker Results. The chapter contains results produced by the type checker prototype applied to the examples from Chapter 6.

Chapter 2

Background

The chapter presents techniques used later on in the thesis or related to the topic of the thesis. First, it presents Xcerpt, an XML query language which is essential for the thesis. Then the main XML schema languages and a major XML query language XQuery are presented. Finally, the chapter provides a brief introduction to DIG interface which is used in our extension of Xcerpt.

2.1 Introduction to Xcerpt

This section introduces Xcerpt [50, 19, 51, 36], a rule-based query and transformation language for XML. We start with an informal introduction to Xcerpt and then we present formally semantics of a substantial fragment of the language.

2.1.1 Language Overview

An Xcerpt program is a set of rules. The body of a rule is a query intended to match data terms. If the query contains variables such matching results in answer substitutions for variables. The head of a rule uses the results of matching to construct new data terms. The queried data is either specified in the body or is produced by rules of the program. There are two kinds of rules: goal rules produce the final output of the program, while construct rules produce intermediate data, which can be further queried by other rules. Their syntax is as follows:

```
GOAL            CONSTRUCT
  head            head
FROM            FROM
  body            body
END             END
```

Usually we will denote the rules as *head* $\leftarrow$ *body* neglecting distinction between goal and construct rules.

Data terms

XML data is represented in Xcerpt by data terms. 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: *XML elements* 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.

Data terms are built from **basic constants** and labels using two kinds of parentheses: brackets [ ] and braces { }. Basic constants represent basic values such as attribute values and text. A label represents an XML element name. The parentheses following a label enclose a sequence of data terms (its direct subterms). Brackets are used to indicate that the direct subterms are ordered (in the order of their occurrence in the sequence), while braces indicate that the direct subterms are unordered. The latter alternative is used to encode attributes of an XML element by a data term of the form $attr\{l_1[v_1], \ldots, l_n[v_n]\}$ where l_i are names of the attributes and v_i are their respective values. To show how XML elements are represented by data terms, consider an XML element

$$E = <name\ attr_1{=}value_1 \cdots attr_1{=}value_k>E_1 \cdots E_n</name>,$$

$(k \geq 0, n \geq 0)$ where each E_i (for $i = 1, \ldots, n$) is an element or a text and for any text element the previous and the next element is not text.

E is represented as a data term $name[\,attributes,\ child_1, \cdots, child_n\,]$, where the data terms $child_1, \ldots, child_n$ represent $E_1, \ldots, E_n$, and the data term

$$attributes = attr\{\, attr_1[value_1], \cdots, attr_k[value_k]\,\}$$

is optional and represents the attributes of E. The subterms representing attributes are not ordered and this is denoted by enclosing them by braces. We assume that there is no syntactic difference between XML element names and attribute names and they both are labels of nodes in our mixed trees (and symbols of our data terms).

Example 1. *This is an XML element and the corresponding data term.*

```
<CD price="9.90">                  CD[ attr{ price[ "9.90" ] },
  <title>Empire</title>                title[ "Empire Burlesque" ],
  <artist>Bob Dylan</artist>           artist[ "Bob Dylan" ]
</CD>                              ]
```

□

Usually data terms are used to represent tree structured data. However, using a reference mechanism, they can be used to represent graphs. A construction of the form *oid*@*t* associates an identifier *oid* with a data term *t*. The identified data term can be referred to by a construction $\uparrow oid$. References in data terms correspond to the various linking mechanisms available for XML such as ID/IDREF, XPointer, URIs, etc. In contrast to other query languages, Xcerpt automatically dereferences references in data terms. For example, a data term $f[\,b[\,o_1@d[\,],\ c[\,\uparrow o_1\,]\,]$ is equivalent to a data term $f[\,b[\,d[\,],\ c[\,d[\,]\,]\,]$. Thus any query yields the same answers for both data terms.

As in XML, data terms may use namespaces. For instance, a data term *a:address-book*{ *a*:*person*{ *a*:*name*{"*John Smith*" } } } uses a namespace prefix *a*. Association of a namespace prefix with a URI is done in Xcerpt programs using a keyword `ns−prefix`. For example

`ns−prefix` *a* = "*http://www.myschemas.org/address-book*"

specifies a URI for the prefix *a*.

Query terms

Query terms are (possibly incomplete) patterns which are used in a rule body (query) to match data terms. In particular, every data term is a query term. Query terms can be ordered or unordered patterns, which is denoted by two kind of parentheses: brackets and braces, respectively. Query terms with double brackets or braces are incomplete patterns. 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 $d = l'\alpha' d_1, \ldots, d_n\beta'$, 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 {}) mean that $m = n$ and each q_i should match some (distinct) d_j. Double parentheses mean that $m \leq n$ and $q_1, \ldots, q_m$ are matched against some m terms out of $d_1, \ldots, d_n$. Curly braces ({} or {{}}) 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. For example, a query term *a*["*c*", "*b*"] is an ordered pattern and it matches neither a data term *a*["*b*", "*c*"] nor *a*{ "*b*", "*c*" }. A query term *a*{ "*b*", "*c*" }, which is an unordered pattern, matches both *a*["*b*", "*c*"] and *a*{ "*b*", "*c*" }. A query term *a*[["*b*", "*d*"]], which is an incomplete pattern, matches a data term *a*["*b*", "*c*", "*d*"]. However, a query term *a*[["*b*", "*d*"]] does not match a data term *a*["*d*", "*b*", "*c*"] because of a different order of subterms. In contrast a query term *a*{{ "*b*", "*d*" }} matches *a*["*d*", "*b*", "*c*"].

To specify subterms at arbitrary depth a keyword `desc` is used: `desc` *q* matches a data term *d* whenever *q* matches some subterm of *d*. For example, a query term `desc` "*d*" matches a data term *a*[*b*["*d*"], "*c*"].

Generally query terms may include variables[1] so that a successful matching binds variables of a query term to data terms. Such bindings are called answer substitutions. A result of a query term matching a data term is a set of answer substitutions as there may be more than one possible answer substitution for a query term. A variable matches any data term. To restrict a set of data terms matched by a variable a construction $X \rightsquigarrow q$ can be used. The construction allows the variable to be bound only to the data terms which are matched by the query term q. For example, $a[$"b", $X \rightsquigarrow$ `desc` "c"$]$ matches a data term $a[$"b", $e[$"c"$]]$ and the matching results in a answer substitution set consisting of a single substitution $\{X/e[$"c"$]\}$. In contrast, the query term does not match a data term $a[$"b", $e[$"d"$]]$.

Variables can be used to match labels of data terms. For example a query term $X[$"b"$]$ matches a data term $a[$"b"$]$ and as the result the variable X is bound to the label a. Query terms may also use namespace variables to bind them to namespace URIs.

A query term may specify an optional subterm using an expression of the form `optional` t. If matching of a query term t against some subterm succeeds then variable bindings for variables in t are obtained. Otherwise, the evaluation of the query term containing optional t does not fail, but it does not yield any bindings for the variables in t.

To specify subterms at a given position within a data term d a query term of the form `position` $n\,q$ can be used. A data term matched by the query term q must occur at the position specified by n in a sequence of direct subterms of d. For example, as a result of matching a query term $tr[[$`position` $2\ td[X]]]$ against a data term $tr[td[$"$sugar$"$], td[$"12.90"$]]$ the variable X obtains a value "12.90". Xcerpt allows also to use a position variable as a parameter n. For instance, the previous data terms is matched by a query term $tr[[$`position` $X\ td[$"$sugar$"$]]]$ and the variable X obtains a value 1.

Query terms may contain subterm negation which allows to express that a data term must not contain subterms matching a certain query term. For example, a query term $a\{\{$"b", "d", `without` "c"$\}\}$ matches a data term $a[$"b", "d", "e"$]$ but it does not match a data term $a[$"b", "c", "d"$]$. Subterm negation is only reasonable in query terms being incomplete patterns.

Query terms may use regular expressions for text processing. The regular expressions are based on POSIX syntax [39] and can be used either in a place of strings or in a place of subterm labels in query terms. The regular expressions are enclosed by $//$. For example, a query term $name\{\{/.{*}son.{*}/\}\}$ matches any data term with a label $name$ and a subterm being a string containing a substring son.

Bodies of Xcerpt rules are queries. Queries are constructed from query terms using logical connectives such as `or`, `and`, and `not`. Furthermore, queries may be associated with external resources storing XML data or data

[1]To simplify notation we will usually denote variables by symbols consisting only of capital letters and we will skip the keyword `var` used in Xcerpt to denote variables.

terms. This is done by a construct of the form `in`[r, Q]. Its meaning is that a query Q is to be evaluated against data specified by a URI r. Queries in the body of a rule which have no associated resources are matched against data generated by rules of the Xcerpt program. The logical connective `or` is used in an expression of the form `or`$\{Q_1, \ldots, Q_n\}$, which results in a union of the answer substitution sets obtained for queries $Q_1, \ldots, Q_n$. An expression `and`$\{Q_1, \ldots, Q_n\}$ results in a set of answer substitutions where each substitution is a union of single substitutions obtained for each $Q_1, \ldots, Q_n$ and binding variables to the same values[2].

A query can be preceded with a keyword `not` which expresses query negation. This is negation as failure like in logic programming i.e. a negated query `not` Q succeeds if the query Q fails. As variables occurring in a negated query do not yield bindings they must occur also in a rule body outside of the negated query.

Queries can be augmented with non structural conditions using an expression of the form Q `where` $\{Condition\}$. *Condition* is a comparison operation using operators such as $>, =, \leq$, etc. The *Condition* expression may use variables but only those which occur in the query Q. It may also use arithmetic expressions.

Construct terms are used in rule heads to construct new data terms. They are similar to data terms, but may contain variables. Data terms are constructed out of construct terms by applying answer substitutions obtained from a rule body. Construct terms may also use grouping constructs `all` and `some` to collect all or, respectively, some instances that result from different variable bindings. The grouping constructs may be accompanied by an expression `group by` which is used to group results by the variables whose values should not appear in the results. Another expression which may follow the grouping constructs is `order by`. The grouping constructs create sequences of data terms in arbitrary order and the expression `order by` can be used to specify the order.

Construct terms may contain functions and aggregations. Functions, such as *add, mult, sub*, etc., use a fixed number of arguments. Aggregations, such as *sum, avg, min*, use a variable number of arguments and their arguments may contain grouping constructs.

Construct terms, similarly as query terms, may use optional subterms. Such subterms may contain variables which remain unbound after evaluation of a body of a rule (e.g. as they appear only in optional query terms). An optional construct term is preceded with a keyword `optional` and may be followed by a default value specified by a keyword `with default`. During an evaluation, if any of the variables of an optional construct term is unbound the construct term is omitted, or if a default value is specified, the construct term is replaced by the default value.

A construct term c in a goal rule head may be associated with a resource

[2]The expressions `or` and `and` can be used also with square brackets in order to enforce a specific evaluation order of the queries.

r to which the goal results are written. This is done by a construction of the form $out[r, c]$. If a head of a goal rule is a construct term which is not associated with a resource the results of the rule are directed to the standard output.

Example 2. *Consider an XML document* recipes.xml*, which is a collection of culinary recipes. The document is represented by a data term:*

```
recipes[
  recipe[ name["Recipe1"],
    ingredient[ name["sugar"], amount[ attr{unit["tbsp"]}, 3 ] ],
    ingredient[ name["orange"], amount[ attr{unit["unit"]}, 1 ] ] ],
  recipe[ name["Recipe2"],
    ingredient[ name["flour"], amount[ attr{unit["dl"]}, 3 ] ],
    ingredient[ name["salt"], amount[ attr{unit["ml"]}, 1 ] ] ],
  recipe[ name ["Recipe3"],
    ingredient[ name["spaghetti"], amount[ attr{unit["kg"]}, 0.5 ] ],
    ingredient[ name["tomato"], amount[ attr{unit["kg"]}, 0.4 ] ] ] ]
```

The Xcerpt rule queries the document and extracts the names of the recipes:

```
  GOAL
    recipe-names [ all var R ]
  FROM
    in[ "file:recipes.xml", recipes[[ recipe [[ name[ var R ] ]] ]] ]
  END
```

Evaluation of the rule results in the answer substitutions: $\{R/"Recipe1"\}$, $\{R/"Recipe2"\}, \{R/"Recipe3"\}$*. Thus, the result returned by the rule is:*

```
recipe-names[ "Recipe1", "Recipe2", "Recipe3" ]
```
□

2.1.2 Formal Semantics

The section provides a formal semantics of a fragment of Xcerpt containing basic and the most important Xcerpt constructions. The semantics (partially presented earlier in [11, 56]) is substantially simpler than that of a full Xcerpt [50] as it does not use the notion of simulation unification (a process of matching terms). Another difference is that our data terms represent trees while in full Xcerpt terms are used to represent graphs. Other Xcerpt features not dealt with are: functions and aggregations, non-pattern conditions, optional subterms, position specifications, negation, regular expressions and label variables.

Now we formally define various constructs of Xcerpt.

Data Terms

As it was mentioned before, data terms are used to represent XML data. XML element names are represented in data terms as labels. The infinite alphabet of labels will be denoted by $\mathcal{L}$. Basic constants represent 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 $l[t_1, \ldots, t_n]$ and $l\{t_1, \ldots, t_n\}$ 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 $l'[t_1, \ldots, t_n]$ and of $l'\{t_1, \ldots, t_n\}$. Data terms $t_1, \ldots, t_n$ will be sometimes called the arguments of l', or the **direct subterms** of $l'[t_1, \ldots, t_n]$ (and of $l'\{t_1, \ldots, t_n\}$). The **root** of a data term t, denoted $root(t)$, is defined as follows . If t is of the form $l[t_1, \ldots, t_n]$ or $l\{t_1, \ldots, t_n\}$ then $root(t) = l$; for t being a basic constant we assume that $root(t) = \$$.

Query Terms

Here we formally define a query term:

Definition 2. Query terms *are inductively defined as follows:*

- *Any basic constant is a query term.*
- *Any variable 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 $X \leadsto q$ 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 $q = l\alpha q_1, \ldots, q_n\beta$, where $\alpha\beta$ are parentheses [ ], [[ ]], {} *or* {{}}, *$root(q) = l$ 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 $X \leadsto q$ are $X \leadsto q$ and all the subterms of q.

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 $dom(\theta)$ is $\{X_1, \ldots, X_n\}$, its application to a (query) term is defined in a standard way.

Definition 3 ([58]). *A substitution θ 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 $d, d_1, \ldots$ data terms; set notation is used for multisets, for instance $\{d, d\}$ and $\{d\}$ are different multisets).*

q	d	condition on q and d
b	b	b is a basic constant
$l[q_1, \ldots, q_n]$	$l[d_1, \ldots, d_n]$	θ is an answer for q_i and d_i, for each $i = 1, \ldots, n$
$l[[q_1, \ldots, q_m]]$	$l[d_1, \ldots, d_n]$	for some subsequence $d_{i_1}, \ldots, d_{i_m}$ of $d_1, \ldots, d_n$ (i.e. $0 < i_1 < \ldots < i_m \leq n$) θ is an answer for q_j and d_{i_j}, for each $j = 1, \ldots, m$,
$l\{q_1, \ldots, q_n\}$	$l\{d_1, \ldots, d_n\}$ or $l[d_1, \ldots, d_n]$	for some permutation $d_{i_1}, \ldots, d_{i_n}$ of $d_1, \ldots, d_n$ (i.e. $\{d_{i_1}, \ldots, d_{i_n}\} = \{d_1, \ldots, d_n\}$) θ is an answer for q_j and d_{i_j} for each $j = 1, \ldots, n$,
$l\{\{q_1, \ldots, q_m\}\}$	$l\{d_1, \ldots, d_n\}$ or $l[d_1, \ldots, d_n]$	for some $\{d_{i_1}, \ldots, d_{i_m}\} \subseteq \{d_1, \ldots, d_n\}$ θ is an answer for q_j and d_{i_j} for each $j = 1, \ldots, m$,
X	d	$X\theta = d$
$X \leadsto q$	d	$X\theta = d$ and θ is an answer for q and d
`desc` q	d	θ is an answer for q and some subterm d' of d

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 $root(q) \neq root(d)$ then no answer for q, d exists. If $q = d$ then any θ is an answer for q, d. A query $l\{\{\}\}$ matches any data term with the label l. If θ, θ' are substitutions and $\theta \subseteq \theta'$ then if θ is an answer for q, d then θ' is an answer for q, d. If a variable X occurs in a query term q then queries $X \rightsquigarrow q$ and $X \rightsquigarrow \texttt{desc}\, q$ match no data term, provided that $q \neq X$ and q is not of the form $\texttt{desc} \cdots \texttt{desc}\, X$.

Each answer for a query term q binds all the variables of the query to some data terms. For any such answer θ' (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. Redundant answers allow for a simpler definition of an answer for a query.

Example 3. *Consider a data term* $d = a[\,b[\,\text{"c"}\,]\,]$ *and query terms* $q_1 = a[\,X\,]$ *and* $q_2 = a[\,b[\,Y\,]\,]$*. An answer* $\theta = \{\,X/b[\,\text{"c"}\,],\, Y/\text{"c"}\,\}$ *is a redundant answer for both query terms and d. A non redundant answer for* q_1 *and d is* $\theta_1 = \{\,X/b[\,\text{"c"}\,]\,\}$ *and a non redundant answer for* q_2 *and d is* $\theta_2 = \{\,Y/\text{"c"}\,\}$*.*

Queries

A query is a connection of zero or more query terms using the connectives `and` and `or`. It may furthermore be associated with resources against which the query terms are evaluated.

A **targeted query term** is a pair $\texttt{in}(r, q)$, of a URI and a query term[3]. We assume that a URI r locates on the Web a data term $\delta(r)$.

Now we formally define a query and an answer for a query and a set of data terms.

Definition 4. *Let Z be a set of data terms. A* **query** *and an* **answer substitution** (*shortly, an* **answer**) *for a query and a set of data terms is inductively defined as follows.*

- *Any query term q is a query. A substitution* θ *is an answer substitution for q and Z iff* θ *is an answer substitution for q and some* $d \in Z$*.*

- *Any targeted query term* $\texttt{in}(r, q)$ *is a query. An substitution* θ *is an answer substitution for* $\texttt{in}(r, q)$ *and Z iff* θ *is an answer substitution for q and* $\delta(r)$*.*

- *If* $Q_1, \ldots, Q_n$ $(n \geq 0)$ *are queries then* $\texttt{and}(Q_1, \ldots, Q_n)$ *and* $\texttt{or}(Q_1, \ldots, Q_n)$ *are queries.*

  A substitution θ *is an* answer substitution *for* $\texttt{and}(Q_1, \ldots, Q_n)$ *(respectively for* $\texttt{or}(Q_1, \ldots, Q_n)$*) and Z iff* θ *is an answer substitution for each of* (*some of*) $Q_1, \ldots, Q_n$ *and Z.*

[3]In Xcerpt syntax, the parameters of targeted query terms, similarly as the arguments of `and` and `or`, are enclosed by brackets or braces. Here we decided to use normal parenthesis instead to make these construct distinct from query terms. Also, from the point of view of our semantics, there is no need to have two kinds of parenthesis in this case.

It follows from the definition that if $Z \subseteq Z'$ and θ is an answer for Q and Z then θ is an answer for Q and Z'.

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

A query term q occurring in a query Q is a **top query term** if it is standalone in Q i.e. it is not a part of a (targeted) query term. For example, $a[\,b[\,X\,]\,]$ is the only top query term of the query $\texttt{and}(\ \texttt{in}(\,"\textit{file:example.xml}",\ c[\,X\,]\,),\ a[\,b[\,X\,]\,]\,)$.

An answer θ (for a query Q and set Z) will be called redundant if it binds a variable which does not occur in Q. Similarly to the case of query terms, for any such answer θ there exists a non redundant answer $\theta' \subseteq \theta$ for Q and Z.

A query can be transformed into an equivalent one in a *disjunctive normal form* $\texttt{or}(Q_1, \ldots, Q_n)$, where each Q_i is of the form $\texttt{and}(Q_{i1}, \ldots, Q_{ik_i})$ and each Q_{ij} is a (targeted) query term (cf. [50, Proposition 6.4]).

Proposition 1. *Let Q be a query, Z a set of data terms and Θ be the set of answers for Q and Z. If Q' is a disjunctive normal form of Q then Θ is the set of answers for Q' and Z.*

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 $\texttt{and}(Q_1, \texttt{or}(Q_2, Q_3))$ and $\texttt{or}(\texttt{and}(Q_1, Q_2), \texttt{and}(Q_1, Q_3))$ are equivalent formulas, and by Definition 4 they have the same set of answers. □

Construct Terms

Construct terms are used in constructing data terms which are results of query rules.

Definition 5. *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*

$$b,\quad X,\quad l[c_1, \ldots, c_n],\quad l\{c_1, \ldots, c_n\},\quad \texttt{all}\ c,\quad \texttt{some}\ k\ c,$$

are construct terms. $FV(b) = \emptyset$, $FV(X) = \{X\}$, $FV(l[c_1, \ldots, c_n]) = FV(l\{c_1, \ldots, c_n\}) = \bigcup_{i=1}^{n} FV(c_i)$, $FV(\texttt{all}\ c) = FV(\texttt{some}\ k\ c) = \emptyset$. Construct terms of the form $l[c_1, \ldots, c_n]$ and $l\{c_1, \ldots, c_n\}$ are called rooted construct terms. The constructs `all` *and* `some` *are called* **grouping constructs**.

Notice that any data term is a construct term. (Also, a construct term without any grouping construct is a query term).

Query Rules

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

An application of a query to a set of data terms may result in many answer substitutions. Thus we will use a notion of a substitution set which is a set of substitutions of data terms for variables. In order to handle properly the grouping constructs in construct terms we also need an equivalence relation on answer substitutions.

Definition 6. *Given a substitution set Θ and a set V of variables, such that $V \subseteq dom(\theta)$ 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. A result of an application of an answer substitution set to a construct term is defined as follows.

Definition 7. *Let c be a construct term and Θ 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 Θ 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 $\theta \in \Theta$*
- $\Theta(l\{c_1, \ldots, c_n\}) = l\{\Theta(c_1) \circ \cdots \circ \Theta(c_n)\}$
- $\Theta(l[c_1, \ldots, c_n]) = l[\Theta(c_1) \circ \cdots \circ \Theta(c_n)]$
- $\Theta(\texttt{all } c') = \Theta_1(c') \circ \cdots \circ \Theta_k(c')$, *where* $\{\Theta_1, \ldots, \Theta_k\} = \Theta/_{\simeq FV(c')}$
- $\Theta(\texttt{some } k\ c') = \Theta_1(c') \circ \cdots \circ \Theta_m(c')$, *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 Θ like in the definition above and 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'.

Now we are ready to define a query rule and a result of a query rule applied to a set of data terms. The set of results of a query rule is determined by the external resources $\delta(r_i)$ referred to in a body of the rule and the set of data terms Z produced by query rules of a program.

Definition 8. *A* **query rule** *(shortly,* 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* $\mathtt{or}(Q_1, \ldots, Q_n)$ *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 Θ *is the set of all answers for* Q *and a set of data terms* Z*, and* $\Theta' \in \Theta/_{\simeq_{FV(c)}}$ *then* $\Theta'(c)$ *is a* **result** *for a query rule* $c \leftarrow Q$ *and* Z*. The* **set of results** *of* $p = c \leftarrow Q$ *and* Z *is denoted as* $res(p, Z)$*.*

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.

In the definition above Θ is the set of all answers for Q and a set of data terms Z. However, it is sufficient to consider only the set of non redundant answers for Q and Z.

Example 4. *Consider a set* Z *consisting of data terms:*

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"]]

The following rule queries the data terms from Z *and extracts titles and artists of the CD's issued in 1988:*

res[name[TITLE], author[ARTIST]]
← cd{title[TITLE], artist[ARTIST], year["1988"]}

Evaluation of the body of the rule results in the following set of non redundant answers: $\Theta = \{\theta_1, \theta_2\}$*, where*

$$\begin{aligned}\theta_1 &= \{TITLE/\text{"Hide your heart"}, ARTIST/\text{"Bonnie Tyler"}\},\\ \theta_2 &= \{TITLE/\text{"Stop"}, ARTIST/\text{"Sam Brown"}\}.\end{aligned}$$

Thus the set of equivalence classes $\Theta/_{\simeq_{\{TITLE,ARTIST\}}} = \{\{\theta_1\}, \{\theta_2\}\}$ *and the results of the rule and the set of data terms* Z *are:*

res[name["Hide your heart"], author["Bonnie Tyler"]]
res[name["Stop"], author["Sam Brown"]]

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.*

results[`all` *res[cds[year[YEAR],* `all` *name[TITLE]]]]*
← cd{{title[TITLE], year[YEAR]}}

The evaluation of the body of the rule against the set of data terms Z results in the set of non redundant answers: $\Theta' = \{\theta'_1, \theta'_2, \theta'_3\}$, where

$$\begin{array}{lcl} \theta'_1 & = & \{\mathit{TITLE}/\text{"Empire Burlesque"}, \mathit{YEAR}/\text{"1985"}\}, \\ \theta'_2 & = & \{\mathit{TITLE}/\text{"Hide your heart"}, \mathit{YEAR}/\text{"1988"}\}, \\ \theta'_3 & = & \{\mathit{TITLE}/\text{"Stop"}, \mathit{YEAR}/\text{"1988"}\}. \end{array}$$

The set of free variables of the construct term being the head of the query rule is empty and the set of equivalence classes $\Theta'/_{\simeq_\emptyset} = \{\{\theta'_1, \theta'_2, \theta'_3\}\}$. The set of free variables of the construct term being the argument of the first `all` *construct is $\{\mathit{YEAR}\}$ and the set of equivalence classes $\Theta'/_{\simeq_{\{YEAR\}}} = \{\{\theta'_1\}, \{\theta'_2, \theta'_3\}\}$. Thus the rule returns the following result:*

$$\begin{array}{ll} \mathit{results}[& \mathit{res}[\mathit{cds}[\mathit{year}[\text{"1985"}], \mathit{name}[\text{"Empire Burlesque"}]]], \\ & \mathit{res}[\mathit{cds}[\mathit{year}[\text{"1988"}], \mathit{name}[\text{"Hide your heart"}], \mathit{name}[\text{"Stop"}]]]\ ] \end{array}$$

Programs

Here we present further definitions related to Xcerpt programs.

Definition 9. *An Xcerpt* **program** *$\mathcal{P}$ 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***.*

Now we describe the effect of applying a set of rules to a set of data terms, and then the semantics of a program.

Definition 10 (Immediate consequence operator for rule results)**.** *Let P be a set of Xcerpt query rules. R_P is a function on sets of data terms such that $R_P(Z) = Z \cup \bigcup_{p \in P} res(p, Z)$.*

Notice that if no grouping constructs appear in the rules from P then $R_P(Z)$ is monotonic.

Definition 11 (Rule result, no grouping constructs)**.** *Let $\mathcal{P} = (P, G)$ be an Xcerpt program without grouping constructs and $P' = P \backslash G$. Given fixed data terms $\delta(r_j)$ associated with external resources occurring in the rules from P, a data term d is a* **result of a rule** *p in P if $d \in res(p, R^i_{P'}(\emptyset))$ for some $i \geq 0$.*

A **result of a program** *$\mathcal{P}$ is a data term which is a result of a goal of $\mathcal{P}$.*

Example 5. *Let $P' = \{p\}$, where $p = c[X] \leftarrow$* `or`*$(X,$* `in`*$(r, b[X]))$ and $\delta(r) = b[\text{"a"}]$. The set of results of p is infinite and it contains subsets $R_{P'}(\emptyset) = res(p, \emptyset) = \{c[\text{"a"}]\}$, $R^2_{P'}(\emptyset) = res(p, R_{P'}(\emptyset)) = \{c[\text{"a"}], c[c[\text{"a"}]]\}, \ldots, R^i_{P'}(\emptyset) = res(p, R^{i-1}_{P'}(\emptyset)) = \{c[\text{"a"}], \ldots, c^i[\text{"a"}]\}$, for $i > 0$.*

□

Example 6. *This example is an extension of the "Clique of Friends" example from [50]. Consider an XML document* addrBooks.xml *with a collection of address books where each address book has its owner and a set of entries with information about people the owner knows. The information contains an annotation about the relation between the owner and the particular person such as: friend, colleague, family. The document is represented by a data term:*

$$
\begin{array}{l}
\textit{addr-books}[\\
\quad \textit{addr-book}[\ \textit{owner}[\text{"}\textit{Donald Duck}\text{"}\,], \\
\qquad\qquad \textit{entry}[\ \textit{name}[\text{"}\textit{Daisy Duck}\text{"}\,], \\
\qquad\qquad\qquad \textit{relation}[\text{"}\textit{friend}\text{"}\,], \\
\qquad\qquad\qquad \textit{phoneNo}[\text{"+112345"}\,], \\
\qquad\qquad\qquad \textit{address}[\ \textit{street}[\text{"}\textit{Hayes 51}\text{"}\,], \\
\qquad\qquad\qquad\qquad \textit{zip-code}[\text{"21213"}\,], \\
\qquad\qquad\qquad\qquad \textit{city}[\text{"}\textit{Los Angeles}\text{"}\,], \\
\qquad\qquad\qquad\qquad \textit{country}[\text{"}\textit{USA}\text{"}\,]\,]\,], \\
\qquad\qquad \ldots, \\
\qquad\qquad \textit{entry}[\ldots]\,], \\
\quad \ldots, \\
\quad \textit{addr-book}[\ldots]\,]
\end{array}
$$

The following Xcerpt program extracts a relation friend of a friend (foaf) which is the transitive closure of a relation friend of (fo). The relation fo is computed by the rule p_1 and its transitive closure is computed by the recursive rule p_2. The third rule g, which is a goal, returns a data term with a sequence of pairs representing the relation foaf.

$$
\begin{array}{rcl}
p_1 & = & \textit{fo}[\,X, Y\,] \leftarrow \texttt{in}(\text{"}\textit{file:addrBooks.xml}\text{"}, \\
 & & \quad \textit{addr-books}\{\{ \\
 & & \qquad \textit{addr-book}\{\{\ \textit{owner}[X], \\
 & & \qquad\quad \textit{entry}\{\{\ \textit{name}[Y],\ \textit{relation}[\text{"}\textit{friend}\text{"}\,]\,\}\}\,\}\}\,\}\}\,) \\
p_2 & = & \textit{foaf}[\,X, Y\,] \leftarrow \texttt{or}(\textit{fo}[\,X, Y\,],\ \texttt{and}(\textit{fo}[\,X, Z\,],\ \textit{foaf}[\,Z, Y\,]\,)\,) \\
g & = & \textit{clique-of-friends}[\,\texttt{all}\ \textit{foaf}\{\,X, Y\,\}\,] \leftarrow \textit{foaf}[\,X, Y\,]
\end{array}
$$

To define semantics of programs with grouping constructs we employ a notion of static rule dependency to split programs into strata. This notion is equivalent to the rule dependency used in [50]. We also introduce a weaker kind of dependency, as the static dependency does not reflect some issues related to types.

Definition 12 (Static rule dependency)**.** *Let $\mathcal{P} = (P, G)$ be an Xcerpt program. A rule $c \leftarrow Q \in P$* **directly statically depends** *on a rule $c' \leftarrow Q' \in P \backslash G$, if a top query term from Q matches some instance of the construct term c'. The fact that a rule p directly statically depends on a rule p' is denoted as $p \succ_s p'$.*

A rule $p \in P$ **statically depends** *on a rule $p' \in P \backslash G$ if $p \succ_s^+ p'$ (where $\succ_s^+$ is the transitive closure of $\succ_s$ i.e. $p \succ_s^+ p'$ if $p \succ_s p_1 \succ_s \ldots \succ_s p_k \succ_s p'$ for some rules $p_1, \ldots, p_k$ in $P \backslash G$ where $k \geq 0$).*

Definition 13 (Weak static rule dependency)**.** *Let $\mathcal{P} = (P, G)$ be an Xcerpt program. A rule $c \leftarrow Q \in P$* **directly weakly statically depends** *(shortly, directly w-depends) on a rule $c' \leftarrow Q' \in P \backslash G$, if a top query term from Q matches some instance of the construct term c'', where c'' is c' with every occurrence of a variable replaced by a distinct variable. The fact that a rule p directly w-depends on a rule p' is denoted as $p \succ_w p'$.*

A rule $p \in P$ **weakly statically depends** *(shortly, w-depends) on a rule $p' \in P \backslash G$ if $p \succ_w^+ p'$. The program $\mathcal{P}$ is* **weakly statically recursive** *(shortly, w-recursive) if $p \succ_w^+ p$ for some $p \in P$. We also say that $P \backslash G$ is w-recursive.*

Static dependency between rules implies weak static dependency.

Example 7. *Consider the following query rules of an Xcerpt program:*

$$p_1 \;=\; a[\,Y\,] \leftarrow b[\,\text{"}d\text{"}, \text{"}e\text{"}, Y\,], \qquad p_2 \;=\; b[\,X, X, Y\,] \leftarrow c[\,X, Y\,].$$

It holds: $p_1 \succ_w p_2$ but $p_1 \not\succ_s p_2$. □

We will need the notion of weak dependency in Chapter 4 describing type inference. A simple algorithm for finding w-dependencies can be obtained by a slight modification of the typing rules for query terms that will be presented in Chapter 4. We skip the details.

Now we generalize the semantics of Definition 11 to programs with grouping constructs. The semantics is used in the proofs but not referred to explicitly in the thesis. Thus the rest of this section may be skipped at the first reading.

If a query rule p in a program contains a grouping construct, it can be executed only after all data terms queried by p have been obtained. This is ensured in the following way. The query rules of the program are divided into sets called strata. A query rule p' with a grouping construct can statically depend only on rules from a lower stratum. Hence no rule of the same stratum as p' can produce data that can be queried by p'. Moreover, for an arbitrary rule p, no rule of a higher stratum than p can produce data that can be queried by p. The rules from a given stratum are not executed until the execution of the rules from lower strata is completed.

Definition 14 (Stratification)**.** *Let $\mathcal{P} = (P, G)$ be an Xcerpt program and $P_1, \dots, P_n$ $(n \geq 0)$ be disjoint sets of query rules such that $P \backslash G = P_1 \cup \ldots \cup P_n$. The sequence $P_1, \dots, P_n, G$ is a stratification of $\mathcal{P}$ if for any pair of rules $p, p' \in P \backslash G$, if $p \succ_s^+ p'$ then $p \in P_i$ and $p' \in P_j$, where $1 \leq j \leq i \leq n$ and if p has a grouping construct in its head then $j < i$.*

Notice, that there may exist many different stratifications for a program. Any program (P, G) without grouping constructs is stratifiable and its stratification is $P \backslash G, G$. As in [50] we assume that we deal with stratifiable programs.

Example 8. *Consider a program* $\mathcal{P} = (\{p_1, p_2, p_3, p_4, g\}, \{g\})$*, where:*

$$\begin{array}{rclrcl} g &=& g[\,\texttt{all}\,X\,] \leftarrow h[\,X\,], & p_3 &=& b[\,Y\,] \leftarrow c[\,f[\,Y\,]\,], \\ p_1 &=& h[\,X\,] \leftarrow a[[\,X\,]], & p_4 &=& c[\,Y\,] \leftarrow \texttt{in}(\,r_1,\, k[[\,Y\,]]\,). \\ p_2 &=& a[\,\texttt{all}\,X\,] \leftarrow b[\,X\,], & & & \end{array}$$

Some of the possible stratifications of $\mathcal{P}$ *are sequences* $\{p_3, p_4\}, \{p_1, p_2\}, \{g\}$ *and* $\{p_3, p_4\}, \{p_2\}, \{p_1\}, \{g\}$. □

Let P be a set of rules and $k \geq 0$ be a number such that $R^k_P(\emptyset) = R^{k+1}_P(\emptyset)$. The set $R^k_P(\emptyset)$ will be denoted as $R^\infty_P(\emptyset)$.

Definition 15. *Let* $\mathcal{P} = (P, G)$ *be an Xcerpt program,* $P_1, \ldots, P_n, G$ *be a stratification of* $\mathcal{P}$*. Let* $Z_0 = \emptyset$ *and, for* $j = 1, \ldots, n$*, let* $Z_j = R^\infty_{P_j}(Z_{j-1})$*. Given fixed data terms* $\delta(r_i)$ *associated with external resources occurring in the rules from* P*, a data term* d *is a* **result of a rule** p *in* P *if* $d \in res(p, Z_n)$*. A* **result of a program** $\mathcal{P}$ *is a result of a goal rule* $p \in G$ *in* P*.*

The set of results of a program $\mathcal{P}$ *will be denoted as* $res(\mathcal{P})$*.*

According to this definition, if the program loops (i.e. $R^{i+1}_{P_j}(Z_{j-1}) \neq R^i_{P_j}(Z_{j-1})$ for some j and every $i = 1, 2, \ldots$) then $Z_j, \ldots, Z_n$ do not exist and no results exist, for any p in P. For simplicity reasons we do not provide a more sophisticated definition describing results of a looping program (i.e. those obtained before the program enters the infinite loop).

All stratifications of a given program yield the same results. We omit a justification.

Example 9. *Consider the program* $\mathcal{P}$ *from Example 8 and assume that the resource* r_1 *is associated with a data term* $k[\,f[\,\text{"}s\text{"}\,],\, f[\,\text{"}t\text{"}\,]\,]$*. We want to find the results of the program. Let* $P_1 = \{p_3, p_4\}$*,* $P_2 = \{p_1, p_2\}$ *and* $G = \{g\}$*. Then* P_1, P_2, G *is a stratification of the program and* $Z_1 = \{\, c[\,f[\,\text{"}s\text{"}\,]\,], c[\,f[\,\text{"}t\text{"}\,]\,], b[\,\text{"}s\text{"}\,], b[\,\text{"}t\text{"}\,]\,\}$*,* $Z_2 = Z_1 \cup \{\, a[\,\text{"}s\text{"},\, \text{"}t\text{"}\,], h[\,\text{"}s\text{"}\,], h[\,\text{"}t\text{"}\,]\,\}$*. The set* $res(g, Z_2)$ *of results of the program is* $\{\, g[\,\text{"}s\text{"},\, \text{"}t\text{"}\,]\,\}$*.*

□

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 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[4], Document Structure Description (DSD)[5], Examplotron[6], Schema for Object-Oriented XML (SOX)[7], Document Definition Markup Language (DDML)[8].

2.2.1 DTD

Document Type Definition (DTD) is a simple and the most popular XML schema language. It is a standard defined by the World Wide Web Consortium (W3C) [33] 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 named `element-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*,
  - a deterministic[9] regular expression over element names, which can contain the standard operators: choice "|", sequence ",", zero or more "*", one or more "+", zero or one "?". The element content is a sequence of elements such that the corresponding sequence of the element names matches the expression.

- Attribute List Declarations, which are of the form:

  $$\texttt{<!ATTLIST element−name attr−name}_1\ \texttt{attr−type}_1\ \texttt{qualifier}_1 \ \ldots\ \texttt{attr−name}_n\ \texttt{attr−type}_n\ \texttt{qualifier}_n\ \texttt{>}$$

[4]http://xml.ascc.net/resource/schematron/Schematron2000.html
[5]http://www.brics.dk/DSD/dsd2.html
[6]http://examplotron.org
[7]http://www.w3.org/TR/NOTE-SOX/
[8]http://www.w3.org/TR/NOTE-ddml
[9]The formal meaning of this requirement is that the regular expressions are 1-unambiguous in the sense of [14].

where the `element−name` is the name of the element for which the list of attributes is being defined, $\texttt{attr−name}_{\texttt{i}}$ is the the name of the i^{th} attribute being defined, $\texttt{attr−type}_{\texttt{i}}$ defines the type of data that may be used for the value. The possible types of attributes are:

- CDATA - character data,
- ENTITY - reference to an external file such as a graphic file for importing an image,
- ENTITIES - used to include multiple entities,
- ID - values of this type are used as identifiers,
- IDREF - used for referring occurrences of identifiers,
- IDREFS - used for referring occurrences of multiple identifiers,
- $(val_1 \mid \ldots \mid val_k)$ - used for an enumeration type. This is a list of allowed values of the attribute.
- NMTOKEN - character data with some additional restrictions,
- NMTOKENS - a list of multiple name tokens,
- NOTATION - it is explained below.

A qualifier $\texttt{qualifier}_{\texttt{i}}$ 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 attribute should be optional,
- #REQUIRED - used if the attribute should be mandatory.

• Entity Declarations, which are of the form:

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

Entities are variables used to define shortcuts to common text (e.g. if an entity is referred in a DTD, during a processing of the DTD the reference is replaced by the declared text). 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:

```
<!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, for example,

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

Example 10. *The following DTD defines a structure of an XML document for a book store:*

```
<!ELEMENT bib  (book* )>
<!ELEMENT book (title,  (authors | editor), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT authors  (author*)>
<!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 )>
```

The main element of the document conforming to this schema has the name bib *and contains zero or more* book *elements, each of them having elements named:* title, editor *or* authors, publisher *and* price. *Additionally, each* book *element has a mandatory attribute* year. *Each element* authors *contains a list of* author *elements which include 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 defining multiple elements with the same name. Thus, for example, it is 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.

2.2.2 XML Schema

XML Schema [34, 44, 54] 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 defining multiple elements with the same name and different content. An important advantage of XML Schema is that schemata are specified in XML so no special syntax is needed. The description of XML Schema presented in this section is based on [44].

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 a 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 simple types,
- by a union: union of simple types,
- by a restriction, for instance a restriction on a list length (*minLength, maxLength*), bounds on numbers (*minInclusive, 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) such as: string, integer, float, date, etc.

Example 11. *This is an example of a declaration of a simple type* april_date*, which is a restriction of a simple type* date*. (The example comes from [44].)*

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

The elements of the type april_date *are those elements of type* 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:
  - `<attribute name="..." type="..." use="..."/>`
    where `type` specifies the attribute type and `use` is either *optional, required*, or *prohibited*,
  - `<anyAttribute ... />`
    allows the insertion of any attribute,

- a content model declaration:
  - 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 declarations or references of the form

    -
      ```
      <element name="..." type="..." minOccurs="..."
                                     maxOccurs="..."/>
      ```

    -
      ```
      <element ref="..." minOccurs="..."
                         maxOccurs="..."/>
      ```
      where *name* and *type* specify respectively the element's name and type, *ref* is a reference to an element definition, and *minOccurs* and *maxOccurs* are constrains on the number of occurrences,
    - `<any .../>` - a declaration allowing the insertion of any element,
  - mixed content model: implemented through a *mixed* attribute in *complexType* declaration. 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 the constraint called *Unique Particle Attribution* [1]. This restriction is similar to the restriction put on content models in DTD and it is equivalent to the requirement that the content models are 1-unambiguous in the sense of [14]. Another restriction related to a complex content model is called *Element Declarations Consistent* [1]. It says that the content model cannot contain two declarations or references to elements of the same name and of a different type.

XML Schema allows to define element and attribute groups which are named content models that can be reused in multiple locations as fragments of content models.

Example 12. *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. (The example comes from [44].)*

```
<complexType name="OrderType" mixed="true">
  <choice>
    <element ref="address"/>
    <sequence>
      <element ref="phone"/>
      <element ref="email" minOccurs="0"/>
    </sequence>
  </choice>
  <attribute name="id" type="unsignedInt" use="required"/>
</complexType>
```

XML Schema provides a mechanism of derived types 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:

  ```
  <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. In the following example, the type *RestrictedPurchaseOrderType* is derived by restriction from the type *PurchaseOrderType*. 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:

```
<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 [22, 55] has been defined by the *Oasis* consortium. It is 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 consists of production rules which are similar to production rules from regular tree grammars. 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 [2]), 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 named *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 sequence 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*.

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 }
```

2.3 XQuery and its Type System

XQuery [25], a dominant XML query language developed by W3C, became W3C recommendation in January 2007. This chapter presents a brief introduction to XQuery. For a more detailed description a reader is referred e.g. to [25, 40] (the book [40] is from 2003 and it refers to a little outdated version of XQuery). Many examples in this chapter originate from the book [40].

XQuery is a functional language where queries are expressions to be evaluated. Expressions can be flexibly combined in order to create new expressions. XQuery extends the language XPath, used for addressing parts of XML documents. XPath provides a means for selecting information within existing XML document but it does not provide a way to construct new XML elements. The newest versions of the languages, XPath 2.0 and XQuery 1.0, are closely related, both of them make use of the same data model (the abstract, parsed, logical structure of an XML document).

2.3.1 Data Model

The input and output of XQuery are defined formally in terms of a data model [3] which provides an abstract representation of an XML document. XML documents are represented as ordered trees with a document node as the root node. There are six other kinds of nodes in the trees: element, attribute, text, comment, processing instruction, and namespace nodes. Nodes have identity and a linear ordering called document order in which each node appears before its children.

Every value of an XQuery expression is an ordered sequence of nodes or atomic values. Atomic values are instances of the built-in data types defined by XML Schema, such as strings, integers, decimals, and dates.

Element and attribute nodes have a name, a type, a string value and a typed value. For XML documents associated with a schema the types of nodes are determined by schema validation process. For not validated data, if no more specific type is known for a node, a type annotation *xs:untyped* is assigned to an element node, and a type annotation *xs:untypedAtomic* is assigned to an attribute node. (A type annotation *xs:untypedAtomic* is also assigned to untyped atomic values). String value of a node is the concatenation of all text from the text node descendants in document order. A typed value of a node is a sequence of zero or more atomic values. It is derived from the node's string value and its type annotation.

2.3.2 Language Constructs

The section briefly describes the most important constructs of XQuery.

Path Expressions

Path expressions are used in XQuery to locate nodes in XML data. A path expression consists of a series of steps, separated by the slash, /. The result of each step is a sequence of nodes. For example, the following simple path expression returns book elements that are children of bib elements in a *books.xml* document:

```
document("books.xml")/bib/book
```

Each step is evaluated in the context of a particular node, called the context node. The context node of the path expression above is a document node given by `document("books.xml")`.

Every step consists of three parts. The first part, called the axis, specifies the direction of travel in the hierarchy. Some of the axes commonly used in XQuery are (x stands for a path expression and y stands for an element name or an attribute name): child axis e.g. `x/child::y` abbreviated as `x/y`, descendant-or-self axis e.g. `x/descendant-or-self::node()/y` abbreviated as `x//y`, attribute axis e.g. `x/attribute::y` abbreviated as `x/@y`, parent axis e.g. `x/parent::node()` abbreviated as `x/..`, self axis e.g. `x/self::node()` abbreviated as `x/.`. The second part of the step is a node test. It can be used to select nodes with a given name or nodes of a given kind such as comment, attribute, processing instruction, etc. The third part of a step is a predicate (or a list of predicates) which is a boolean condition that selects a subset of nodes computed by a step expression. Predicates are enclosed in square brackets. For example the query

```
document("books.xml")//book/author[last="Smith"]
```

returns *author* elements that are children of a *book* element and that have a child element *last* whose value is ”Smith”. The query

```
document("books.xml")//book/author[1]
```

returns *author* elements that are the first child of any *book* elements occurring in the document *books.xml.*

Element Constructors

A limitation of path expressions is that they only can select nodes. A full query language needs a facility to construct new elements and attributes. This facility in XQuery is called an element constructor and it uses XML syntax. For example, here is an element constructor that creates an author element:

```
<author>
  <last>Johnson</last>
  <first>Crockett</first>
</author>
```

In the example above, the content of elements is constant. However it can be generated dynamically by using XQuery expressions. Such expressions are enclosed in curly braces to indicate that they are to be evaluated rather than treated as literal text. The expressions are evaluated and replaced by their values. For example, the following query returns an element books whose content is a sequence of titles of books from a document "books.xml".

```
<books>{ document("books.xml")//book/title }</books>
```

FLWOR Expressions

The name FLWOR is an abbreviation formed by the initial letters of the clauses that may occur in a FLWOR expression:

- `for` clause: introduces variables and for each variable it provides a sequence of values over which the variable is to iterate. Thus it generates a sequence of tuples of variable bindings.
- `let` clause: introduces variables and binds them to the entire result of an expression. Thus it adds the bindings to the tuples generated by `for` clause.
- `where` clause: filters tuples by discarding the ones which do not satisfy a condition.
- `order` clause: sorts the tuples.
- `return` clause: is evaluated once for each retained tuple and builds the corresponding result.

This is an example of an FLWOR expression:

```
for $b in document("books.xml")//book
let $e := $b/author[1]/last
where $b/@year = "2005"
order by $e
return $b
```

The expression introduces a variable $\$b$ which iterates over book elements. In each iteration the variable $\$e$ obtains a new value which is a last name of the first author for the currently chosen book. The FLWOR expression returns a list of book elements with the value of the attribute *year* equal "2005". The returned book elements are sorted by the last name of the first author of a book.

Functions

XQuery provides a library of predefined functions such as *min*(), *max*(), *count*(), *sum*(), *avg*(), etc. Also users can define functions. A function definition must specify the name of the function and the names of its parameters. Optionally types of the parameters and the result type of the function can also be specified. A function definition may be recursive. This is an example of a function definition:

```
define function books-by-year($year as xs:integer)
  as element(book)*{
    for $b in document("books.xml")//book
      let $e := $b/author[1]/last
      where $b/@year = $year
      order by $e
      return $b
}
```

The result type of the function is declared as `element(book)*` (the type notation is explained in the next section). The function returns book elements of the given year sorted by the last name of the first author.

2.3.3 Type System

XQuery is a strongly-typed language. Its type system is based on XML Schema [2] and it supports XML Schema atomic types (such as string, integer, dateTime, etc.), complex types from imported schemas, and XML document structure types (such as element, attribute, node, comment, etc.). XQuery types are integral part of the language: they belong to its semantics and they may influence query results.

A type is defined as a set of constraints that defines a set of values in the XQuery data model. A value matches a type if it satisfies the constraints. XQuery uses types to check correctness of queries and to ensure that operations are being applied to data in appropriate ways (such as the arithmetic operators which require numeric data). An important relation defined on types is subtyping. A type T is a subtype of another type T' if all values matching the type T match also the type T'. Based on this relation the equivalence of types can be defined. Two types T and T' are equivalent only if T is a subtype of T' and T' is a subtype of T.

XQuery type system is intended to help in finding errors in programs. Type errors can be detected during a static analysis phase which is performed on a query only and which is independent on input data. In static analysis XQuery type system can be used to find static type errors in queries such as "abc" + 3. A result of static typing of an XQuery program is an abstract syntax tree which assigns to each subexpression its static type. Assigning an empty type to an expression results in a type error. Besides static types which are assigned to expressions during static analysis there are dynamic types which are assigned to values during query execution. Dynamic types can also be used to detect type errors in the case when static type analysis is not performed. For instance `sum(doc("products.xml")//price)` raises an error if `price` does not contain a valid numeric data (required by the function *sum*()). The type system has a property which guarantees that any value returned by an expression conforms to the static type inferred for the expression. As a consequence, a query which raises no type errors during static analysis will also raise no type errors during execution on valid input data.

Type Notation

XQuery has a formal type notation for defining types which is simpler and more concise than XML Schema. Imported XML Schema type definitions are translated into this notation. For example, the following XML Schema type definition (the example comes from [40])

```
<xs:element name="rating" type="xs:string">
<xs:element name="user" type="User">
<xs:complexType name="User">
  <xs:sequence>
    <xs:element name="name">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="first" type="xs:string" minOccurs="0">
          <xs:element name="last" type="xs:string">
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element ref="rating" minOccurs="0">
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
```

is translated into

```
define element rating of type xs:string
define element user of type User

define type User {
  attribute id of type xs:ID,
  element name of type AnonymousType1,
  element rating ?
}

define type AnonymousType1 {
  element first of type xs:string ?,
  element last of type xs:string
}
```

During the translation a unique name for each anonymous type is invented, thus every element has a named type.

The translation simplifies definitions by ignoring value constraints. For example,

```
<xs:simpleType name="myPositiveInteger">
  <xs:restriction base="xs:positiveInteger">
    <xs:maxExclusive value="100">
  </xs:restriction>
</xs:simpleType>
```

is translated into

```
define type myPositiveInteger {xs:positiveInteger}
```

Interleaving operator "&" is used to represent *All* groups of XML Schema. As XQuery needs closure for inferred types (any inferred type is a valid type), the XML Schema restrictions such as *Element Declarations Consistent* [1] and *Unique Particle Attribution* [1] are not present in XQuery. Thus XQuery type formalism allows to define types that cannot be defined by XML Schema.

The formal type notation is used in the formal XQuery semantics but it is not available in XQuery syntax. The notation for referring to types in XQuery queries, called *sequence types*, is a subset of the formal type notation. The term sequence type is used to refer to a type of an XQuery expression as such an expression evaluates always to a sequence (of nodes or atomic values).

Some of the built-in XQuery types which can be used as sequence types are:

- *element*() matches any element node,
- *attribute*() matches any attribute node,
- *node*() matches any node,
- *text*() matches any text node,
- *empty*() matches an empty sequence,
- *xs:string*, *xs:decimal*, *xs:anyType*, etc. match instances of the specific XML Schema built-in types

The sequence type notation may refer to types imported from a schema. A sequence type which refer to an element defined by a schema has two parameters: the name of the element from the imported schema and the name of the type. In order for an element to match a sequence type, both the name of the element and its type must match. For example, a sequence type *element*(*creator*, *person*) matches elements (one element sequences) with a name *creator* and a type annotation *person* (or any other type derived by restriction or extension from *person*). The second parameter can be omitted, and if it is, the sequence type matches any element with the given name. Similar notation is used to refer to attributes defined by a schema. For instance, a declaration *attribute*(@*price*, *currency*) matches attributes named *price* of type *currency*.

In the notation for sequence types, occurrence indicators may be used to indicate the number of items in a sequence: the character "?" indicates zero or one items, " $*$ " indicates zero or more items and " $+$ " indicates one or more items. Additionally, type operators can be used for concatenation (",") and union ("|"). For example, a sequence type $(element(users)\,|\,element(articles))+$ matches any non empty sequence of elements named *users* or *articles*.

Type Conversions

To simplify typing rules of the XQuery type system, types of expressions are approximated in a process called **factorization** [40]. In this process an initial type is transformed to a type which is the choice of the item types (representing one element sequences) that occur in the initial type followed by an occurrence indicator. For example, a type $(element\,a?, element\,b?)$ is approximated as $(element\,a\ |\,element\,b)*$.

Some operations of XQuery, such as function calls or processing of operators that require numeric operands, require **type promotion**. Type promotion is a process in which atomic values are promoted from one type to another, for example during function calls or during processing of operators that require numeric operands. For example, a type *xs*:*decimal* can be promoted to *xs*:*double*.

Type checking

Typechecking is performed in queries wherever types are declared e.g. in function declarations, variable declarations. Function declarations may specify types of their arguments and results. A function which require an argument of a type T will accept an argument of different type T' if T' can be promoted to T or if T can be derived (by restriction or extension) from T'. In variable declarations, types of variables can be declared explicitly. A type error is raised if a variable obtains a value of a type which cannot be promoted to the required type or from which the required type cannot be derived.

The elements constructed by XQuery expressions can be validated against schema types using *validate* expression. The expression returns a new element node (or a document node) with no parent. The new node and its descendants are given type annotations resulting from the validation process of the operand node.

Some other type based operations in XQuery are:

- *instance of* - an operator which tests whether a value matches a given type and returns a boolean value. For example,

```
3.14 instance of xs:decimal
```

returns *true*.

- *typeswitch* - it chooses an expression to evaluate based on the dynamic type of an input value. Its usage can be illustrated by the example:

```
typeswitch($customer/billing-address)
  case $a as element(*, USAddress) return $a/state
  case $a as element(*, CanadaAddress) return $a/province
  case $a as element(*, JapanAddress) return $a/prefecture
  default return "unknown"
```

- *treat as* - the expression can be used to change the static type of the result of an expression without changing its value and its dynamic type. Its effect is twofold: (1) it assigns a specific static type to its operand (a restriction of its actual static type), which can be used for type-checking in a static analysis; and (2) at run-time, if the actual value of the expression does not conform to the named type, it returns an error value. For example, in the expression

```
$myaddress treat as element(*, USAddress)
```

the static type of $\$myaddress$ may be $element(*, Address)$, a less specific type than *element*(∗, *USAddress)*. However, at run-time, the value of $\$myaddress$ must match the type *element*(∗, *USAddress)*; otherwise an error is raised.

2.4 DIG Interface

Ontologies provide information about concepts, roles and individuals in a given application domain. Thus an ontology gives a common vocabulary to be understood in the same way by various applications in the domain. A main language used to define ontologies is OWL, developed by W3C. OWL is based on description logics.

An OWL file representing an ontology is just an encoding of a set of axioms. To make use of the axioms one needs an ontology reasoner. An ontology reasoner makes it is possible to draw conclusions from the set of axioms such as discovering implicit subclass relationships and discovering class equivalence. To communicate with the reasoner we need to use a reasoner interface. For this purpose we have chosen DIG interface [8] which is supported by many reasoners. The DIG interface is an API for a general description logic system. It is capable of expressing class and property expressions common to most description logics. Using DIG, clients can communicate with a reasoner through the use of HTTP POST requests. A *request* is an XML encoded message of one of the following types: management, ask or tell. Management requests are used e.g. to identify the reasoner along with its capabilities or to allocate a new knowledge base and return its unique identifier. *Tell requests*, containing *tell statements*, are used to make assertions into the reasoner's knowledge base. *Ask requests*, containing *ask statements*, are used to query the knowledge base. *Responses* to ask requests contain *response statements*. Tell, ask and response statements are built out of *concept statements* which are used to denote classes, properties, individuals etc. Here we present an extract of DIG statements used in our examples ($C, C_1, C_2, \ldots$ are concept statements):

- Concept statements:
  - `<catom name="`*CN*`"/>` – a concept (class) *CN*,
  - `<ratom val="`*RN*`"/>` – a role (property) *RN*,
  - `<some>` R C `</some>` – the concept whose objects are in relation R with some objects of a concept C (it corresponds to $\exists R.C$ in description logics).
- Tell statements:
  - `<defconcept name="`*CN*`"/>` – introduces a concept *CN*,
  - `<defrole name="` *RN*`"/>` – introduces a role *RN*,
  - `<impliesc>`C_1 C_2`</impliesc>` – introduces an axiom stating that a concept C_1 is subsumed by a concept C_2.
- Ask statements:
  - `<subsumes>`C_1 C_2`</subsumes>` – a Boolean query, it asks whether a concept C_2 is subsumed by a concept C_1,
  - `<descendants>`C`</descendants>` – it asks for the list of subclasses of a concept C.
- Response statements:
  - `<true/>` – if a statement is a logical consequence of the axioms in the knowledge base,
  - `<false/>` – if a statement is not a logical consequence of the axioms in the knowledge base,
  - `<error/>` – if, for instance, a concept queried about is not defined in the knowledge base,
  - `<conceptSet>` `<synonyms>` $C_{11} \ldots C_{1n_1}$ `</synonyms>`

    $\ldots$

    `<synonyms>` $C_{m1} \ldots C_{mn_m}$ `</synonyms>`

    `</conceptSet>`

    The response statement contains a list of concepts grouped by *synonyms* i.e. equivalent concepts.

DIG requests and responses are XML documents, some of their elements contain attributes. For instance, the attribute *id* is used to associate the obtained answers with the submitted queries.

Example 13. *This is an example of a query request to be sent to an ontology reasoner. It contains three DIG ask statements. The first two ask whether concepts* sugar *and* potato *are subclasses of the concept* gluten-containing. *The third one asks for direct subclasses of the class* gluten-containing. *(We skip namespace declarations in the elements* asks *and* responses.*)*

```
<asks uri="uri_of_knowledge-base" ... >
   <subsumes id="q1">
      <catom name="gluten-containing"/>
      <catom name="sugar"/>
   </subsumes>
   <subsumes id="q2">
      <catom name="gluten-containing"/>
      <catom name="potato"/>
   </subsumes>
   <children id="q3">
      <catom name="gluten-containing"/>
   </children>
</asks>
```

This is a possible response to the query:

```
<responses ... >
   <false id="q1"/>
   <error id="q2" message="Undefined concept name potato
                           in TBox DEFAULT"/>
   <conceptSet id="q3">
      <synonyms> <catom name="flour"/> </synonyms>
      <synonyms> <catom name="spaghetti"/> </synonyms>
   </conceptSet>
</responses>
```

Chapter 3

Type Specification

This chapter introduces a formalism called *Type Definitions* which we use for defining classes of data terms. The formalism is a generalisation of the formalism presented in [15]. 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 [24] (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 [13, 45] 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. Similar formalisms are employed for XML processing languages such as XDuce [37], CDuce [9] or XCentric [23]. A novelty of Type Definitions is that they 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 [5, 42] 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 results in more efficient algorithms for type checking. The restricted class of Type Definitions is called *proper* and it corresponds [27] to a single type tree grammar in the sense of [45].

3.1 Type Definitions

This section introduces a formalism for specifying decidable sets of data terms representing XML documents. First we specify a set of **type names** $\mathcal{T} = \{\mathit{Top}\} \cup \mathcal{C} \cup \mathcal{S} \cup \mathcal{V}$ which consist of a type name *Top* and

- **type constants** from the alphabet $\mathcal{C}$,
- **enumeration type names** from the alphabet $\mathcal{S}$,
- **type variables** from the alphabet $\mathcal{V}$.

A Type Definition associates type names with sets of data terms. The set $[\![T]\!]$ associated with a type name T is called the **type** denoted by T (or simply type T). The type $[\![\mathit{Top}]\!]$ is the set of all data terms. For T being a type constant or an enumeration type name, the elements of $[\![T]\!]$ are basic constants.

Type constants correspond to basic types of XML schema languages such as *String* or *Integer*. The set of type constants is fixed and finite; for each type constant $T \in \mathcal{C}$ the set of basic constants $[\![T]\!]$ is fixed. In our examples we will use a type constant *Text* assuming that $[\![\mathit{Text}]\!]$ 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 [15] 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 ϵ. A *regular expression* over an alphabet Σ is ε, ϕ, any $a \in \Sigma$ 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 Σ is assigned to each regular expression r in a standard way: $L(\phi) = \emptyset$, $L(\varepsilon) = \{\epsilon\}$, $L(a) = \{a\}$, $L(r_1 r_2) = L(r_1)L(r_2)$, $L(r_1|r_2) = L(r_1) \cup L(r_2)$, and $L(r_1^*) = L(r_1)^*$.

Definition 16. *A* **regular type expression** *is a regular expression over the alphabet of type names $\mathcal{T}$. We abbreviate a regular expression $r^n|$*

$r^{n+1}|\cdots|r^m$, where $n \leq m$, as $r^{(n:m)}$, $r^n r^*$ as $r^{(n:\infty)}$, rr^* as r^+, and $r^{(0:1)}$ as $r^?$. A regular type expression of the form

$$T_1^{(l_1\,:\,u_1)} \cdots T_k^{(l_k\,:\,u_k)}$$

where $k \geq 0$, $0 \leq l_i \leq u_i \leq \infty$ 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. Formally, a multiplicity list r represents the set $perm(L(r))$ of all permutations of the language $L(r)$ [1].

Definition 17. *A* **Type Definition** *is a set D of rules of the form*

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

where T is a type variable, T' is an enumeration type name, l is a label, r is a regular type expression, s is a multiplicity list, $n \geq 0$, and $c_1, \ldots, c_n$ are basic constants. A rule $U \to G \in D$ will be called a **rule for** *U in D. We require that for any type name $U \in \mathcal{V} \cup \mathcal{S}$ 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 $label_D(T) = l$. For T being a type constant or an enumeration type name we define $label_D(T) = \$$. The regular expression in a rule for a type variable T is called the* **content model** *of T. If a rule for a type variable T in D is $T \to l[r]$ (or $T \to l\{r\}$) then $[\,]$ (or $\{\,\}$, respectively) are called the* **parentheses** *of T.*

We assume that alphabet of labels $\mathcal{L} \cup \{\$\}$ is totally ordered by a relation $\leq$; we call this ordering *alphabetic ordering*. A multiplicity list $W_1 \ldots W_k$, where each $W_i = T_i^{(n_{i,1}:n_{i,2})}$ is **sorted** wrt. D if

- none of $T_1, \ldots, T_k$ is *Top* and $label_D(T_1) \leq \ldots \leq label_D(T_k)$, or
- $T_k = Top$ and $label_D(T_1) \leq \ldots \leq label_D(T_{k-1})$.

Notice that for any multiplicity list, a sorted multiplicity list representing the same multiset of type names can be obtained.

The formalism of Type Definitions is a slight generalisation of the formalism of [15] as it deals with a type *Top* and enumeration type names. Another 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 14. *Consider a Type Definition D:*

[1]For a set of sequences Z, $perm(Z)$ will denote the set of all permutations of the sequences from Z.

$$\begin{array}{l} \mathit{Cd} \rightarrow \mathit{cd}[\mathit{Title}\ \mathit{Artist}^{+}\ \mathit{Category}^{?}] \\ \mathit{Title} \rightarrow \mathit{title}[\mathrm{Text}\ \mathit{Subtitle}^{?}] \\ \mathit{Subtitle} \rightarrow \mathit{subtitle}[\mathrm{Text}] \\ \mathit{Artist} \rightarrow \mathit{artist}[\mathrm{Text}] \\ \mathit{Category} \rightarrow \text{"}\mathit{pop}\text{"} \mid \text{"}\mathit{rock}\text{"} \mid \text{"}\mathit{classic}\text{"} \end{array}$$

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:* cd, title, subtitle, artist*, and* pop, rock, 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 $[\![T]\!]$, or from the rule for T. This can be concisely formalized as follows (treating Type Definitions similarly to tree automata).

Definition 18. *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 ν from the subterms of t to type names such that $\nu(t) = T$ and for each subterm u of t*

- *if $T = Top$ then $\nu(u) = Top$,*
- *otherwise, if u is a basic constant then $\nu(u) \in \mathcal{C}$ and $u \in [\![\nu(u)]\!]$ or $\nu(u) \in \mathcal{S}$ and there exists a rule $\nu(u) \rightarrow \cdots |u| \cdots$ in D,*
- *otherwise $\nu(u) = U \in \mathcal{V}$ and*
  - *there is a rule $U \leftarrow l[r] \in D$, $u = l[t_1, \ldots, t_n]$, and $\nu(t_1) \cdots \nu(t_n) \in L(r)$,*
  - *or there is a rule $U \leftarrow l\{r\} \in D$, $u = l\{t_1, \ldots, t_n\}$, and $\nu(t_1) \cdots \nu(t_n)$ is a permutation of a string in $L(r)$.*

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

Example 15. *Given the Type Definition D from the previous example, a data term*

$$t = \mathit{cd}[\,\mathit{title}[\text{"}\mathit{Stop}\text{"}],\ \mathit{artist}[\text{"}\mathit{Sam\ Brown}\text{"}],\ \text{"}\mathit{pop}\text{"}]$$

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 $[\![\ ]\!]_D$ 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 $T_1 \cdots T_n \in L(r)\,\}$. Notice that if $D \subseteq D'$ are Type Definitions then $[\![T]\!]_D = [\![T]\!]_{D'}$ for any type name T occurring in D. We use $\mathbf{types}(r)$ to denote the set of all type names occurring in the regular expression r. We define a set o type names with a given label l occurring in a regular expression r as $\mathbf{types}_D(l, r) = \{T \mid T \in \mathit{types}(r)$ and $\mathit{label}(T) = l\}$. Observe that if $d \in [\![T]\!]$ then either $T = \mathit{Top}$ or $\mathit{root}(d) = \mathit{label}(T)$.

In the next section we will need an algorithm for computation of a multiplicity list r representing the union of multisets represented by multiplicity lists $r_1, \ldots, r_n$. In general the union of such multisets cannot be represented by a multiplicity list. Thus we show a way how a multiplicity list representing a superset of such union can be constructed. So we show how to construct r such that $\mathit{perm}(L(r_1)) \cup \ldots \cup \mathit{perm}(L(r_n)) \subseteq \mathit{perm}(L(r))$. Let $Q = \{U_1, \ldots, U_m\}$ be the set of type names occurring in multiplicity lists $r_1, \ldots, r_n$. Let, for $i = 1, \ldots, n$, $r'_i = U_1^{(l_1^i:u_1^i)} \cdots U_m^{(l_m^i:u_m^i)}$, where for $j = 1, \ldots, m$, either $U_j^{(l_j^i:u_j^i)}$ is a subexpression of r_i, or U_j does not appear in r_i and $l_j^i = u_j^i = 0$. The multiplicity list r is $U_1^{(l_1:u_1)} \cdots U_m^{(l_m:u_m)}$, where $l_j = \mathit{min}_{i=1,\ldots,n}(l_j^i)$ and $u_j = \mathit{max}_{i=1,\ldots,n}(u_j^i)$ for $j = 1, \ldots, m$.

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 an efficient algorithm for the inclusion check we impose a restriction on Type Definitions which is discussed in this section.

Consider a Type Definition D and a content model r occurring in D. We call the content model r **proper**, if the following property holds. The only type name occurring in r is *Top*, or r does not contain *Top* and 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 content models occurring in D are proper. Thus given a term $l[c_1 \ldots c_n]$ and a rule $T \to l[r] \in D$ or a term $l\{c_1 \ldots c_n\}$ and a rule $T \to l\{r\} \in D$, for each c_i the root of c_i (and the parentheses of c_i, if c_i is not a basic constant) determines at most one type name S such that S occurs in r and S is *Top* or $\mathit{label}_D(S) = \mathit{root}(c_i) = l_i$ (and the parentheses of c_i are the same as the parentheses of S, if c_i is not a basic constant). 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 \$.

The class of proper Type Definitions, when restricted to ordered terms (i.e. without $\{\}$), is essentially the same as single-type tree grammars of [45]. 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.

Example 16. *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, the same parentheses, 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.*

Now we explain how a non proper content model r from a Type Definition D can be approximated by a proper one r'. The content model r is treated in D either as a regular expression or as a multiplicity list. The algorithm we present, creates new types, thus it creates a new Type Definition D' that extends the Type Definition D with rules defining the new types. The new content model has a property $[\![r]\!]_D \subseteq [\![r']\!]_{D'}$ if r is treated as a regular expression and $perm([\![r]\!]_D) \subseteq perm([\![r']\!]_{D'})$ if r is treated as a multiplicity list.

If a type name Top occurs in r then we replace each type name in r by Top obtaining r''. If r is treated as a multiplicity list then r'' is of the form $Top^{(l_1,u_1)} \cdots Top^{(l_k,u_k)}$ and $r' = Top^{(l_1+\ldots+l_k, u_1+\ldots+u_k)}$. Otherwise $r' = r''$.

Now we assume that Top does not occur in r. Let $S_C, S_1, \ldots, S_n$ be disjoint sets of type names such that $S_C \cup S_1 \ldots \cup S_n = types(r)$ 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 the unions (or supersets approximating the unions) of the types from each set:

- If S_C contains only enumeration type names then T_C is an enumeration type name and the rule defining $[\![T_c]\!]$ is obvious. 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 $T_{ij} \rightarrow l_i[r_{ij}]$ the corresponding type T_i is defined as $T_i \rightarrow l_i[r_{i1} \,|\, \cdots \,|\, r_{ik_i}]$, for $j = 1, \ldots, k_i$, where $k_i = |S_i|$.

- For a set S_i of type names defined by rules $T_{ij} \rightarrow l_i\{r_{ij}\}$ for $j = 1, \ldots, k_i$ and $k_i = |S_i|$, the corresponding type T_i is defined as $T_i \rightarrow l_i\{u_i\}$, where u_i is a multiplicity list representing the union (or an approximation of the union) of the languages represented by multiplicity lists r_{ij} for $j = 1, \ldots, k_i$.

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 $S_s \in \{S_C, S_1, \ldots, S_n\}$).

Additionally, if r is treated as a multiplicity list, in order for r' to be a multiplicity list, all its subexpressions of the form $T_S^{(l_1,u_1)}, \ldots, T_S^{(l_k,u_k)}$ must be replaced by an expression $T_S^{(l_1+\ldots+l_k, u_1+\ldots+u_k)}$.

The algorithm can be used for creation of a proper Type Definition defining types that are approximations of the types defined by a non proper Type Definition. In order to obtain such a proper Type Definition the algorithm must be applied to all non proper content models of D and to the new content models created by the algorithm. Termination of this process is due to the fact that the new content models are created only for the types from D and for types representing unions of the types from D. As the number of the types defined by D is finite, the number of possible unions of the types is also finite.

Example 17. *Consider a Type Definition $D = \{A \rightarrow a[\,A \,|\, B \,|\, C\,], B \rightarrow a[\,C\,], C \rightarrow c[\,Text\,]\}$, which is not proper. We want to construct a proper Type Definition D' defining types that are approximations of the types defined by D. As the content model $A \,|\, B \,|\, C$ of A is not proper we approximate it by a proper one $A\dot{\cup}B \,|\, A\dot{\cup}B \,|\, C$, where the type $A\dot{\cup}B$ represents the union of types A and B and it is defined by a rule $A\dot{\cup}B \rightarrow a[\,A \,|\, B \,|\, C\,]$. The content model of the type $A\dot{\cup}B$ which is not proper must be replaced by a proper content model being its approximation. This approximation, which is already computed, is $A\dot{\cup}B \,|\, A\dot{\cup}B \,|\, C$. Thus the type Definition D' is $\{A \rightarrow a[\,A\dot{\cup}B \,|\, A\dot{\cup}B \,|\, C\,], B \rightarrow a[\,C\,], C \rightarrow c[\,Text\,], A\dot{\cup}B \rightarrow a[\,A\dot{\cup}B \,|\, A\dot{\cup}B \,|\, C\,]\}$.*

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 [38]. Notice that the XML definition [33] requires (Section 3.2.1) that content models specified by regular expressions in element type declarations of a DTD are *deterministic* in the sense of Appendix E of [33]. A similar requirement in XML Schema is called *Unique Particle Attribution.* The formal meaning of this requirement is that the regular type expressions are 1-unambiguous in a sense of [14]. 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 **useless** in a regular expression r if no string in $L(r)$ contains T. (If r contains a useless symbol then the regular expression ϕ occurs in r.)

A type name T in a Type Definition D will be called **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 occurrences of a type name *Top*, 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 $T_i \to l[r_i]$ or of the form $T_i \to l\{r_i\}$ such that there exists a sequence of marked type names $S_1 \cdots S_m \in L(r_i)$ $(m \geq 0)$. 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 $S_1 \cdots S_m \in L(r)$ $(m \geq 0)$. Let λ be a parse tree of r (e.g. a parse tree for a regular expression $((T_1^*|T_2)T_1)|T_3$ represented as a term is $or(then(or(star(T_1), T_2), T_1), T_3)$). 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 ϕ (the node is a leaf),
- if a node is *star* we replace it by ϵ and remove its child (the node becomes a leaf),
- if a node is *or* we visit its children. If both of them were replaced by ϕ we replace the node by ϕ; otherwise we replace it by a child which was not replaced by ϕ,
- if a node is *then* we visit its children.

If the result tree does not have any ϕ 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 λ 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 18. *Let us use the algorithm to find nullable type names in a Type Definition* $D = \{\, A \to a[AB],\ B \to b[B^*] \,\}$. *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. To denote the intersection of types T and U we introduce a new type name $T\dot{\cap}U$. We do not require that the Type Definition D is proper. A simpler algorithm for type intersection of types defined by a proper Type Definition was presented in [58]. The algorithm we present in general may produce results which are approximations i.e. the set $[\![T\dot{\cap}U]\!]$ is a superset of the set $[\![T]\!] \cap [\![U]\!]$. Such approximation is necessary if there is a need to intersect types whose content model is not a proper multiplicity list or if it is a multiplicity list distinct from Top^* with a type name Top appearing in it. This is because the intersection of two languages represented by multiplicity lists not satisfying these conditions may be not expressible by a multiplicity list. Thus, we introduce a notion of an **intersectable** multiplicity list. A multiplicity list is intersectable if it is Top^* or if it is proper and Top does not appear in it. A multiplicity list with a type name Top appearing in it can be approximated by an intersectable multiplicity list Top^*. A multiplicity list without Top can be approximated by a proper multiplicity list (obtained using the algorithm from Section 3.1.1). Such a multiplicity list is intersectable.

Example 19. *Consider multiplicity lists $r_1 = A^{(1:2)}B^?$ and $r_2 = Top^{(2:2)}$. Intersection of the languages represented by r_1 and r_2, which is $perm(L(AA|AB))$, cannot be represented by a multiplicity list.*

The following algorithm produces exact results i.e. $[\![T\dot{\cap}U]\!] = [\![T]\!] \cap [\![U]\!]$ if the Type Definition D does not contain non intersectable multiplicity lists. Otherwise non intersectable multiplicity lists must be first approximated by intersectable ones. This produces a Type Definition D' to which the algorithm is applicable. As a result we obtain D'' such that $[\![T\dot{\cap}U]\!]_{D''} \supseteq [\![T]\!]_D \cap [\![U]\!]_D$.

Now, we are ready to present an algorithm for obtaining the intersection of types T, U defined by a Type Definition D, which contains only intersectable multiplicity lists. In order to simplify the presentation we introduce an algorithm which computes intersection for each pair of types from D. In what follows we do not distinguish type names $T\dot{\cap}U$ and $U\dot{\cap}T$, and T and $T\dot{\cap}Top$, for any type names T, U. We assume that, for any pair T, U of type constants from D there exists another type constant $T\dot{\cap}U$ such that $[\![T\dot{\cap}U]\!] = [\![T]\!] \cap [\![U]\!]$. For each pair T, U of type names defined by D distinct from Top and such that at least one of them is not a type constant we proceed as follows:

- If T, U are enumeration type names, or one of them is an enumeration type name and the other is a type constant then D is augmented by the rule $T\dot{\cap}U \rightarrow c_1|\ldots|c_n$, where $[\![T]\!] \cap [\![U]\!] = \{c_1, \ldots, c_n\}$ and $T\dot{\cap}U$ is an enumeration type name.

- If one of the type names T, U is a type variable and another is an enumeration type name or a type constant then their intersection is empty and D is augmented by the rule $T\dot{\cap}U \to \phi$, where $T\dot{\cap}U$ is an enumeration type name.
- If T and U are type variables and $label_D(T) \neq label_D(U)$ then their intersection is empty and D is augmented by the rule $T\dot{\cap}U \to \phi$, where $T\dot{\cap}U$ is an enumeration type name.
- If T and U are type variables and D contain rules of the form $T \to l[r_1]$ and $U \to l\{r_2\}$ or of the form $T \to l\{r_1\}$ and $U \to l[r_2]$ then the intersection of T, U is empty and D is augmented the rule $T\dot{\cap}U \to \phi$, where $T\dot{\cap}U$ is an enumeration type name.
- If D contains rules of the form $T \to l[r_1]$ and $U \to l[r_2]$ then D is augmented by the rule $T\dot{\cap}U \to l[r]$, where $T\dot{\cap}U$ is a type variable, $L(r) = L(r_1') \cap L(r_2')$ and r_1', r_2' are obtained in the following way. For each T_i occurring in r_1 let $S_i = \{U_{i1} \dots U_{ik_i}\}$ be the set of type names occurring in r_2 such that if $T_i \neq \mathit{Top}$
  - U_{ij} is Top, or
  - U_{ij} has the same label as T_i and if T_i is a type variable then U_{ij} has also the same kind of parentheses as T_i.

  If $T_i = \mathit{Top}$ then S_i is the set of all type names occurring in r_2. Then r_1' is r_1, where each symbol T_i is replaced by a regular expression $T_i\dot{\cap}U_{i1}|\dots|T_i\dot{\cap}U_{ik_i}$ (or ϕ, if $k_i = 0$). r_2' is obtained analogically to r_1'.
- If D contains rules of the form $T \to l\{r_1\}$ and $U \to l\{r_2\}$ (where r_1, r_2 are intersectable) we try to compute a multiplicity list r representing the set $M = perm([\![r_1]\!]) \cap perm([\![r_2]\!])$ of sequences of data terms, in the following way.
  - if $r_1 = \mathit{Top}^*$ then $r = r_2$,
  - otherwise, if $r_2 = \mathit{Top}^*$ then $r = r_1$,
  - otherwise, (there is no Top in r_1, r_2) let r_1' be r_1, where each T_i is replaced by a type name $T_i\dot{\cap}U_i$ and U_i is a type name occurring in r_2 such that U_i has the same label as T_i and if T_i is a type variable then U_i has also the same kind of parentheses as T_i. As r_2 is proper only one such U_i can exist. If there is no such a type name U_i in r_2 then T_i is replaced by ϕ. Let r_2' be obtained from r_2 in the same way as r_1' is obtained from r_1. If r_1' or r_2' contain an expression of the form $\phi^{l:u}$, where $l > 0$, then $M = \emptyset$ and a multiplicity list representing M does not exist. Otherwise, r is a regular expression obtained by the concatenation of regular expressions r_1' and r_2', by removing subexpressions of the form $\phi^{0:u}$ and by replacing each pair of subexpressions of the form

$S^{(l:u)}$ and $S^{(l':u')}$ (where S is a type name) by the expression $S^{(l'':u'')}$, where $l'' = max(l, l')$ and $u'' = min(u, u')$.

If a multiplicity list r representing M is found then D is augmented by the rule $T\dot{\cap}U \to l\{r\}$, where $T\dot{\cap}U$ is a type variable. Otherwise, ($M = \emptyset$) D is augmented by the rule $T\dot{\cap}U \to \phi$, where $T\dot{\cap}U$ is an enumeration type name.

If the Type Definition D is proper the new Type Definition defining the type $T\dot{\cap}U$ is also proper.

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 [14], 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 contains only 1-unambiguous regular expressions then the complexity of the type intersection algorithm is polynomial. Otherwise, it is exponential.

Example 20. *Consider a Type Definition* $D = \{\, A \to l[B|C], B \to l[A^+], C \to m[\,],\ A' \to l[A'^*|C'],\ C' \to m[C'^*]\,\}$. *We construct a Type Definition* D' *which defines a type* $A\dot{\cap}A'$ *being the intersection of types* A *and* A' $([\![A\dot{\cap}A']\!]_{D'} = [\![A]\!]_D \cap [\![A']\!]_D)$. $D' = \{\, A\dot{\cap}A' \to l[B\dot{\cap}A'|C\dot{\cap}C'],\ B\dot{\cap}A' \to l[(A\dot{\cap}A')^+],\ C\dot{\cap}C' \to m[\,]\,\}$. *Example 22 will show that* $[\![A]\!]_D \subseteq [\![A']\!]_D$ *and that is why* $[\![A\dot{\cap}A']\!]_{D'} = [\![A]\!]_D$.

Example 21. *The example shows how to obtain the intersection of types* T_1, T_2*, where the content model of* T_1 *is a non intersectable multiplicity list. Consider a Type Definition* $D = \{T_1 \to l\{A_1^? A_2^?\}, T_2 \to l\{A^+\}, A \to a[C^*], A_1 \to a[\,], A_2 \to a[CC], C \to c[\,]\}$. *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* $T_1\dot{\cap}T_2 \to l\{A_1|A_2|(A_1A_2)\}$. *This is however not allowed as the content model of* $T_1\dot{\cap}T_2$ *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* $A_1^? A_2^?$ *by an intersectable 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* $A' \to a[(CC)^?]$. *Thus the multiplicity list* $A_1^? A_2^?$ *can be approximated as* $A'^? A'^?$ *which is equivalent to an intersectable multiplicity list* $A'^{(0:2)}$. *The type* T_1 *from* D *can be approximated as a type* T_1' *with an intersectable multiplicity list* $T_1' \to \{A'^{(0:2)}\}$. *The intersection of types obtained using the presented algorithm is* $T_1'\dot{\cap}T_2 \to l\{A'^{(1:2)}\}$. *The type* $[\![T_1'\dot{\cap}T_2]\!]$ *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 [15].

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 at least one of type names T_1, T_2 is *Top* then $(T_1, T_2) \in C(T_1, T_2)$,
- if T_1, T_2 are type constants or enumeration type names then $(T_1, T_2) \in C(T_1, T_2)$,
- if T_1, T_2 are type variables with the same kind of parentheses and $label(T_1) = label(T_2)$ 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 $T_1' \to l[r_1]$ and $T_2' \to l[r_2]$, or $T_1' \to l\{r_1\}$ and $T_2' \to l\{r_2\}$ (with the same label l), and
  - type names T_1'', T_2'' occur respectively in r_1, r_2,and
    - T_2 is *Top*, or
    - $label_{D_1}(T_1'') = label_{D_2}(T_2'')$ and, if T_1, T_2 are type variables they have the same parenthesis,

  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 $(T_1', T_2') \in C(T_1, T_2)$. We assume that the multiplicity lists occurring in D_1, D_2 are sorted.

```
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 at least one of $T_1', T_2'$ is a type name $Top$
        IF $T_2'$ is a type name $Top$ return true
        ELSE return false
    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 $T_1' \to l[r_1]$ and $T_2' \to l[r_2]$, or $T_1' \to l\{r_1\}$ and $T_2' \to l\{r_2\}$
        be rules of $D_1, D_2$, respectively
```

```
IF a type name $Top$ occurs in $r_2$
     (as $r_2$ is proper no other type name except $Top$ occurs in $r_2$)
     THEN let $r'_1$ be $r_1$ where every type name is replaced by $Top$
     Check whether $L(r'_1) \subseteq L(r_2)$
ELSE
     Let $s_2$ be the regular expression $r_2$ where every type name is
     replaced by its label
     Let $s_1$ be the regular expression $r_1$ where every type name,
     except $Top$, is replaced by its label and the type name $Top$
     is replaced by an arbitrary symbol not appearing in $s_2$
     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) and no useless symbols then $[\![T_1]\!]_{D_1} \not\subseteq [\![T_2]\!]_{D_2}$. We omit a justification which could be similar to the one presented in [15].

Example 22. *Consider the Type Definitions from the Example 20: $D = \{\, A{\to}l[B|C],\ B{\to}l[A^+],\ C{\to}m[\,]\,\}$ and $D' = \{\, A'{\to}l[A'^*|C'],\ C'{\to}m[C'^*]\,\}$. To check whether $[\![A]\!]_D \subseteq [\![A']\!]_{D'}$, first we construct set $C(A, A')$ which is $\{(A, A'), (B, A'), (C, C')\}$. Then the second part of the algorithm checks if $L(l|m) \subseteq L(l^*|m)$, $L(l^+) \subseteq L(l^*|m)$ and $L(\epsilon) \subseteq L(m^*)$. Since all the checks give positive results, we conclude that $[\![A]\!]_D \subseteq [\![A']\!]_{D'}$.*

Notice that for a proper Type Definition D_2 and 1-unambiguous regular expressions [14] 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 EXPTIME-complete [24].

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, for example, 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 lefts out the aspect related to defining specific types of text nodes. Thus we do not discuss here in details 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 type constant 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 string, integer, float etc. In this case some additional mechanism must be developed for validation according to these types. This is however out of scope of this work.

To employ Type Definitions to specify attributes, one has to follow the way of representing XML documents by data terms (described in Section 2.1.1). The attributes of elements of a particular type are represented by a type name occurring at the beginning of the content model of the type. If all the attributes of elements of a particular type are optional the type name representing the attributes is followed by the question mark '?'. In our notation the type name representing attributes of elements of a type T is T_attr. Examples illustrating how attributes are represented by Type Definitions are presented in the following sections.

3.3.1 DTD

From the point of view of the formal language theory DTD is a local tree grammar (in the sense of [45]). Thus any set of documents which can be defined by DTD can be also defined by a proper Type Definition. 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 DTDs cannot define two different sets of elements with the same label the corresponding Type Definition is proper.

Example 23. *This is an example of a DTD:*

```
<!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:

$$\begin{array}{lll}
bib & \rightarrow & bib\,[\,book^*\,] \\
book & \rightarrow & book\,[\,book\_attr\ title\ (author^+ \mid editor^+)\,] \\
book\_attr & \rightarrow & attr\,\{\,book\_year\ book\_isbn^?\ book\_language\,\} \\
book\_year & \rightarrow & year\,[\,\mathrm{Text}\,] \\
book\_isbn & \rightarrow & isbn\,[\,\mathrm{Text}\,] \\
book\_language & \rightarrow & language\,[\,lang\,] \\
lang & \rightarrow & ''en'' \mid ''sw'' \mid ''pl'' \\
author & \rightarrow & author\,[\,last\ first\,] \\
editor & \rightarrow & editor\,[\,last\ first\,] \\
title & \rightarrow & title\,[\,\mathrm{Text}\,] \\
first & \rightarrow & first\,[\,\mathrm{Text}\,] \\
last & \rightarrow & last\,[\,\mathrm{Text}\,]
\end{array}$$

3.3.2 XML Schema

Generally, due to the *Element Declarations Consistent* requirement [1], XML Schema corresponds to a single type tree grammar [45] and can be represented by proper Type Definition. However some features of XML Schema (such as *xsi:type* mechanism which is explained later on) can cause problems and they make it impossible to represent XML Schema by proper Type Definition. In this subsection we discuss relation of XML Schema to Type Definitions focusing on the features of XML schemata whose transformations to Type Definitions may be not clear, problematic or impossible.

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

Example 24. *This is a fragment of XML Schema defining a set of elements named* book*:*

```
<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>
          <attribute name="isbn" type="string">
       </complexType>
    </element>
```

and a proper Type Definition defining a corresponding type Book*:*

$$\begin{array}{lll}
Book & \rightarrow & book\,[\,Book\_attr\;\; Title\;\; (Author \mid Editor)^{+}\,] \\
Book\_attr & \rightarrow & attr\,\{\,Book\_isbn\,\} \\
Book\_isbn & \rightarrow & isbn\,[\,\mathrm{Text}\,] \\
Author & \rightarrow & author\,[\,\mathrm{Text}\,] \\
Editor & \rightarrow & editor\,[\,\mathrm{Text}\,] \\
Title & \rightarrow & title\,[\,\mathrm{Text}\,]
\end{array}$$

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 of element definitions can be later used in different element definitions or as a basis for type derivation. For example, we can define a type *Book* and then use it in definition of an element *book*:

   ```
   <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"/>
   ```

   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 24.

2. XML Schema provides a type called *anyType* which is the most general type from which all simple and complex types are derived. *anyType* can be seen as a set of all XML documents. It is possible to use *anyType* like other types. This work extended the formalism of Type Definitions by introducing the type *Top*, which plays the same role as *anyType*.

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

4. Type derivation by restriction of a complex type is a declaration that the derived type is a subset of a base type. When a new type is derived, its full content model must be specified in such way 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 a 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 named content models that can be reused in multiple locations as fragments of content models. This can be also seen as 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.

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

<element name="comment" type="string"/>
<element name="authorComment" type="string"
         substitutionGroup="comment"/>
<element name="reviewerComment" type="string"
         substitutionGroup="comment"/>
```

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

$$\begin{array}{lll} Article & \rightarrow & article\,[\,Title\ Author\ (Comment \mid \\ & & \quad \mid aComment \mid rComment)\,] \\ Comment & \rightarrow & comment\,[\,Text\,] \\ aComment & \rightarrow & authorComment\,[\,Text\,] \\ rComment & \rightarrow & reviewerComment\,[\,Text\,] \end{array}$$

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 *Book*1 and *Book*2. 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>
            <element name="author" type="string"
                 minOccurs="0" maxOccurs="unbounded"/>
            <element name="title" type="string"/>
         </sequence>
      </restriction>
   </complexContent>
</complexType>

<complexType name="Book2">
   <complexContent>
      <restriction base="Book">
         <sequence>
            <element name="editor" type="string"
                 minOccurs="0" maxOccurs="unbounded"/>
            <element name="title" type="string"/>
         </sequence>
      </restriction>
   </complexContent>
</complexType>

<element name="book" type="Book"/>

<element name="library">
   <complexType>
     <sequence>
       <element ref="book" minOccurs="0" maxOccurs="unbounded"/>
     </sequence>
   </complexType>
</element>
```

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. For example,

```
<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:

$$
\begin{array}{lll}
\mathit{Library} & \rightarrow & \mathit{book}\,[\,(\mathit{Book_1} \mid \mathit{Book_2})^*\,] \\
\mathit{Book_1} & \rightarrow & \mathit{book}\,[\,\mathit{Book_1\_attr}\ \mathit{Author}^*\ \mathit{Title}\,] \\
\mathit{Book_2} & \rightarrow & \mathit{book}\,[\,\mathit{Book_2\_attr}\ \mathit{Editor}^*\ \mathit{Title}\,] \\
\mathit{Book_1\_attr} & \rightarrow & \mathit{attr}\,\{\,\mathit{Type_1}\,\} \\
\mathit{Book_2\_attr} & \rightarrow & \mathit{attr}\,\{\,\mathit{Type_2}\,\} \\
\mathit{Author} & \rightarrow & \mathit{author}\,[\,\mathrm{Text}\,] \\
\mathit{Editor} & \rightarrow & \mathit{editor}\,[\,\mathrm{Text}\,] \\
\mathit{Type_1} & \rightarrow & \mathit{xsi{:}type}\,\{\,\mathit{Text_1}\,\} \\
\mathit{Type_2} & \rightarrow & \mathit{xsi{:}type}\,\{\,\mathit{Text_2}\,\} \\
\mathit{Text_1} & \rightarrow & \text{"}\mathit{Book_1}\text{"} \\
\mathit{Text_2} & \rightarrow & \text{"}\mathit{Book_2}\text{"}
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{Library} & \rightarrow & \mathit{book}\,[\,\mathit{Book}^*\,] \\
\mathit{Book} & \rightarrow & \mathit{book}\,[\,\mathit{Book\_attr}\ (\mathit{Author} \mid \mathit{Editor})^*\ \mathit{Title}\,] \\
\mathit{Book\_attr} & \rightarrow & \mathit{attr}\,\{\,\mathit{Type}\,\} \\
\mathit{Type} & \rightarrow & \mathit{xsi{:}type}\,\{\,\mathit{Text_1} \mid \mathit{Text_2}\,\} \\
\mathit{Text_1} & \rightarrow & \text{"}\mathit{Book_1}\text{"} \\
\mathit{Text_2} & \rightarrow & \text{"}\mathit{Book_2}\text{"} \\
\mathit{Author} & \rightarrow & \mathit{author}\,[\,\mathrm{Text}\,] \\
\mathit{Editor} & \rightarrow & \mathit{editor}\,[\,\mathrm{Text}\,]
\end{array}
$$

Note that the type *Library* defined in this way is a superset of the corresponding set defined by the schema. This is because elements of type *Library* can contain *book* elements with both *authors* and *editors* elements.

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

```
First = element first { text }
Second = element second { text }
Author = element author { First, Second }
Editor = element editor { First, Second }

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

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

$$
\begin{array}{lll}
\mathit{Book} & \rightarrow & \mathit{book}\,[\,\mathit{Book\_attr}\ \ \mathit{Title}\ (\mathit{Author}^{+} \mid \mathit{Editor}^{+})\,] \\
\mathit{Book\_attr} & \rightarrow & \mathit{attr}\,\{\,\mathit{Book\_isbn}\,\} \\
\mathit{Book\_isbn} & \rightarrow & \mathit{isbn}\,[\,\mathrm{Text}\,] \\
\mathit{Author} & \rightarrow & \mathit{author}\,[\,\mathit{First}\ \ \mathit{Second}\,] \\
\mathit{Editor} & \rightarrow & \mathit{editor}\,[\,\mathit{First}\ \ \mathit{Second}\,] \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathrm{Text}\,] \\
\mathit{First} & \rightarrow & \mathit{first}\,[\,\mathrm{Text}\,] \\
\mathit{Second} & \rightarrow & \mathit{second}\,[\,\mathrm{Text}\,]
\end{array}
$$

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 *book* containing an attribute *isbn* and a list of authors or an attribute *publisher* and a list of editors:

```
Book = element book {
  (attribute isbn { text },
   element title { text },
   Author+)
  |
  (attribute publisher { text },
   element title { text },
   Editor+)
}
First = element first { text }
Second = element second { text }
Author = element author { First, Second }
Editor = element editor { First, Second }
```

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

$$
\begin{array}{lll}
\mathit{Book} & \rightarrow & \mathit{book}\,[\,(\mathit{Book\_attr}_{1}\ \ \mathit{Title}\ \ \mathit{Author}^{+}) \mid \\
 & & \qquad\qquad \mid (\mathit{Book\_attr}_{2}\ \ \mathit{Title}\ \ \mathit{Editor}^{+})\,] \\
\mathit{Book\_attr}_{1} & \rightarrow & \mathit{attr}\,\{\,\mathit{Book\_isbn}\,\}
\end{array}
$$

$$\begin{array}{lll}
\mathit{Book\_isbn} & \rightarrow & \mathit{isbn}\,[\,\mathrm{Text}\,] \\
\mathit{Book\_attr_2} & \rightarrow & \mathit{attr}\,\{\,\mathit{Book\_publisher}\,\} \\
\mathit{Book\_publisher} & \rightarrow & \mathit{publisher}\,[\,\mathrm{Text}\,] \\
\mathit{Author} & \rightarrow & \mathit{author}\,[\,\mathit{First}\ \ \mathit{Second}\,] \\
\mathit{Editor} & \rightarrow & \mathit{editor}\,[\,\mathit{First}\ \ \mathit{Second}\,] \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathrm{Text}\,] \\
\mathit{First} & \rightarrow & \mathit{first}\,[\,\mathrm{Text}\,] \\
\mathit{Second} & \rightarrow & \mathit{second}\,[\,\mathrm{Text}\,]
\end{array}$$

An interesting feature of Relax NG are co-occurrence constraints. They allow to choose a 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*:

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

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

$$\begin{array}{lll}
\mathit{Bibliography} & \rightarrow & \mathit{bibliography}\,[\,\mathit{Entry}_1^*\ \ \mathit{Entry}_2^*\,] \\
\mathit{Entry_1} & \rightarrow & \mathit{entry}\,[\,\mathit{Entry_1\_attr}\ \ \mathit{Author}\ \ \mathit{Title}\ \ \mathit{Journal}\,] \\
\mathit{Entry_1\_attr} & \rightarrow & \mathit{attr}\,\{\,\mathit{Entry_1\_type}\,\} \\
\mathit{Entry_1\_type} & \rightarrow & \mathit{type}\,[\,\mathit{Article}\,] \\
\mathit{Article} & \rightarrow & \text{"}\mathit{article}\text{"} \\
\mathit{Entry_2} & \rightarrow & \mathit{entry}\,[\,\mathit{Entry_2\_attr}\ \ \mathit{Author}\ \ \mathit{Title}\ \ \mathit{BookTitle}\,] \\
\mathit{Entry_2\_attr} & \rightarrow & \mathit{attr}\,\{\,\mathit{Entry_2\_type}\,\} \\
\mathit{Entry_2\_type} & \rightarrow & \mathit{type}\,[\,\mathit{Proceedings}\,] \\
\mathit{Proceedings} & \rightarrow & \text{"}\mathit{inProceedings}\text{"} \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathrm{Text}\,] \\
\mathit{Author} & \rightarrow & \mathit{author}\,[\,\mathrm{Text}\,] \\
\mathit{BookTitle} & \rightarrow & \mathit{bookTitle}\,[\,\mathrm{Text}\,]
\end{array}$$

There is another way to construct a Type Definition equivalent to the Relax NG schema from above. The Type Definition could define only one type corresponding to the element *entry* and the content model of the type would be a disjunction of the content models of the two types (from the previous Type Definition) corresponding to the element *entry*. However, the new Type Definition would again be non proper.

Another thing that cannot be used in a straightforward way in Type Definitions is the *interleave* operator &. The operator can be used in a content model in addition to the standard operators from regular expressions. The operator can be used to specify 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 [41] 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

```
Book = element book {
   element title { text },
   element * { text } }
```

defines an element *book* containing a *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 Program Results

This chapter presents a method of type inference for results of Xcerpt programs. The method is presented formally for a substantial fragment of Xcerpt whose semantics was presented in Section 2.1.2.

First, we present a way of performing type inference for single Xcerpt rules. This is presented on an abstract level using typing rules which are based on the syntax of Xcerpt rules. Then we describe a way of using the single rule type inference method for type inference for programs. We also introduce theorems expressing soundness of the type system; we provide their proofs in Appendix A.1.

The chapter also provides a practical algorithm for type inference for single query rules which is a concretization of the typing rules. Additionally it provides typing rules for most of the remaining Xcerpt constructs i.e. the constructs that are not included in the considered Xcerpt fragment. It shows how type analysis can be used to determine dependencies between rules in a program. It also discusses relations between errors in programs and results of type inference and type checking.

4.1 Motivation

The main goal of the type system is to infer a type of program results given a specification of the types of input data (e.g. XML documents that are queried). The inferred result type is a superset of the set of results of the program. The type analysis can be useful for the programmer for finding errors in a program or to assure that the program is correct wrt. a type specification.

Type inference for a program requires a specification of the types of input data. However if such specification is not given by the programmer it is assumed that input data is of type *Top*. In such case, usually a less precise approximation of the set of program results is inferred by the system. The main purposes the type system can serve, depend on whether a specification of the result type, called specified result type, for a program is given (i.e. the specification can be provided by the programmer). If such a specification is missing usage of the type system is limited to the following:

- The programmer 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. Formally such a program is correct with respect to the type specification. Practically this is another kind of error.
- Inferred result types for the rules of the program can be used to automatically determine dependencies between rules. This knowledge can be used e.g. for optimization of evaluation of programs.

Moreover, if a specification of the required result type is provided, it can be used for the following purposes:

- 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 type specification.
- It can be checked whether the intersection of the inferred result type and the specified result type is not empty. If it is empty then the program will not produce any results of the required type.

Section 4.4 presents further discussion on applying the type system to discovering and locating errors in programs

4.2 Type Inference for Xcerpt

The section presents a way of performing type inference for Xcerpt programs. The method we present can be seen as a descriptive type system: 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 (or roughly approximated by *Top*), a way of inferring a type of results of a program is presented.

First we introduce a notion of variable-type mappings. Then we present typing rules which are used for typing various syntactic Xcerpt constructs. They allow to infer a result type for a rule given a type of intermediate data queried by the rule. Then we present a way of using the typing rules to infer a type of results of an Xcerpt program. The presented method can be seen as an abstract version of a an algorithm for type inference for Xcerpt programs. A precise algorithm can be easily developed based on the presented method.

4.2.1 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 $\Gamma\colon V \to \mathcal{E}$ (called a *variable-type mapping*), where V is the set of variables occurring in the considered query rule and $\mathcal{E}$ is a set of expressions. $\mathcal{E}$ contains 0, 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: $[\![0]\!] = \emptyset$, $[\![T]\!] = [\![T]\!]_D$ 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 $\Gamma\colon V \to \mathcal{E}$ is

$$\mathit{substitutions}_D(\Gamma) = \{\, \theta \mid \forall_{X \in V}\ X\theta \in [\![\Gamma(X)]\!]\, \}.$$

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

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

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

We will not distinguish between expressions $T \cap \mathit{Top}$ and T, between $T \cap 0$ and 0, and between $T \cap U$ and $U \cap T$ (where $T, U \in \mathcal{E}$).

For any $\Gamma_1, \Gamma_2\colon V \to \mathcal{E}$ we introduce $\Gamma_1 \cap \Gamma_2\colon V \to \mathcal{E}$ 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 $\Gamma\colon V \to \mathcal{E}$.

Inclusion of types induces a pre-order $\sqsubseteq$ on the mappings from $V \to \mathcal{E}$, as follows. If Γ and Γ' 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 $\mathit{substitutions}(\Gamma) \subseteq \mathit{substitutions}(\Gamma')$, 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 $V \to \mathcal{E}$. For such sets Ψ_1 and Ψ_2 we define:

$$\begin{array}{rcl} \Psi_1 \sqcap \Psi_2 & = & \{\Gamma_1 \cap \Gamma_2 \mid \Gamma_1 \in \Psi_1, \Gamma_2 \in \Psi_2\}, \\ \Psi_1 \sqcup \Psi_2 & = & \Psi_1 \cup \Psi_2. \end{array}$$

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

4.2.2 Typing of Query Rules

Here we present typing rules for the syntactic constructs of Xcerpt: query terms, queries, construct terms and query rules. The rules abstract from lower level details and may be seen as an abstraction of an algorithm for type inference. They are similar to proof rules of logic, rules used in operational semantics [47], and those used in prescriptive typing [20]. Employing rules makes it possible to describe type inference in a formal and concise way.

The typing rules were introduced earlier in a joint work [11]. However, there are some minor changes in the rules presented here with respect to the rules in [11]. For example, the errors reported in the errata for [11] are fixed and the type *Top* is handled.

Let p be a query rule which may query intermediate results of the program (i.e. results of the rules of the program) as well as external resources (e.g. XML data). We assume that the intermediate results queried by p are of a type expressed by a set of type names U defined by a Type Definition D. We also assume that specifications of types of the external resources (such as DTD's which can be translated into Type Definitions) are given by a mapping *type*. The mapping associates each resource r occurring in p with a type $T = \mathit{type}(r)$ defined by D. The type contains the data term $\delta(r)$ referred to by r (i.e. $\delta(r) \in [\![T]\!]$). If a type specification for a resource r is missing then $\mathit{type}(r) = \mathit{Top}$.

Query terms

The rules in this subsection provide a way to derive variable-type mappings for a query term given a type of data terms to which the query term is applied. They derive facts of the form $D \vdash q : T \triangleright \Gamma$, where D is a Type Definition, q a query term, T a type name, and Γ a variable-type mapping, whose domain is the set of variables occurring in the considered query rule. The intention is that if q is applied to a data term $d \in [\![T]\!]$ then the resulting answer substitution is in *substitutions*(Γ) for some Γ such that $D \vdash q : T \triangleright \Gamma$ can be derived.

A typing rule has a number of premises (written above the solid line) and one conclusion (written below the solid line). The rules may also have a number of conditions (written below the rule) that have to be fulfilled whenever the rule is applied.

$$\frac{b \in [\![T]\!]}{D \vdash b : T \triangleright \Gamma} \qquad \text{(Const)}$$

where b is a basic constant.

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

$$\frac{\Gamma \sqsubseteq [X \mapsto T]}{D \vdash X : T \triangleright \Gamma} \qquad (\textsc{Var})$$

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 $[\![T']\!]_D \subseteq [\![T]\!]_D$.

$$\frac{D \vdash q : T \triangleright \Gamma \qquad \Gamma \sqsubseteq [X \mapsto T]}{D \vdash X \leadsto q : T \triangleright \Gamma} \qquad (\textsc{As})$$

$$\frac{D \vdash q : T \triangleright \Gamma}{D \vdash \texttt{desc}\ q : T \triangleright \Gamma} \qquad (\textsc{Descendant})$$

$$\frac{D \vdash \texttt{desc}\ q : T' \triangleright \Gamma}{D \vdash \texttt{desc}\ q : T \triangleright \Gamma} \qquad (\textsc{Descendant Rec})$$

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

$$\frac{D \vdash q_1 : T_1 \triangleright \Gamma \ \cdots \ D \vdash q_n : T_n \triangleright \Gamma}{D \vdash l\ \alpha q_1, \cdots, q_n \beta : T \triangleright \Gamma} \qquad (\textsc{Pattern})$$

where $T = T_1 = \ldots = T_n = \mathit{Top}$, or
the rule for T in D is of the form $T \to l[\,r\,]$
or it is of the form $T \to l\{\,r\,\}$ and ($\alpha\beta = \{\}$ 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 $\alpha\beta = [[\,]]$,
$T_1 \cdots T_n \in \mathit{perm}(L(r))$ if $\alpha\beta = \{\}$,
$T_1 \cdots T_n \in \mathit{perm}(L(s))$ if $\alpha\beta = \{\{\}\}$.

We explain the fact that given a query term $q = l\alpha q_1, \ldots, q_n\beta$, the typing rule (Pattern) requires the same variable-type mapping Γ to be obtained for all the query terms $q_1, \ldots, q_n$. (A similar fact will hold for the typing rule (And Query) in the next subsection.) Obviously, typing rules may produce different Γ_i for each q_i ($i = 1, \ldots, n$). However, (due to the rules (Var) and (As)) they can produce also any "smaller" mapping Γ'_i for each q_i i.e. $\Gamma'_i \sqsubseteq \Gamma_i$. In particular, each Γ'_i may be $\Gamma'_i = \Gamma_1 \cap \ldots \cap \Gamma_n = \Gamma$.

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.

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

$$\frac{D \vdash q : T \triangleright \Gamma \qquad T \in U}{D \vdash q : U \triangleright \Gamma} \qquad \text{(Query Term)}$$

where q is a query term.

$$\frac{D \vdash q : T \triangleright \Gamma}{D \vdash \mathtt{in}(r, q) : U \triangleright \Gamma} \quad \text{(Targeted Query Term)}$$

where $\mathit{type}(r) = T$.

$$\frac{D \vdash Q_1 : U \triangleright \Gamma \quad \cdots \quad D \vdash Q_n : U \triangleright \Gamma}{D \vdash \mathtt{and}(Q_1, \ldots, Q_n) : U \triangleright \Gamma} \qquad \text{(And Query)}$$

$$\frac{D \vdash Q_i : U \triangleright \Gamma}{D \vdash \mathtt{or}(Q_1, \ldots, Q_i, \ldots, Q_n) : U \triangleright \Gamma} \qquad \text{(Or Query)}$$

where $1 \leq i \leq n$.

Construct terms

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

Definition 19. *Given a Type Definition D, a set Ψ of variable-type mappings and a set V of variables, such that $V \subseteq \mathit{dom}(\Gamma)$ and $\mathit{substitutions}(\Gamma) \neq \emptyset$ 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 $\overset{*}{\sim}_V$ of $\sim_V$ is denoted by $\Psi/_{\overset{*}{\sim}_V}$.*

The following rules allow to derive facts of the form $D \vdash c : \Psi \triangleright s$, where c is a construct term, Ψ 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 the application of a substitution set Θ to c results in a data term sequence $\Theta(c) = d_1, \ldots, d_n$ and $\Theta \subseteq \mathit{substitutions}(\Psi)$ then $D \vdash c : \Psi \triangleright s$ can be derived such that each $d_i \in [\![T_i]\!]$ and $T_1 \cdots T_n \in L(s)$. For correctness of the rules it is required that for any $\Gamma \in \Psi$, $\mathit{substitutions}(\Gamma) \neq \emptyset$ and for any $\Gamma_1, \Gamma_2 \in \Psi$, $\Gamma_1 \overset{*}{\sim}_{FV(c)} \Gamma_2$.

$$\frac{(T_b \to b) \in D}{D \vdash b : \Psi \triangleright T_b} \qquad (\textsc{Const})$$

where b is a basic constant.

$$\frac{[\![T_1]\!] = [\![\Gamma_1(X)]\!] \quad \cdots \quad [\![T_n]\!] = [\![\Gamma_n(X)]\!]}{D \vdash\ X : \{\Gamma_1, \ldots, \Gamma_n\} \triangleright T_1 \mid \cdots \mid T_n} \qquad (\textsc{Var})$$

where $T_1, \ldots, T_n$ are type names.

$$\frac{D \vdash c_1 : \Psi \triangleright s_1 \quad \cdots \quad D \vdash c_n : \Psi \triangleright s_n \quad (T_c \to l\alpha\, r\, \beta) \in D}{D \vdash l\alpha c_1, \ldots, c_n\beta : \Psi \triangleright T_c} \qquad (\textsc{Pattern})$$

where if $\alpha\beta = [\,]$ then $r = s_1 \cdots s_n$,
otherwise, r is a multiplicity list approximating
the regular expression $s_1 \cdots s_n$ (i.e. $[\![s_1 \cdots s_n]\!] \subseteq \mathit{perm}([\![r]\!])$) and
obtained using the algorithm from the next subsection "Dealing
with multiplicity lists".

$$\frac{D \vdash c : \Psi_1 \triangleright s_1 \quad \cdots \quad D \vdash c : \Psi_n \triangleright s_n \quad \{\Psi_1, \ldots, \Psi_n\} = \Psi/_{\overset{*}{\sim}_{FV(c)}}}{D \vdash \texttt{all}\ c : \Psi \triangleright (s_1 \mid \cdots \mid s_n)^+} \qquad (\textsc{All})$$

$$\frac{D \vdash c : \Psi_1 \triangleright s_1 \quad \cdots \quad D \vdash c : \Psi_n \triangleright s_n \quad \{\Psi_1, \ldots, \Psi_n\} = \Psi/_{\overset{*}{\sim}_{FV(c)}}}{D \vdash \texttt{some}\ k\ c : \Psi \triangleright (s_1 \mid \cdots \mid s_n)^{(1:k)}} \qquad (\textsc{Some})$$

Note, that for construct terms not being of the form $\texttt{some}\,k\,c$ and $\texttt{all}\,c$ the derived facts are of the form $D \vdash c_n : \Psi \triangleright T_1|\cdots|T_n$, where $T_1, \ldots, T_n$ are type names.

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. This 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'.

Dealing with multiplicity lists. Here we discuss some issues related to handling multiplicity lists by the typing rules for construct terms.

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 $A_{i1} \cap \ldots \cap A_{ik_i}$, T_i is a type name representing the intersection of types $A_{i1}, \ldots, A_{ik_i}$. However, if D contains non intersectable multiplicity lists 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).

$$\frac{[\![T_1]\!] \supseteq [\![\Gamma_1(X)]\!] \quad \cdots \quad [\![T_n]\!] \supseteq [\![\Gamma_n(X)]\!]}{D \vdash\ X : \{\Gamma_1, \ldots, \Gamma_n\} \triangleright T_1 \mid \cdots \mid T_n} \quad (\textsc{Var Approx})$$

We assume that the types $T_1, \ldots, T_n$ in the rule (Var Approx) are computed by, first, approximating the multiplicity lists from D by intersectable multiplicity lists using the method described in Section 3.2.2 and then computing the needed intersections of types.

Now we present a way of construction of a multiplicity list approximating a regular expression $s_1 \cdots s_n$ used by the typing rule (Pattern) for construct terms. If $c_1, ..., c_n$ are rooted construct terms then $s_1, ..., s_n$ are type names. Thus $s_1 \cdots s_n$ is a multiplicity list and we take $r = s_1 \cdots s_n$. Moreover, if the labels of $c_1, ..., c_n$ are distinct then r is an intersectable multiplicity list. In a general case, each subexpression s_i of the regular expression $s_1 \cdots s_n$ is of the form $(e_1 | \cdots | e_m)^{(1:k)}$, where $m > 0$, $k > 0$ is a number or ∞ and each subexpression of e_j of s_i is a type name or it has the same form as s_i. Let $s'_1 \cdots s'_{n'}$ be a proper regular expression which is an approximation of the regular expression $s_1 \cdots s_n$ (i.e. $[\![s_1 \cdots s_n]\!] \subseteq [\![s'_1 \cdots s'_{n'}]\!]$) and which is obtained using the algorithm from Section 3.1.1. Then each s'_i is of the form $(e_1 | \cdots | e_m)^{(1:k)}$ as above. Moreover, the intersection of any pair of types represented by distinct type names occurring in $s'_1 \cdots s'_{n'}$ is empty.

To approximate $s_1 \cdots s_n$ by a multiplicity list, for each type name T appearing in $s'_1 \cdots s'_{n'}$ we need to count the minimal number l_T and maximal number k_T of its occurrences in the elements of the language $L(s'_1 \cdots s'_{n'})$. To do that we transform each s'_i of the form $(a_1^{(l_1:k_1)} | \cdots | a_m^{(l_m:k_m)})^{(l:k)}$ into a regular expression $s''_i = (a_1^{(l'_1:k_1 * k)} \cdots a_m^{(l'_m:k_m * k)})$, where $l'_j = l * l_j$ if $m = 1$ and $l'_j = 0$ if $m > 0$. The minimal and the maximal numbers of occurrences of each type name are the same in the strings of the language $L(s'_i)$

and $L(s''_i)$. We proceed with the same transformation for each subexpression $a_j^{(l'_j:k'_j)}$ of s''_i. In this way we obtain an expression s'''_i of the form $U_1^{(l_{U_1}:k_{U_1})} \cdots U_p^{(l_{U_p}:k_{U_p})}$, where each U_j is a type name. Now we concatenate the expressions s'''_i, for $i = 1, \dots, n'$ and replace multiple occurrences of the same type name U_j by one occurrence for which the values of k_{U_j} and l_{U_j} are sums of the corresponding values for different occurrences of U_j. The resulting expression r is a multiplicity list approximating the regular expression $s_1 \cdots s_n$ i.e. $[\![s_1 \cdots s_n]\!] \subseteq perm([\![r]\!])$.

Query Rules

For a given Type Definition D, query Q and a set U of types names, the rules for queries 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 20. *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 $\{\Gamma_1, \dots, \Gamma_n\}$ of variable-type mappings is* **complete** *for Q and W wrt. D if*

- $D \vdash Q : W \triangleright \Gamma_i$ *for* $i = 1, \dots, n$, *and*
- *if* $D \vdash Q : W \triangleright \Gamma$ *and* $substitutions(\Gamma) \neq \emptyset$, *then there exists* $i \in \{1, \dots, n\}$ *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 [47] 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 `desc` q
- at least one node labelled $D \vdash Q' : W' \triangleright \Gamma$ (for some W'), if Q' is of the form `desc` q

Let us construct a derivation tree for $D \vdash Q : W \triangleright \Gamma$. As Γ will be computed at the end we start the construction from a root labelled $D \vdash Q : W \triangleright$. From the conditions in the typing rules it follows that for each newly constructed node labelled $D \vdash Q' : W' \triangleright$ 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 $D \vdash$ `desc` $q : W' \triangleright$

(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 Γ 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 $T_{i1} \cap \ldots \cap T_{im_i}$, and T_{ij} occurs in S_i iff the condition $\Gamma \sqsubseteq [X_i \mapsto T_{ij}]$ 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 $\texttt{desc}\, q$, 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 Λ of trees which can be constructed in this way is finite (for given Q and W).

Consider an arbitrary derivation tree λ for $D \vdash Q : W \triangleright \Gamma$. If we remove from it iteratively parts of paths of the form $D \vdash \texttt{desc}\, q : W' \triangleright \Gamma$, ..., $D \vdash \texttt{desc}\, q : W' \triangleright \Gamma$ we will obtain a tree which is isomorphic to some tree $\lambda' \in \Lambda$. Moreover, for each node in λ labelled $D \vdash Q' : W' \triangleright \Gamma$ the corresponding node in λ' is labelled $D \vdash Q' : W' \triangleright \Gamma'$. Additionally, $\Gamma \sqsubseteq \Gamma'$, as Γ' is the most general variable type mapping satisfying the conditions of the tree λ. Thus, the set of variable-type mappings corresponding to the trees from Λ 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 $D \vdash (c \leftarrow Q) : U \triangleright s_1 \mid \cdots \mid s_n$ 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 set of data terms of a type $[\![U]\!]$ then we obtain results belonging to the set $[\![s_1 \mid \cdots \mid s_n]\!]$.

$$\frac{D \vdash c : \Psi_1 \triangleright s_1 \quad \cdots \quad D \vdash c : \Psi_n \triangleright s_n \qquad \{\Psi_1, \ldots, \Psi_n\} = \Psi/_{\overset{*}{\sim}_{FV(c)}}}{D \vdash (c \leftarrow Q) : U \triangleright s_1 \mid \cdots \mid s_n}$$

(Query Rule)

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

Note, that as a construct term c cannot be of the form $\texttt{some}\, k\, c'$ and $\texttt{all}\, c'$, the derived facts are of the form $D \vdash (c \leftarrow Q) : U \triangleright T_1 \mid \cdots \mid T_n$, where $T_1, \ldots, T_n$ are type names. The set of type names derived for a query rule p and a set of type names U will be denoted as $\mathit{resType}(p, U)$. Thus

$$\mathit{resType}(p, U) = \{T_1, \ldots, T_n\},$$

where $D \vdash p : U \triangleright T_1 \mid \cdots \mid T_n$.

Example 25. *Consider a Type Definition* $D = \{\, T \to l[A^* B\, C],\ A \to \text{"a"},\ B \to \text{"b"},\ C \to \text{"c"},\ R_1 \to a[A^+ A],\ R_2 \to a[A^+ B],\ R_3 \to a[(A \mid B)^+ C] \,\}$ *and the query rule*

$$a\,[\,\texttt{all}\, X,\, Y\,] \;\leftarrow\; l\,[[\, X,\, Y\,]]$$

abbreviated as $c_0 \leftarrow q$*. 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* Ψ_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* $FV(c_0) = \{Y\}$*,* $\Psi_0/\overset{*}{\sim}_{FV(c_0)} = \{\Psi_1, \Psi_2, \Psi_3\}$*, where* $\Psi_1 = \{\Gamma_1\}$*,* $\Psi_2 = \{\Gamma_2\}$*,* $\Psi_3 = \{\Gamma_3, \Gamma_4\}$*. Now we apply each of* Ψ_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* $D \vdash c_0{\leftarrow}q : U \triangleright R_1 \mid R_2 \mid R_3$*. 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* $[\![R_1 \mid R_2 \mid R_3]\!]$*.*

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 type 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 p and a set of type names U, is a correct approximation of the set of results for p and any set Z of data terms of the type denoted by U i.e. $res(p, Z) \subseteq [\![resType(p, U)]\!]$.

Theorem 1 (Soundness of type inference for a rule)**.** *Let* D *be a Type Definition and* p *be a query rule, where for each targeted query term* $\mathtt{in}(r, q)$ *in* p *there is a type name* $T = type(r)$ *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 p *and* Z *exists then there exist* s *and* D' *such that* $D' \supseteq D$ *and* $D' \vdash p : U \triangleright s$*.*

If $D \vdash p : U \triangleright s$ *and* d *is a result for* p *and* Z*, then* $d \in [\![s]\!]$*.*

Proof. See Appendix A.1.1. □

4.2.3 Typing of Programs

In the previous sections we presented a method for type inference for a query rule given a type of intermediate data queried by the rule. The method allows to compute the set $resType(p, U)$ of type names for a rule p and a type U of intermediate data (produced by rules of the program). Here we show how the method can be used in a context of a program where for a particular rule the type U is not known. Thus we present a way of typing Xcerpt programs instead of single rules.

The following algorithm is based on the fixed point semantics of Xcerpt expressed by the Definitions 11 and 15. It iteratively computes types of intermediate results of query rules given a type of intermediate results obtained in the previous step (in the first iteration the type of intermediate results is empty). This process is repeated until a fixed point is reached i.e. the type of intermediate results does not change in the consecutive steps.

Definition 21 (Immediate consequence operator for rule result types)**.** *Let P be a set of Xcerpt query rules. T_P is a function defined on sets of type names such that*

$$T_P(U) = \bigcup_{p \in P} resType(p, U).$$

Our proofs of Theorems 2, 3 below are based on monotonicity of T_P, which implies $[\![T_P^i(\emptyset)]\!] \subseteq [\![T_P^j(\emptyset)]\!]$ for $i \leq j$. Sufficient conditions for monotonicity (Corollary 1 in Appendix A.1.2) are:

- no grouping constructs in the rules of P,
- if the head of a rule from P contains a construct term $l\{c_1, \ldots, c_n\}$ then $c_1, \ldots, c_n$ are rooted construct terms with distinct labels,
- the type names occurring in the argument of T_P and the type of each external resource occurring in P are specified by a Type Definition in which all multiplicity lists are intersectable.

We conjecture that Theorems 2, 3 hold in a more general case.

Theorem 2. *Let $\mathcal{P} = (P', G)$ be an Xcerpt program and $P = P' \backslash G$. Assume that T_P is monotonic. If d is a result of a rule p in P' then there exists $i > 0$ such that*

$$d \in [\![resType(p, T_P^i(\emptyset))]\!] \subseteq [\![T_{P'}(T_P^i(\emptyset))]\!].$$

If $[\![T_P^{j+1}(\emptyset)]\!] = [\![T_P^j(\emptyset)]\!]$ for some $j > 0$ then the above holds for $i = j$.

Proof. See Appendix A.1.2. □

Example 26. *Consider the Xcerpt program $\mathcal{P} = (\{p_1, p_2, g\}, \{g\})$ from Example 6 (in Section 2.1):*

$$\begin{array}{lll}
p_1 & = & fo[\,X,\,Y\,] \leftarrow \texttt{in}(\text{"}\mathit{file{:}addrBooks.xml}\text{"}, \\
 & & \quad \mathit{addr\text{-}books}\{\{\ \mathit{addr\text{-}book}\{\{ \\
 & & \qquad \mathit{owner}[X],\ \mathit{entry}\{\{\ \mathit{name}[Y],\ \mathit{relation}[\text{"}\mathit{friend}\text{"}\,]\,\}\}\,\}\}\,\}\}\,), \\
p_2 & = & \mathit{foaf}[\,X,\,Y\,] \leftarrow \texttt{or}(\mathit{fo}[\,X,\,Y\,],\ \texttt{and}(\mathit{fo}[\,X,\,Z\,],\ \mathit{foaf}[\,Z,\,Y\,]\,)\,), \\
g & = & \mathit{clique\text{-}of\text{-}friends}[\,\texttt{all}\,\mathit{foaf}\{\,X,\,Y\,\}\,] \leftarrow \mathit{foaf}[\,X,\,Y\,]
\end{array}$$

and the following Type Definition

$AddrBs$	$\rightarrow$ *addr-books*[$AddrB^*$]
$Owner$	$\rightarrow$ *owner*[Text]
$AddrB$	$\rightarrow$ *addr-book*[$Owner\ Entry^*$]
$Name$	$\rightarrow$ *name*[Text]
$Entry$	$\rightarrow$ *entry*[$Name\ Rel\ PhNo^*\ Address^?$]
Rel	$\rightarrow$ *relation*[$RelCat$]
$RelCat$	$\rightarrow$ *"friend"* | *"family"* | *"colleague"* | *"acquaintance"*
$PhNo$	$\rightarrow$ *phoneNo*[Text]
$Address$	$\rightarrow$ *address*[$Street\ ZipC^?\ City\ Country^?$]
$Street$	$\rightarrow$ *street*[Text]
$ZipC$	$\rightarrow$ *zip-code*[Text]
$City$	$\rightarrow$ *city*[Text]
$Country$	$\rightarrow$ *country*[Text]

Assume that the XML document queried by the rule p_1 is of the type AddrBs from the Type Definition i.e. type("file:addrBooks.xml") = AddrBs. We want to infer result types of the rules of the program.

We employ T_P where $P = \{p_1, p_2\}$. $T_P(\emptyset) = resType(p_1, \emptyset) \cup resType(p_2, \emptyset)$. The type inference algorithm returns $resType(p_1, \emptyset) = \{Fo\}$, where the type Fo is defined as $Fo \rightarrow fo[Text\ Text]$, and $resType(p_2, \emptyset) = \emptyset$ (as the rule p_2 does not query any external data). Thus $T_P(\emptyset) = \{Fo\}$.

$T_P^2(\emptyset) = T_P(T_P(\emptyset)) = T_P(\{Fo\}) = resType(p_1, \{Fo\}) \cup resType(p_2, \{Fo\}) = \{Fo\} \cup \{Foaf\} = \{Fo, Foaf\}$, where the rule for Foaf is $Foaf \rightarrow foaf[Text\ Text]$. Similarly $T_P^3(\emptyset) = \{Fo, Foaf\}$. Hence, $U^\infty = \{Fo, Foaf\}$ is a fixed point of T_P.

Now, we can obtain the final result types of the rules of $\mathcal{P}$: $resType(p_1, U^\infty) = \{Fo\}$, $resType(p_2, U^\infty) = \{Foaf\}$ and $resType(g, U^\infty) = \{Cof\}$, where the type Cof is defined as $Cof \rightarrow clique\text{-}of\text{-}friends\{Foaf^+\}$. □

Termination

There are two difficulties related to computing a fixed point of T_P. First, we have to check whether the current iteration of T_P produces a fixed point. Then, the iterations T_P may not terminate (all the sets $[\![T_P^i(\emptyset)]\!]$ may be distinct).

As $[\![T_P^i(\emptyset)]\!] \subseteq [\![T_P^{i+1}(\emptyset)]\!]$, for checking $[\![T_P^i(\emptyset)]\!] = [\![T_P^{i+1}(\emptyset)]\!]$ it is sufficient to check if $[\![T_P^i(\emptyset)]\!] \supseteq [\![T_P^{i+1}(\emptyset)]\!]$. This cannot be done efficiently (the task is EXPTIME-hard), and an algorithm is complicated. The algorithm for checking type inclusion presented in Section 3.2.3 is simple and efficient but it can only be applied to proper Type Definitions. This restriction is often not satisfied by the Type Definitions created by evaluating T_P.

Here we show that for non w-recursive programs the computing of a fixed point terminates and determining when the fixed point is obtained is easy.

Proposition 2. *Let P be a set of rules and $n > 0$. If $[\![T_P^{n-1}(\emptyset)]\!] \neq [\![T_P^n(\emptyset)]\!]$ then there exist $p_1, \ldots, p_n \in P$ such that $p_n \succ_w \cdots \succ_w p_1$.*

Proof. See Appendix A.1.2. □

From the Proposition it follows that if P is not w-recursive then the fixed point of T_P is reached in at most $|P|$ steps: $[\![T_P^i(\emptyset)]\!] = [\![T_P^{i+1}(\emptyset)]\!]$ for any $i \geq |P|$. Thus $T_P^{|P|}(\emptyset)$ is a fixed point of T_P. Moreover, if the longest chain $p_k \succ_w \cdots \succ_w p_1$ of rules in P contains k rules then the fixed point is reached in k steps.

Dealing with Recursion

Weak static recursion in a program $\mathcal{P}$ can prevent reaching a fixed point of T_P, thus it may make impossible inferring result types of query rules of $\mathcal{P}$. Now we show how this problem can be overcome.

One way of assuring that a fixed point will be reached in a w-recursive program $\mathcal{P}$ is breaking the cycles in the graph of relation $\succ_w$ of $\mathcal{P}$. This can be achieved by selecting a rule p belonging to the cycle, finding an approximation $[\![W_p]\!]$ of the set of results of p in some independent way (described later on), and removing p from the program. Instead, W_p is added to the type computed at each iteration. Thus instead of computing $T_P^i(\emptyset)$, we compute $\widehat{T}_P^i(\emptyset)$, where $\widehat{T}_P(U) = T_{P\setminus\{p\}}(U) \cup W_p$.

This approach can be applied to break all cycles detected in the graph. Let $\mathcal{P} = (P', G)$ and $P = P' \setminus G$. Assume that $P_0 = \{p_1, \ldots, p_m\}$ are rules removed from P to break all cycles. Assume also that a set of type names $W = W_{p_1} \cup \ldots \cup W_{p_m}$ is an approximation of their results, i.e. that if d is a result of p_i in $\mathcal{P}$ then $d \in [\![W_{p_i}]\!]$. Instead of T_P, we employ $\widehat{T}_P$, defined by $\widehat{T}_P(U) = T_{P\setminus P_0}(U) \cup W$. If all cycles are broken in the the program, i.e. there is no w-recursion in $P\setminus P_0$, then the fixed point U^∞ of $\widehat{T}_P$ will be found after at most $|P| - m$ iterations: $U^\infty = \bigcup_{i=1}^{\infty} \widehat{T}_P^i(\emptyset) = \bigcup_{i=1}^{|P|-m} \widehat{T}_P^i(\emptyset)$. (This follows from Proposition 2, which also holds for $\widehat{T}_P$ with basically the same proof.) The fixed point U^∞ approximates the set of results of the non goal rules of $\mathcal{P}$.

To make the approach work, we must know how to find a correct approximation W_p of the set of results of a rule p in $\mathcal{P}$. A rough approximation can be obtained based on the head h of p alone. If no variable occurs twice in h then the approximation is the type of all instances of h. Otherwise we take the set of all instances of h', where h' is h with each variable occurrence replaced by a distinct variable. For instance, such an approximation for a rule $c[\,a[\,X\,],\, X\,] \leftarrow Q$ is the type T defined by a Type Definition $D = \{\,T \rightarrow c[\,A\ Top\,],\ A \rightarrow a[\,Top\,]\,\}$.

A more precise approximation can be provided by the user. In this case it should be checked that the approximation is indeed correct. This can be achieved by checking whether $[\![resType(p, U^\infty)]\!] \subseteq [\![W_p]\!]$ for each employed approximation W_p of the results of a rule p. (The problems with inefficiency of inclusion checking, discussed in the previous section, can be avoided by requiring that the Type Definition provided by the user is proper.)

We presented a method of approximating the result sets of w-recursive programs. The following theorem establishes its correctness.

Theorem 3. *Let $\mathcal{P} = (P', G)$ be an Xcerpt program, $P = P' \setminus G$, and $P_0 \subseteq P$ such that $P \setminus P_0$ is not w-recursive. Assume that T_P is monotonic. Let W be a set of type names, and let $\widehat{T}_P(U) = T_{P\setminus P_0}(U) \cup W$ for any set U of type names. Let $U^\infty = \widehat{T}_P^k(\emptyset)$ be a fixed point of $\widehat{T}_P$ (i.e. $[\![\widehat{T}_P(U^\infty)]\!] = [\![U^\infty]\!]$). If $[\![resType(p, U^\infty)]\!] \subseteq [\![W]\!]$ for each $p \in P_0$ then*

$$\begin{array}{ll} d \in [\![resType(p, U^\infty)]\!] \subseteq [\![U^\infty]\!] & \text{for any result } d \text{ of a rule } p \in P, \\ d \in [\![resType(p, U^\infty)]\!] \subseteq [\![T_{P'}(U^\infty)]\!] & \text{for any result } d \text{ of a rule } p \in P', \\ [\![T_P(U^\infty)]\!] \subseteq [\![U^\infty]\!] \ \text{ and } \ [\![T_P^j(\emptyset)]\!] \subseteq [\![U^\infty]\!] \text{ for any } j > 0. & \end{array}$$

Moreover, U^∞ in the last three lines may be replaced by $T_P^j(U^\infty)$, for any $j > 0$.

Proof. See Appendix A.1.2. □

It follows from the theorem that $\cdots \subseteq [\![T_P^i(\emptyset)]\!] \subseteq [\![T_P^{i+1}(\emptyset)]\!] \subseteq \cdots \subseteq [\![T_P^{j+1}(U^\infty)]\!] \subseteq [\![T_P^j(U^\infty)]\!] \subseteq \cdots$.

The theorem shows a way of more precise approximation of the set of program results. After obtaining a fixed point U^∞ of $\widehat{T}_P$, we iteratively apply T_P a few times. An intuitive explanation is that in U^∞ the approximation of the results of a rule $p \in P_0$ is the same as that given by W. Analyzing the results of p applied to the data from $[\![U^\infty]\!]$ may exclude some data terms from $[\![W]\!]$. Thus it may improve the approximation of results of p, which in turn may improve the approximation of results of the rules which w-depend on p.

Example 27. *Consider an Xcerpt program $\mathcal{P} = (\{p_1, p_2, g\}, \{g\})$, where*

$$\begin{array}{rcl} p_1 & = & c[\,b[\,X\,]\,] \leftarrow \texttt{and}(\,c[\,X\,],\ \texttt{in}(\,res,\, \texttt{desc}\,X\ )\ ), \\ p_2 & = & c[\,X\,] \leftarrow \texttt{in}(\,res,\, b[[\,a[\,X\,]\,]]\,), \\ g & = & r[\,\texttt{all}\ X\,] \leftarrow\ c[\,X\,] \end{array}$$

and a Type Definition $D = \{\,A \to a[\,Text\,], T \to b[\,(A \,|\, T \mid Text)^*\,]\}$. Assume that the type of the resource res *is T.*

We want to approximate the set of results of $\mathcal{P}$. We show that a fixed point cannot be obtained by computing $T_P^i(\emptyset)$ where $P = \{p_1, p_2\}$. Then we apply Theorem 3. As $[\![W]\!]$ we first use the set of all instances of the head of the w-recursive rule p_1. Then we show how a better approximation can be obtained by employing a more precise initial specification W.

We first find that, independently from U, $resType(p_2, U) = \{C_1\}$, where the rule for C_1 is $C_1 \to c[\,Text\,]$. This is because the query $\texttt{in}(\,res,\, b[[\,a[\,X\,]\,]]\,)$

binds X to a value from $[\![\,Text]\!]$. Thus $T_P(U) = resType(p_1, U) \cup resType(p_2, U) = resType(p_1, U) \cup \{C_1\}$.

Hence $T_P(\emptyset) = resType(p_1, \emptyset) \cup \{C_1\} = \emptyset \cup \{C_1\} = \{C_1\}$. Now $resType(p_1, \{C_1\}) = \{C_2\}$, where type C_2 is defined by the rules $C_2 \rightarrow c[\,B_1\,]$, $B_1 \rightarrow b[\,Text\,]$ (as the query $c[X]$ binds X to a value from $[\![\,Text]\!]$ and `in(` *res*, `desc` *X* `)` *binds X to a value from $[\![\{\,Text, A, T\}]\!]$). Hence $T_P(T_P(\emptyset)) = T_P(\{C_1\}) = \{C_2\} \cup \{C_1\}$.*

Generally we obtain $T_P^i(\emptyset) = \{C_1, \ldots, C_i\}$ ($i > 1$), with rules $C_j \rightarrow c[B_{j-1}]$, $B_j \rightarrow b[B_{j-1}]$ (for $j > 1$). All the sets $[\![T_P^i(\emptyset)]\!]$ are distinct and a fixed point will never be reached.

We can however approximate the results of $\mathcal{P}$ by applying Theorem 3. The program with p_1 removed is not w-recursive. The set of results of p_1 can be approximated by the set $[\![C_a]\!]$ of all the instances of the head of p_1; the type C_a is defined by rules $C_a \rightarrow c[\,B_a\,]$, $B_a \rightarrow b[\,Top\,]$. We look for a fixed point of $\widehat{T}_P$, where $\widehat{T}_P(U) = T_{\{p_2\}}(U) \cup \{C_a\} = resType(p_2, U) \cup \{C_a\}$. By the discussion following Proposition 2, the fixed point is $U^\infty = \widehat{T}_P^1(\emptyset) = \{C_1, C_a\}$. As an approximation of the set of results of $\mathcal{P}$ we obtain $resType(g, \{C_1, C_a\}) = \{R\}$ where type R is defined as $R \rightarrow r[\,(\,Text \mid B_a)^+\,]$.

To obtain a better approximation we can apply T_P to the set U^∞. $U_1^\infty = T_P(U^\infty) = resType(p_1, U^\infty) \cup \{C_1\} = \{C_1'\} \cup \{C_1\}$, where type C_1' is defined by the rules $C_1' \rightarrow c[\,B_1'\,]$, $B_1' \rightarrow b[\,Text \mid B'\,]$, $B' \rightarrow b[\,Text \mid A \mid T\,]$. This allows to obtain a more precise type of the goal rule which is $resType(g, U_1^\infty) = \{R_1\}$, where type R_1 is defined as $R_1 \rightarrow r[\,(\,Text \mid B_1')^+\,]$.

By applying T_P to U^∞ multiple times we can further improve the precision of the approximation. $U_i^\infty = T_P^i(U^\infty) = \{C_1, C_i'\}$, where type C_i' is defined by the rules $C_i' \rightarrow c[\,B_i'\,]$, $B_i' \rightarrow b[\,Text \mid B_{i-1}'\,]$ for $i > 1$. This produces a type R_i of results of $\mathcal{P}$ defined as $R_i \rightarrow r[\,(\,Text \mid B_i')^+\,]$.

The above approximations are obtained based on the automatic rough approximation C_a of the set of results of the rule p_1. However, the user can provide a more precise result type of the rule p_1 than C_a e.g. a type C_u defined by the rules $C_u \rightarrow c[\,B_u\,]$, $B_u \rightarrow b[\,Text \mid B_u \mid C_u\,]$. Based on this a fixed point of the operator $\widehat{T}_P'(U) = T_{\{p_2\}}(U) \cup \{C_u\}$ can be computed, which is $U_u^\infty = \{C_1, C_u\}$. To make sure that the approximation C_u provided by the user is correct we test whether $[\![resType(p_1, U_u^\infty)]\!] \subseteq [\![C_u]\!]$. $resType(p_1, U_u^\infty) = \{C\}$, where type C is defined by the rules $C \rightarrow c[\,B\,]$, $B \rightarrow b[\,Text \mid B\,]$. As $[\![C]\!] \subseteq [\![C_u]\!]$ the test is successful.

To improve the approximation U_u^∞ of the set of results of p_1, p_2 we can apply the operator T_P to U_u^∞. $U_{u_1}^\infty = T_P(U_u^\infty) = \{C_1, C\}$. Further applications of T_P to U_u^∞ provide the same results i.e. $T_P^i(U_u^\infty) = U_{u_1}^\infty$, for $i > 0$. Based on $U_{u_1}^\infty$ we obtain a precise type R_u of the goal rule which is defined as $R_u \rightarrow r[\,(\,Text \mid B)^+\,]$. □

4.2.4 Exactness of Type Inference

In this section we present reasons of inaccuracies of the type inference and discuss when the type inference is exact. Also the section suggests some ways of possible improvements in the type system.

Query terms

The set of variable type mappings Ψ produced by the typing rules for query terms expresses a superset Θ of the set of possible answers for a query term q. If q does not contain restricted variables (i.e. a construct $\rightsquigarrow$) then the set Θ 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 type names. Let q be a query term, T a type name from D, and $\Theta = \{\, \theta \,|\, D \vdash q : T \triangleright \Gamma, \theta \in substitutions_D(\Gamma) \,\}$. If q does not contain $\rightsquigarrow$ then each $\theta \in \Theta$ is an answer for q and some $d \in [\![T]\!]_D$.*

Proof. See Appendix A.2. $\square$

Example 28. *Consider a Type Definition $D = \{T \to l[A],\ A \to a[B^*],\ B \to \text{"}b\text{"}\}$ and a query term $q_1 = l[\,X \rightsquigarrow a[\,]\,]$. 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 $substitutions_D(\Gamma)$ 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. $\theta = \{X/a[\text{"}b\text{"}]\}$. In contrast, the same mapping Γ can be inferred for the query term $q_2 = l[\,X\,]$ and $substitutions_D(\Gamma)$ is the exact set of answers for q_2 and a database of type T.*

The paper [26] presents a way of avoiding approximations when handling query terms with a construct $\rightsquigarrow$.

Queries

The set of variable type mappings Ψ produced by the typing rules for queries expresses a superset Θ of the set of possible answers for a query Q. If Q does not contain restricted variables (i.e. a construct $\rightsquigarrow$) and multiple occurrences of the same resource (as an argument of a construct `in`(...)) under the scope of a construct `and`(...) then the set Θ is the exact set of answers.

Example 29. *Consider a Type Definition $D = \{A \to a[\,B|C\,],\ B \to b[\,Text\,],\ C \to c[\,Text\,]\}$ and a query $Q = \texttt{and}(\,\texttt{in}(r, a[\,b[\,X\,]\,]), \texttt{in}(r, a[\,c[\,X\,]\,]))$. Assuming that $type(r) = A$ the typing rules for query terms and queries allow to infer the fact $D \vdash Q : \emptyset \triangleright \Gamma$, where $\Gamma = [X \mapsto Text]$. The set $substitutions_D(\Gamma)$ is not the exact set of answers for Q and $\emptyset$ as $substitutions_D(\Gamma) \neq \emptyset$ and the set of answers for Q and $\emptyset$ is empty.*

Proposition 4. *Let D be a Type Definition without nullable type names, and whose content models do not contain useless type names. Let U be a set of type names from D, Q be a query and $\Theta = \{\, \theta \,|\, D \vdash Q : U \rhd \Gamma,\, \theta \in substitutions_D(\Gamma) \,\}$. Let $T_1, \ldots, T_n$ be type names in D such that $type(r_i) = T_i$ for each targeted query term $\texttt{in}(r_i, q_i)$ in Q ($i = 1, \ldots, n$). If Q does not contain $\leadsto$ and multiple occurrences of the same resource (as an argument of a construct* in$(\ldots)$*) under the scope of a construct* 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 set $Z \subseteq [\![U]\!]_D$ of data terms*

such that θ is an answer for Q' and Z, where Q' is Q with each targeted query term $\texttt{in}(r_i, q_i)$ replaced by a targeted query term $\texttt{in}(r'_i, q_i)$, such that $\delta(r'_i) = d_i$.

Proof. See Appendix A.2. □

Construct Terms and Query Rules

This section presents sources of inaccuracies 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, and $T_1, \ldots, T_n$ be type names in D such that $type(r_i) = T_i$ for each targeted query term $\texttt{in}(r_i, q_i)$ in Q. A type $[\![R]\!]_D$ 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. $d_r \in [\![R]\!]_D$
- 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 a result for $c \leftarrow Q'$ and Z, where Q' is Q with each targeted query term $\texttt{in}(r_i, q_i)$ replaced by a targeted query term $\texttt{in}(r'_i, q_i)$, such that $\delta(r'_i) = d_i$.

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 30. *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 Γ_1, Γ_2 belong to one equivalence class with respect to the free variables of $c[X, Y]$, the inferred 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 $t_1, t_2 \in [\![T_1]\!] \backslash [\![T_2]\!]$.*

The inaccuracies related to equivalence classes of variable-type mappings do not occur, if all equivalence classes that appear in the derivation of the inferred result type for a query rule, consist of one variable-type mapping only.

Unfortunately, this condition is not sufficient for the inferred result type to be exact. There is another reason of inexactness which is related to grouping constructs. For example, an inferred result type for a rule $c[\,$`all` $X\,] \leftarrow Q$ can be defined as $R \rightarrow c[\,T^+\,]$ for some T. A data term d of the type R can have the same data term appearing multiple times as a direct subterm of d. However, according to the Xcerpt semantics (Section 2.1.2) all direct subterms of each result d' of the rule are distinct. Thus e.g. a data term $c[\,$"a", "a"$\,]$ cannot be a result of the rule although it can be of type R.

Abandoning the constructs `all` and `some` would make it possible to create a simpler typing rule for query rules. Such a simpler typing rule allows to avoid the approximations related to equivalence classes:

$$\frac{D \vdash c : \Psi \triangleright s \quad D \vdash Q : U \triangleright \Gamma \quad \Psi = \{\Gamma\}}{D \vdash (c \leftarrow Q) : U \triangleright s} \quad \text{(Query Rule 2)}$$

where c does not contain constructs `all` and `some`.

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

Example 31. *Let $T_1, T_2, \Gamma_1, \Gamma_2, \Psi$ be defined as in Example 30. 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 Γ_1 and once for Γ_2. The first application of* (Query Rule*2*) *results in the result type R_1 defined as $R_1 \rightarrow c[\,T_1 T_2\,]$ and the second application results in the result type R_2 defined as $R_2 \rightarrow c[\,T_2 T_1\,]$. 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.*

Another reason of impreciseness of the inferred result type is the approximation introduced in the rule (Pattern). In the case where $\alpha\beta = \{\,\}$ in the rule, the regular expression $s_1 \cdots s_n$ must be approximated by a multiplicity list.

An inferred result type for a query rule p may be also inexact if a construct term in p contains multiple occurrences of the same variable. In such case each occurrence of 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. It may be impossible to represent such a set of results by our formalism.

Example 32. *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 $D = \{ T \rightarrow l[\mathrm{Text}] \}$. The inferred result type for the query rule would be R defined as $R \rightarrow c[\mathrm{Text}\,\mathrm{Text}]$. Although a data term $d = c[\text{"}text1\text{"}, \text{"}text2\text{"}]$ belongs to $[\![R]\!]$, d cannot be a result for the query rule as every result is a data term with two identical subterms.*

Another issue which may lead to inexactness of an inferred result type is a Type Definition containing non intersectable multiplicity lists. For such Type Definitions it may be needed to 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 inferred result type of a query rule to be exact. 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 $p = c \leftarrow Q$ wrt. $U, T_1, \ldots, T_n$ if

- the Type Definition D
  - does not contain non intersectable multiplicity lists,
  - does not contain content models with useless or nullable type names,
- the body Q of p
  - does not contain a construct $\rightsquigarrow$,
  - does not contain multiple occurrences of the same resource under the scope of a construct `and`(...),
- the head c of p
  - does not contain constructs `all` and `some`,
  - does not contain curly braces ({ }),
  - does not contain multiple occurrences of a variable,
- all equivalence classes that appear in the derivation of the inferred type consist of one variable-type mapping only.

The last condition is satisfied if (1) any two types defined by D, except *Top*, are disjoint, and (2) *Top* appears in no content model in D, and (3) for each external resource r appearing in Q, $type(r) \neq Top$.

The abovementioned conditions can be generalized for a program. The type inferred for a program is exact if the program is not recursive, and if the conditions are satisfied for the rules of the program and the Type Definition defining types of queried data. Moreover it is required that the Type Definition defining result types inferred for rules in a program also satisfies the abovementioned conditions for a Type Definition. The first of the conditions, namely, that the Type Definition does not contain non intersectable multiplicity lists, is satisfied if the heads of rules of the program do not contain curly braces ($\{\,\}$). In order to assure that the second condition (no content models with useless or nullable type names) is satisfied the presented type inference algorithm must be augmented by a procedure which eliminates useless and nullable type names from content models of the inferred types.

4.2.5 Type Inference Algorithm for Query Rules

We presented a type inference algorithm for programs given an algorithm for type inference for single rule. The latter was presented only in an abstract way by means of typing rules. Here we present a concrete algorithm for a single query rule, that is an implementation of the typing rules from Section 4.2.2. The algorithm computes the type of results $resType(p, U)$ of the query rule $p = c \leftarrow Q$ 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* for each resource r_i). All these types are defined by a given Type Definition D. The algorithm consists of two main steps. First, a complete set of variable-type mappings Ψ for Q and U must be found. Then, based on Ψ types of query rule results are built.

Computing a complete set of variable-type mappings

Here we describe a method for computing 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 $T' \in reachable(T)$ iff

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

Now we are ready to present the procedure *match*.

```
$match(q, T)$ :
    IF $q$ is a variable $X$ THEN
        return $\{[X \mapsto T]\}$
    IF $q$ is of the form $X \leadsto q'$
        return $\{[X \mapsto T]\} \sqcap match(q', T)$
    IF $q$ is of the form $desc\, q'$ THEN
        return $match(q', T) \sqcup \bigsqcup_{T' \in reachable(T)} match(q', T')$
    (Now $q$ is a rooted query term or a basic constant).
    IF $T = Top$ THEN return $\{\top\}$
    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 $q = l\alpha q_1 \cdots q_n \beta$ $(n \geq 0)$,
    IF $\{\,\}$ are the parentheses for $T$ and ($\alpha\beta = [\,]$ or $\alpha\beta = [[\,]]$) 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 $U|\epsilon$
    let $L' = \begin{cases} L(r), & \text{if } \alpha\beta = [\,] \\ perm(L(r), & \text{if } \alpha\beta = \{\,\} \\ L(s), & \text{if } \alpha\beta = [[\,]] \\ perm(L(s)), & \text{if } \alpha\beta = \{\{\,\}\} \end{cases}$
    return $\{\, \Gamma_1 \cap \ldots \cap \Gamma_n \mid T_1 \ldots T_n \in L',$
                $\Gamma_1 \in match(q_1, T_1), \ldots, \Gamma_n \in match(q_n, T_n) \,\}$
```

The procedure *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 are needed. Notice, that the elements of the sequence $T_1, \ldots, T_n$ are to be matched[1] with respective subterms of the query term $q = l\alpha q_1 \cdots q_n \beta$. If a subterm q_i is a rooted query term it can only match the type *Top* or a type whose label is the same as the root of q_i. Often, there will be only one or two type names occurring in r with a given label as we expect that the

[1]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.

algorithm will often deal with proper Type Definitions. 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 relatively 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. Our experiments show that the time of the computation is reasonable for up to four distinct variables among $q_1, \ldots, q_n$ when the corresponding content model consists of four distinct type names.

Example 33. *Consider a Type Definition* $D = \{T \rightarrow l\{T_1\, T_2^*\}, T_1 \rightarrow a[Text], T_2 \rightarrow b[Text]\}$ *and a query term* $q = l\{\{X \rightsquigarrow b[''s''],\ Y\}\}$. *We execute* $match(q, T)$. *In the first run of the procedure we obtain* $L' = perm(T_1^?\, T_2^*)$. *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* match *for relevant query terms and types:*

- $match(X \rightsquigarrow b[''s''], T_1)$ *and receive* $\emptyset$
- $match(Y, T_1)$ *and obtain* $\{[Y \rightarrow T_1]\}$
- $match(X \rightsquigarrow b[''s''], T_2)$ *and obtain* $\{[X \rightarrow T_2]\}$
- $match(Y, T_2)$ *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* $match(q, T)$ *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 *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 4.2.2. Thus, the set $\Psi = mappingSet(Q, U)$ 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 $type(r_i)$.

```
$mappingSet(Q, U)$ :
    IF $Q$ is of the form $\texttt{or}(Q_1, \ldots, Q_n)$ then
        return $mappingSet(Q_1, U) \sqcup \ldots \sqcup mappingSet(Q_n, U)$
    IF $Q$ is of the form $\texttt{and}(Q_1, \ldots, Q_n)$ then
        return $mappingSet(Q_1, U) \sqcap \ldots \sqcap mappingSet(Q_n, U)$
    IF $Q$ is of the form $\texttt{in}(r, q)$ then
        return $match(q, type(r))$
    IF $Q$ is a query term $q$ then
        return $\bigsqcup_{T \in U} match(q, T)$
```

The values of the mappings from $\Psi = mappingSet(Q, U)$ 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_Ψ of all such expressions

$$W_\Psi = \left\{ T_1 \cap \ldots \cap T_n \;\middle|\; \begin{array}{l} T_1 \cap \ldots \cap T_n = \Gamma(X),\ \Gamma \in \Psi,\ X \in V \\ n > 1,\ \text{each } T_i \text{ is a type name} \end{array} \right\}.$$

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

From the obtained set of mappings Ψ 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 Ψ' obtained in this way is still a complete set of variable type mappings. Moreover, for each $\Gamma \in \Psi'$, $substitutions(\Gamma) \neq \emptyset$.

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 Ψ of variable type mappings for Q and U is given. Moreover, all variable-type mappings from Ψ are of the form $[X_1 \mapsto T_1, \ldots, X_n \mapsto X_n]$ where $T_1, \ldots, T_n$ are not nullable type names defined by the Type Definition D.

First we present a way to obtain a set of equivalence classes $\Psi/_{\overset{*}{\sim}_V}$ given a set of variable type mappings $\Psi = \{\Gamma_1, \ldots, \Gamma_n\}$ and a set of variables V (such that $V \subseteq dom(\Gamma)$ for each $\Gamma \in \Psi$). We divide the set of mappings Ψ 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 Ψ_i and Ψ_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 the 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 set of sets of variable type mappings obtained in this way is the set of equivalence classes $\Psi/_{\overset{*}{\sim}_V}$. Assume that the method is implemented as a function $eqClasses(\Psi, V)$ returning the set $\Psi/_{\overset{*}{\sim}_V}$. 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 $buildType(c, \Psi_i)$ returns a regular type expression r_{Ψ_i}. The arguments of the function are a construct term c, a set of variable type mappings $\Psi' \in \Psi/_{\overset{*}{\sim}_{FV(c)}}$ (where $FV(c)$ stands for the set of the free variables of c) and a Type Definition D. Let $\Psi_1, \ldots, \Psi_n$ be the equivalence classes of Ψ. The function $buildType(c, \Psi_i)$ is called for each Ψ_i ($i = 1, \ldots, n$). The union of the types produced by all the calls $[\![r_{\Psi_1}|\cdots|r_{\Psi_n}]\!]$ is a superset of the set of results of the query rule $c \leftarrow Q$.

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 $define(N \to ...)$ adds a rule $N \to ...$ to the Type Definition, and that N is a new type name, not occurring elsewhere. In this way a Type Definition D_{Ψ_i} is constructed such that $D \subseteq D_{\Psi_i}$.

```
$buildType(c, \Psi')$ :
    IF $c$ is a basic constant THEN
        $define(T_c \to c)$
        return $T_c$
    IF $c$ is a variable THEN
        let $\{\Gamma_1, \ldots, \Gamma_n\} = \Psi'$
        let $T_i = \Gamma_i(c)$ for $i = 1, \ldots, n$
        return $(T_1|\ldots|T_n)$
    IF $c$ is of the form $l\,\alpha c_1, \ldots, c_n \beta$ THEN
        let $r_i = buildType(c_i, \Psi')$ for $i = 1, \ldots, n$
        $define(T_c \to l\,\alpha\, r_1 \ldots r_n\, \beta)$
        return $T_c$
    IF $c$ is of the form all $c$ THEN
        let $\{\Psi_1, \ldots, \Psi_n\} = eqClasses(\Psi', FV(c))$
        let $r_i = buildType(c, \Psi_i)$ for $i = 1, \ldots, n$
        return $(r_1|\ldots|r_n)^+$
    IF $c$ is of the form some $k$ $c$ THEN
        let $\{\Psi_1, \ldots, \Psi_n\} = eqClasses(\Psi', FV(c))$
        let $r_i = buildType(c, \Psi_i)$ for $i = 1, \ldots, n$
        return $(r_1|\ldots|r_n)^{(1:k)}$
```

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 $p = c \leftarrow Q$ is a subset of

$$R = \bigcup_{\Psi_i \in eqClasses(\Psi, FV(c))} [\![buildType(c, \Psi_i)]\!]$$

The conditions for the set R to be the exact set of results are specified in Section 4.2.4.

For a construct term c being the head of a query rule the function $buildType(c, \Psi_i)$ returns a regular type expression of the form $T_{i1}|\cdots|T_{ik_i}$. The set of results R is the union of types $T_{11}, \ldots, T_{1k_1}, \ldots, T_{n1}, \ldots, T_{nk_n}$ produced by the calls *buildType(c,*Ψ_i*)* (where $\{\Psi_1, \ldots, \Psi_n\} = \Psi/\overset{*}{\sim}_{FV(c)}$). Each of the types T_{ij} is defined by a Type Definition D_{Ψ_i}. However we can assume that the types T_{ij} are defined by one Type Definition D_Ψ which is a union of Type Definitions D_{Ψ_i} for $i = 1, \ldots, n$ (type name conflicts should be resolved by renaming type names). Assuming that $M = \{T_{11}, \ldots, T_{1k_1}, \ldots, T_{n1}, \ldots, T_{nk_n}\}$ the set of results R can be expressed as $[\![M]\!]_{D_\Psi}$. The inferred result type for a rule p is expressed as

$$resType(p, U) = \{T_{11}, \ldots, T_{1k_1}, \ldots, T_{n1}, \ldots, T_{nk_n}\}$$

In general, the newly constructed Type Definition D_Ψ is not proper. It may be impossible to describe R by a proper Type Definition. If needed, D_Ψ can be approximated by a proper Type Definition using the algorithm from Section 3.1.1.

Example 34. *Consider the query rule p:*

$$\begin{array}{l} result[\,name[TITLE],\ author[ARTIST]\,] \ \leftarrow \\ \qquad \texttt{in}\,(\text{"}http://www.example.com/cds.xml\text{"}, \\ \qquad\quad catalogue\{\{\ cd\{title[TITLE],\ artist[ARTIST]\,\}\}\}) \end{array}$$

We will abbreviate the rule p by $c \leftarrow Q$*, and Q by* $\texttt{in}(url, q)$*. Assume that* type*(url) = Catalogue defined by a Type Definition D:*

$$\begin{array}{l} Catalogue \rightarrow catalogue\,[\,Cd^*\,] \\ Cd \rightarrow cd\,[\,Title\ \ Artist^+\,] \\ Title \rightarrow title\,[\,\mathrm{Text}\,] \\ Artist \rightarrow artist\,[\,\mathrm{Text}\,] \end{array}$$

We want to use the presented algorithm to infer the type of the results for the query rule. First, we call 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* match$(q, Catalogue)$*. As a result* mappingSet *returns a set of mappings* $\Psi = \{\Gamma_1, \Gamma_2\}$ *where* $\Gamma_1 = [TITLE \rightarrow Artist, ARTIST \rightarrow Artist]$, $\Gamma_2 = [TITLE \rightarrow Title, ARTIST \rightarrow Artist]$*. The set of equivalence classes of* Ψ *is* $\{\{\Gamma_1\}, \{\Gamma_2\}\}$*. A call of* buildType$(c, \{\Gamma_1\})$ *results in extending the Type Definition D with the following definitions of types:*

$$\begin{array}{l} \mathit{Result_1} \rightarrow \mathit{result}\,[\,\mathit{Name_1}\ \mathit{Author_1}\,] \\ \mathit{Name_1} \rightarrow \mathit{name}\,[\,\mathit{Artist}\,] \\ \mathit{Author_1} \rightarrow \mathit{author}\,[\,\mathit{Artist}\,] \end{array}$$

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

$$\begin{array}{l} \mathit{Result_2} \rightarrow \mathit{result}\,[\,\mathit{Name_2}\ \mathit{Author_2}\,] \\ \mathit{Name_2} \rightarrow \mathit{name}\,[\,\mathit{Title}\,] \\ \mathit{Author_2} \rightarrow \mathit{author}\,[\,\mathit{Artist}\,] \end{array}$$

Thus, finally the inferred result type of the query rule is $\mathit{resType}(p, \emptyset) = \{\mathit{Result}_1, \mathit{Result}_2\}$*, where the types* $\mathit{Result}_1, \mathit{Result}_2$ *are defined by a Type Definition* D'*:*

$$\begin{array}{l} \mathit{Result_1} \rightarrow \mathit{result}\,[\,\mathit{Name_1}\ \mathit{Author}\,] \\ \mathit{Result_2} \rightarrow \mathit{result}\,[\,\mathit{Name_2}\ \mathit{Author}\,] \\ \mathit{Name_1} \rightarrow \mathit{name}\,[\,\mathit{Artist}\,] \\ \mathit{Name_2} \rightarrow \mathit{name}\,[\,\mathit{Title}\,] \\ \mathit{Author} \rightarrow \mathit{author}\,[\,\mathit{Artist}\,] \\ \mathit{Title} \rightarrow \mathit{title}\,[\,\mathit{Text}\,] \\ \mathit{Artist} \rightarrow \mathit{artist}\,[\,\mathit{Text}\,] \end{array}$$

4.2.6 Typing of Remaining Xcerpt Constructs

The type system defined formally in the previous sections is restricted to the fragment of Xcerpt whose semantics was presented in Section 2.1.2. Here we present a way of extending the type system to most of the remaining Xcerpt construct whose semantics we do not formally define in this thesis. Thus we do not provide a soundness proof for this extension of the type system. The types which can be inferred using the typing rules from this section roughly approximate sets of results of Xcerpt rules. To handle new Xcerpt constructs we introduce here two new type constants, namely *Num* and *Int*. They denote types being the sets of, respectively, all strings representing numbers and all strings representing integers. Thus the following relation holds: $[\![\mathit{Int}]\!] \subseteq [\![\mathit{Num}]\!] \subseteq [\![\mathit{Text}]\!]$.

We present typing rules for new constructs related to query terms, queries, and construct terms. The typing rules should be treated as a complement to the rules introduced in Section 4.2.2.

Query terms

$$\frac{D \vdash l\ \alpha q_1, \cdots, q_k, \cdots, q_n \beta : T \triangleright \Gamma}{D \vdash l\ \alpha q_1, \cdots, \texttt{optional}\ q_k, \cdots, q_n \beta : T \triangleright \Gamma} \quad (\textsc{Optional 1})$$

$$\frac{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma}{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, \texttt{optional}\ q_k, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma} \quad (\textsc{Optional 2})$$

$$\frac{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma}{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, \texttt{without}\ q_k, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma} \quad (\textsc{Without})$$

where $\alpha\beta = \{\{\}\}$ or $\alpha\beta = [[\,]]$.

$$\frac{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, q_k, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma}{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, \texttt{position n}\ q_k, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma} \quad (\textsc{Position})$$

where $T = \mathit{Top}$ or the parentheses of T are $[\,]$ and
n is a number.

$$\frac{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, q_k, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma}{D \vdash l\ \alpha q_1, \cdots, q_{k-1}, \texttt{position}\ X\ q_k, q_{k+1}, \cdots, q_n\beta : T \triangleright \Gamma \sqcap [X \mapsto \mathit{Int}]} \quad (\textsc{PositionVar})$$

where $T = \mathit{Top}$ or the parentheses of T are $[\,]$.

$$\frac{D \vdash l\ \alpha q_1, \cdots, q_n\beta : T \triangleright \Gamma}{D \vdash X\ \alpha q_1, \cdots, q_n\beta : T \triangleright \Gamma \sqcap [X \mapsto \mathit{Text}]} \quad (\textsc{LabelVar})$$

where l is an arbitrary label.

$$\frac{}{D \vdash r : T \triangleright \Gamma} \quad (\textsc{RegExpr})$$

where r is a regular expression and
either $T = \mathit{Top}$ or T is a type constant or an enumeration type name.

Queries

$$\frac{}{D \vdash \texttt{not}\, Q : U \triangleright \Gamma} \quad (\textsc{Not Query})$$

$$\frac{D \vdash Q : U \triangleright \Gamma}{D \vdash Q \ \texttt{where}\ con : U \triangleright \Gamma} \qquad (\textsc{WhereQuery})$$

Construct terms

$$\frac{D \vdash c_1 : \Psi \triangleright s_1 \quad \cdots \quad D \vdash c_k : \Psi \triangleright s_k \quad \cdots \quad D \vdash c_n : \Psi \triangleright s_n \quad (T_c \rightarrow l\alpha s_1 \cdots s_k? \cdots s_n\beta) \in D}{D \vdash l\alpha c_1, \ldots,\ \texttt{optional}\ c_k, \ldots, c_n\beta : \Psi \triangleright T_c} \qquad (\textsc{PatternOpt})$$

$$\frac{D \vdash c_1 : \Psi \triangleright s_1 \cdots D \vdash c_k : \Psi \triangleright s_k \cdots D \vdash c_n : \Psi \triangleright s_n \quad D \vdash c'_k : \Psi \triangleright s'_k \quad (T_c \rightarrow l\alpha s_1 \cdots (s_k|s'_k) \cdots s_n\beta) \in D}{D \vdash l\alpha c_1, \ldots,\ \texttt{optional}\ c_k\ \texttt{default}\ c'_k, \ldots, c_n\beta : \Psi \triangleright T_c} \qquad (\textsc{PatternOptDef})$$

$$\frac{D \vdash c_1 : \Psi \triangleright s_1 \quad D \vdash c_2 : \Psi \triangleright s_2}{D \vdash f(c_1, c_2) : \Psi \triangleright T_c} \qquad (\textsc{Funct})$$

where $T_c = Num$, s_1, s_2 are type names such that $[\![s_1]\!] \subseteq [\![Num]\!]$, $[\![s_2]\!] \subseteq [\![Num]\!]$ and f is one of the functions: `add`, `sub`, `mult`, `div` or
$T_c = Text$, s_1, s_2 are type names such that $[\![s_1]\!] \subseteq [\![Text]\!]$, $[\![s_2]\!] \subseteq [\![Text]\!]$ and f is the function `concat`.

$$\frac{D \vdash c_1 : \Psi \triangleright s_1 \quad \cdots \quad D \vdash c_n : \Psi \triangleright s_n}{D \vdash g(c_1, \ldots, c_n) : \Psi \triangleright T_c} \qquad (\textsc{Aggreg})$$

where g is an aggregation:
`count` and $T_c = Int$, or
`first` and $T_c = s_1$, or
`join`, $T_c = Text$ and for each type name T occurring in regular expressions $s_1, \ldots, s_n$, $[\![T]\!] \subseteq [\![Text]\!]$, or
either `sum`, `avg`, `min` or `max`, $T_c = Num$ and for each type name T occurring in regular expressions $s_1, \ldots, s_n$, $[\![T]\!] \subseteq [\![Num]\!]$.

4.3 Type-based Rule Dependency

In this section we discuss rule dependencies that describe data flow in Xcerpt programs. Determining dependencies between rules is needed, for example to divide programs into strata. Rule dependencies are also important from the point of view of an efficient evaluation of programs [35]. So far, in the thesis, we have used two kinds of rule dependency: static dependency and weak static dependency (Def. 12 and 13). Both notions of rule dependency only approximate the data flow in programs i.e. the fact that a rule (weakly) statically depends on some other rule does not mean that the rule will actually use the data produced by the other rule. That is why we present another notion of rule dependency, which precisely reflects the data flow between rules.

Definition 22 (Dynamic rule dependency)**.** *Let $\mathcal{P} = (P, G)$ be an Xcerpt program and $p \in P$ and $p' \in P \backslash G$ be query rules. The rule p* directly dynamically depends *on p' (which is denoted as $p \succ p'$), if a top query term from the body of p matches a result of p' in P.*

A rule $p \in P$ dynamically depends *on a rule $p' \in P \backslash G$ if $p \succ^+ p'$.*

It follows that $p \succ p'$ implies both $p \succ_s p'$ and $p \succ_w p'$.

Example 35. *Consider the rules p_3, p_4 from Example 8:*

$$\begin{aligned} p_3 &= b[\,Y\,] \leftarrow c[\,f[\,Y\,]\,], \\ p_4 &= c[\,Y\,] \leftarrow \mathtt{in}(\,r_1,\,Y\,). \end{aligned}$$

$p_3 \succ p_4$ iff the data term $\delta(r_1)$, specified by the URI r_1, is of the form $f[\,t\,]$ (for some data term t), while $p_3 \succ_s p_4$ independently from $\delta(r_1)$.

Dynamic rule dependency, which is essential for an efficient program evaluation, cannot be determined without knowing the actual data on which a program is evaluated. Thus it cannot be determined during a static analysis of the program. Static rule dependency can be used to approximate dynamic rule dependency of rules in a program. Here we show how to employ types and the presented type inference methods to approximate dynamic dependency. The goal is to obtain better approximations than those given by static dependency.

Let $\mathcal{P} = (P', G)$ be an Xcerpt program and $P = P' \setminus G$. By a **typing** of P we mean an approximation of the set of rule results of P by a set of type names. Formally, U is a typing for P if $[\![U]\!]$ contains each result of each rule from P. We presented two ways of obtaining such a typing: finding a fixed point of T_P or applying Theorem 3 (for recursive programs). Let U^P be a typing for P and $U_i^P = resType(p_i, U^P)$ for each $p_i \in P$.

The function *resType* is not useful for finding dynamic rule dependencies. This is because the fact $resType(p_i, U_j^P) \neq \emptyset$ does not imply $p_i \succ p_j$. The rule p_i may query external data so *resType* may return a non empty type independently of U_j^P (including cases when no data in U_j^P is matched by any top query term in p_i). The inverse implication, i.e. $p_i \succ p_j$ implies $resType(p_i, U_j^P) \neq \emptyset$, is neither true[2]. Thus we need some better way of

[2]Consider query rules: $p_1 = c[\,X\,] \leftarrow \mathtt{and}[\,a[\,X], \, b[\,X\,]\,],$
$p_2 = a[\,X\,] \leftarrow d[\,X\,].$
$resType(p_1, U_2^P) = \emptyset$ although $p_1 \succ p_2$ (unless p_2 produces no results).

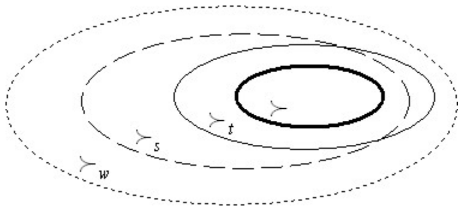


Figure 4.1: Relation between dependencies for rules in a program.

using types to approximate dynamic rule dependencies.

A part of the process of obtaining *resType*(p, U) is typing of query terms. Given a query term q and a type $T \in U$ of data to which q is applied, mappings of variables occurring in q to types are constructed. The mappings specify sets of substitutions (of data terms for variables). If q matches some data term d from $[\![T]\!]$ then a variable-type mapping Γ is produced; Γ describes a non empty set of substitutions. (The set contains the result, or some of the results, of matching q with d.)

The algorithm for *resType* can be easily augmented to compute a Boolean function *matchesType* whose arguments are a query rule p and a set of type names U; *matchesType*(p, U) is true iff a Γ describing a non empty set of substitutions is obtained for a top query term q from p and a type $T \in U$. Thus the following holds.

Proposition 5. *Let $\mathcal{P} = (P', G)$ be an Xcerpt program and $p, p' \in P'$. If $p \succ p'$ then matchesType(p, U) for any set U of type names such that the results of p' are contained in $[\![U]\!]$.*

Now we can define a new kind of rule dependency.

Definition 23 (Type-based dependency)**.** *Let $\mathcal{P} = (P', G)$ be an Xcerpt program, $P = P' \backslash G$, $p_i \in P$, U^P be a typing of P, and $U_i^P = resType(p_i, U^P)$. A rule $p \in P'$* type-based directly depends *on p_i (denoted by $p \succ_t p_i$) if matchesType(p, U_i^P).*

Proposition 6. *If $p \succ p'$ then $p \succ_t p'$. If $p \succ_t p'$ then $p \succ_w p'$.*

On the other hand, neither $p \succ_s p'$ implies $p \succ_t p'$ nor $p \succ_t p'$ implies $p \succ_s p'$. Both the type-based rule dependency and the static rule dependency approximate dynamic rule dependency in a program. Combining them provides better approximation than any of them separately. Figure 4.1 presents relation between different kinds of rule dependencies.

Example 36. *Consider a Type Definition $D = \{\, T \rightarrow l[\, T_1\, T_2\, T_1^* \,],$ $T_1 \rightarrow$ "e" $|$ "f", $T_2 \rightarrow$ "e" $\}$ and an Xcerpt program $\mathcal{P} = (\{g, p_1, p_2, p_3\}, \{g\})$ which queries an external resource* res *of type T. Next to the rules of the program there are specifications of the corresponding result types inferred for them:*

$$\begin{array}{llll}
g & = a[\,X\,] \leftarrow b[\text{"}e\text{"},\, \text{"}f\text{"},\, X\,] & & A \rightarrow a[\,T_2\,] \\
p_1 & = b[\,X,\, Y,\, Z\,] \leftarrow \mathtt{in}(\,\text{res},\, l[\,X,\, Y,\, Z\,]\,) & & A_1 \rightarrow b[\,T_1\, T_2\, T_1\,] \\
p_2 & = b[\,X,\, X,\, Y\,] \leftarrow \mathtt{in}(\,\text{res},\, l[\,X,\, Y\,]\,) & & A_2 \rightarrow b[\,T_1\, T_1\, T_2\,] \\
p_3 & = b[\,X,\, Z,\, Y\,] \leftarrow \mathtt{in}(\,\text{res},\, l[\,X,\, Y,\, Z\,]\,) & & A_3 \rightarrow b[\,T_1\, T_1\, T_2\,]
\end{array}$$

The rule g can dynamically depend only on the rule p_3 and only this dependency should be considered by optimal evaluation of the program. The rule g type-based depends only on the rules p_2, p_3 and statically depends on the rules p_1, p_3. (Hence $g \succ_w p_1$, $g \succ_w p_2$ and $g \succ_w p_3$.) Thus combination of type-based dependency and static dependency better approximates dynamic dependency than both dependencies separately. □

4.4 Discovering of Type Errors

We describe here how the presented type system can be used for discovering errors. Examples illustrating this topic can be found in Chapter 6.

Type Errors

We define correctness of Xcerpt programs wrt. a type specification of the input data and a type specification of program results. The type specification of the input data specifies the type of the data queried by a program. Thus it is given by the mapping *type* which assigns a type *type*(r) for each external resource r occurring in the program. The type specification of program results specifies the type of expected results of the program. This type is called *specified result type.* Formally the program is **correct** wrt. a type specification if the data produced by the program is of the specified result type provided that the data terms corresponding to the external resources queried by the program are of the types determined by the mapping *type* (i.e. provided that $\delta(r) \in [\![type(r)]\!]$ for each resource r occurring in the program). Correctness of a program wrt. a type specification is called *type correctness.* If the program is not type correct we say that there is a type error in it.

The type system presented in this chapter allows checking type correctness of programs. Given the type of the queried data the type system is able to infer a result type $[\![T_1 | \cdots | T_n]\!]$ of the program which is a superset of the set of results of the program. We refer to this type as the *inferred result type.* Then type correctness of the program can be proved by checking whether the inferred result type is included in the specified result type T_S of the program. A positive result of such typechecking (i.e. checking whether $[\![T_i]\!] \subseteq [\![T_S]\!]$ for $i = 1, \ldots, n$) implies type correctness of the program.

Generally, a typechecking failure is not a proof of type incorrectness of a program because the inferred result type is only an approximation (a superset) of the set of program results. However, for some restricted form of Xcerpt programs and Type Definitions (described in Section 4.2.4) 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. Otherwise a typechecking failure is a hint that the program results may not be of the specified result type.

In addition to defining type correctness of a program we define type correctness of a particular rule of the program. Type correctness of a rule p is defined wrt. a specification of types of the external resources queried by the rule and a specification describing the set $[\![U]\!]$ of allowed results of non goal rules of the program. (Thus U is the specified result type for non goal rules.) Moreover, if p is a goal rule, its type correctness is defined wrt. a specification describing the set $[\![T_S]\!]$ of allowed results of goal rules (i.e. the specified result type of the program). A query rule p is *type correct* if it satisfies one of the conditions:

- p is a non goal rule and when it is applied to a set of data terms $Z \subseteq [\![U]\!]$ it produces results belonging to the set $[\![U]\!]$ i.e. $res(p, Z) \subseteq [\![U]\!]$ for any $Z \subseteq [\![U]\!]$,
- p is a goal rule and when it is applied to a set of data terms $Z \subseteq [\![U]\!]$ it produces results belonging to the set $[\![T_S]\!]$ i.e. $res(p, Z) \subseteq [\![T_S]\!]$ for any $Z \subseteq [\![U]\!]$.

If a rule is not type correct we say that there is a type error in the rule. Type correctness of all rules in a program implies type correctness of the program. Hence finding the type incorrect rule(s) can be understood as locating the reason(s) why the program is type incorrect.

Type correctness of particular rules can be proved similarly as type correctness of programs, by checking whether the inferred result type of the rule is included in the specified result type i.e. by checking whether $[\![resType(p, U)]\!] \subseteq [\![U]\!]$, if p is a non goal rule; and whether $[\![resType(p, U)]\!] \subseteq [\![T_S]\!]$, if p is a goal rule.

Checking type correctness of rules and programs requires checking whether the inferred result type is included in the specified result type U or T_S. This checking can be done using the algorithm for Type Inclusion. The algorithm requires that the types T_S and U are specified by a proper Type Definition D. Sometimes it may be impossible to represent a specified result type by a single type name. Thus $[\![T_S]\!]$ and $[\![U]\!]$ may represent unions of types and T_S, U may be sets of type names. In order to be able to use the algorithm for Type Inclusion to check whether a type $[\![T]\!]$ is included in the union of types $[\![U_1]\!] \cup \ldots \cup [\![U_n]\!]$, where $T, U_1, \ldots, U_n$ are type names, $U_1, \ldots, U_n$ must satisfy the following condition. If U_i, U_j $(1 \leq i \leq n, 1 \leq j \leq n)$ are distinct type variables, they must have a different label or a different kind of parentheses. Then, if T is a type variable, checking whether $[\![T]\!] \subseteq [\![U_1]\!] \cup \ldots \cup [\![U_n]\!]$, can be reduced to checking whether $[\![T]\!] \subseteq [\![U_k]\!]$ $(1 \leq k \leq n)$, where U_k is *Top* or it is a type variable with the same label and the same kind of parentheses as T. If there is no such U_k among $U_1, \ldots, U_n$ the result of the inclusion check is negative. If T is not a type variable the inclusion check is obvious.

Weak Type Errors

There is another method of discovering type errors in programs. It can be checked whether the intersection of the inferred result type and the specified result type of a program is empty. Empty intersection of both types implies that the program does not produce any results of the specified result type. If the intersection is empty and the program produces any results then there is a type error in the program. In order to refer to such situation we introduce a notion of a *weak type error*. A weak type error does not imply a type error in the case when the set of possible program results is empty.

Emptiness Errors

The situation when the inferred result type of a program (or of a rule) is empty suggests that something is wrong in the program. Formally the program is type correct as the empty type is a subtype of any specified result type. However, as a program (or a rule) with an empty result type will not produce any results, this situation is usually not intended by the programmer and she should be informed about this. If the inferred result type of a program (or of a rule) is empty we say that there is an *emptiness error* in the program. Notice that for discovering emptiness errors there is no need for any specification of the result type. An algorithm for checking type emptiness was presented in Section 3.2.1.

Location of Errors

Based on the type analysis the programmer may suspect an existence of an error in a particular rule as e.g. the type system indicated (a possibility of) a type error for the rule. The available information about the rule (the type of the queried data and the specified result type) is not sufficient to automatically discover a reason of an error in the rule. It is not possible to define in a natural way what is the actual reason of a rule being type incorrect. For example, in general it cannot be automatically determined whether the error is due to a wrong body or a wrong head of the rule. Finding an actual reason of the error is a task of the programmer who can interpret the type information provided by the type system, such as the types inferred for particular variables in the rule. Thus type analysis can be used as a tool which facilitates understanding what the rule actually does.

Example 37. *Consider a non goal rule of some program*

$$c[\,X\,] \leftarrow \mathtt{in}(\,r,\ a[\,b[\,X\,]\,]\,)$$

and the following Type Definition $D = \{\, U{\to}c[\,B\,],\ A{\to}a[\,B\,],\ B{\to}b[\,Text\,]\,\}$. *Let us assume that the specified result type for the rule is* U *and that* $type(r) =$

A. The inferred result type for the rule obtained from the type system is a type C defined by the rule $C \rightarrow c[\,\mathit{Text}\,]$. As $[\![C]\!] \not\subseteq [\![U]\!]$ and as the inferred result type is the exact set of results of the rule (i.e. no approximations were performed), there is a type error in the rule. However we cannot say whether the type error is due to a wrong body or due to a wrong head of the rule. The error can be eliminated by correcting the body or by correcting the head of the rule. We can change the body of the rule obtaining

$$c[\,X\,] \leftarrow \mathtt{in}(\,r,\, a[\,X\,]\,).$$

The inferred result type for this rule is C' defined by the rule $C' \rightarrow c[\,B\,]$. As $[\![C']\!] \subseteq [\![U]\!]$ the rule is type correct. We can change only the head of the initial rule obtaining

$$c[\,b[X]\,] \leftarrow \mathtt{in}(\,r,\, a[\,b[\,X\,]\,]\,).$$

The inferred result type for this rule is also C' and the rule is type correct.

In practice it is likely that an error in one rule will generate emptiness errors for many rules in the program. In order to facilitate finding the actual reason of an emptiness error, it would be useful to locate the rules for which the emptiness error is a direct consequence of the actual error, and not a consequence of an empty result type inferred for other rules. Finding such rules is easy for non recursive programs. In this case the rule dependency tree for the program can be constructed. Then the programmer is informed about those rules with an emptiness error that do not depend on other rules with an emptiness error.

4.5 Relation to XQuery Type System

Section 2.3 shortly introduced XQuery and its type system. Here we discuss some differences and similarities between our type system and XQuery type system.

- Similarly as XQuery, Xcerpt has its own data model (i.e. data terms) for representing XML data. However XQuery data model is more complex and it takes into account more features of XML documents. In XQuery, XML documents are represented as ordered trees with nodes of different kinds such as attribute, comment, processing instruction, etc. In contrast, data terms are mixed trees where children of a node can be ordered or unordered. Moreover all nodes of such trees are of the same kind and such features of an XML document as comments and processing instructions are neglected. In contrast to data terms, in the XQuery data model, each node has a unique identity. Hence a node and its copy are not considered identical.

  Another difference is that element and attribute nodes from XQuery data model are associated with types. The type annotations are either *xs:untyped* (or *xs:untypedAtomic*) or are determined in a schema validation process. Thus, when XML data is processed by XQuery, each XML element is of a unique type. In contrast, in our approach any data term can belong to an unlimited number of types. Even given a fixed Type Definition, a data term can belong to a number of types.

While our types are sets of data terms, XQuery types can be seen as expressions. A notion of a type as a set in XQuery is derived – relation *matches* defines the set of values that match a given type [25, Section 2.5.4]. Notice that elements of such a set are sequences of trees (not single trees as in the types defined by Data Definitions). In both approaches the notions of subtyping are similar; both are defined by means of set inclusion.

- XQuery is a strongly typed language and its type system is its integral part. The results of some operations of XQuery depend on the type annotation of data. On the other hand, the Xcerpt semantics does not deal with types at all. Our type system is added on the top of the Xcerpt language. Hence our type system can be classified as descriptive, and that of XQuery as prescriptive.

- XQuery defines correctness of programs wrt. types by a number of conditions that must be satisfied. For example, a value of a function argument has to be in the relation *matches* with the required type of the argument. Similar conditions are stated for function result types, for types of variables etc. In our type system the only condition for type correctness of a program (or a rule) is that all its results belong to the specified result type.

- XQuery provides both dynamic and static typing and static typing is optional. In the case of dynamic typing computed values of expressions are checked wrt. types when a program is being executed. We deal only with static typing where type analysis is performed before program execution.

- In XQuery, the result of static typing is an abstract tree which assigns a type to each subexpression. In our Xcerpt type system, we assign types only to variables in expressions and for the whole Xcerpt query rules. In XQuery, assigning the empty type to an expression results in a type error. In our system, assigning the empty type to a rule (a variable) does not imply that the rule is incorrect. Such a fact is instead reported as a specific kind of error, namely emptiness error.

- Similarly as in XQuery we use an internal formalism to represent types, which originally can be specified in various XML schema languages. However, the formalism for defining types of XQuery is more complex and provides more flexibility. XQuery allows to define elements with specified content and unspecified name. It handles XML Schema simple types and includes mechanisms dealing with them. Our type system does not deal with the simple types of XML Schema, but we expect that it can be easily extended to do so. XQuery type formalism resigns from XML Schema restrictions such as *Element Declarations Consistent* and *Unique Particle Attribution*. Based on our experience with Type Definitions we can argue that relaxing such restrictions results in inefficient algorithms (non polynomial) for performing operations on types such as checking type inclusion. Checking type inclusion cannot be done efficiently, as this task for a less general formalism of tree automata is EXPTIME-complete [24]. There is no discussion related to typing algorithms, and to complexity of operations on types in [25].

- In both type systems type analysis can be performed without any type specification of the queried data. Also in this case type analysis can facilitate discovering errors. In our type system the queried data is assumed to be of type *Top* if no more specific type is given. In XQuery non validated data is of a type `xs:untyped` or `xs:untypedAtomic`.

Chapter 5

Type System 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. The prototype 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 for which the formal semantics is provided (in Section 2.1.2). Moreover, it is restricted to non recursive Xcerpt programs. 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`.

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 invoked 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 invoked 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 <*program file*> is an Xcerpt program and <*type specification*> is a text file specifying the types of external resources (i.e. the resources referred to in targeted query terms in(r, q) in the program) and the types of expected results. <*type specification*> file may contain:

- a Type Definition i.e. rules defining types,
- input type specifications; each such specification specifies a type *type*(r) of a queried resource r,
- output type specifications specifying result types for particular rules.

The input type specification has the syntax:

```
Input::
  [ resource = <resource URI> ]
  [ typedef  = <typedef location> ]
  typename = <type name>
```

and the output type specification has the syntax:

```
Output::
  [ rule    = <index>]
  [ typedef  = <typedef location>]
  typename = <type name>
```

where

- <*resource URI*> is an URI of the resource being queried whose type we specify. If the parameter `resource` is omitted the input type specification specifies a type of every resource occurring in the <*program file*> whose type was not specified (overridden) by other input type specification. A <*type specification*> can contain at most one input type specification without the parameter `resource`.
- <*typedef location*> is a URI of an external file containing a Type Definition (or DTD). If the the parameter `typedef` is omitted the input or output type specification refers to the local Type Definition i.e. specified in the current <*type specification*> file.
- <*type name*>, if used in an input type specification, it 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 rule the specification refers to. It can be the most general type `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.

- <*index*> is a number of the query rule in the Xcerpt program whose output type we specify. It can be obtained by counting the query rules in the program starting from one e.g. the index of the second query rule in a program is 2. If the parameter `rule` is omitted the output type specification concerns the first goal in the Xcerpt program (or the first query rule if the program contains no goals). A <*type specification*> can contain at most one output type specification without the parameter `rule`.

Example 38. *This is an example of a <type specification> file* books.xts*:*

```
Publications -> publications[ Book* Article* ]
Article -> article[ Title Author+ Proceedings ]
Proceedings -> proceedings[ Title Editor+ ]
Book -> book[ Title Author+ Editor+ ]
Title -> title[Text ]
Author -> author[ P ]
Editor -> editor[ P' ]
P -> person[ S ]
P' -> person[ F? S? ]
Person -> person[ F+ S ]
F -> firstname[ Text ]
S -> surname[ Text ]
AuthorsEditors -> authors-editors[ Person+ ]

Input::
  resource = file:publications.xml
  typename = Publications

Output::
  rule = 1
  typename = AuthorsEditors
```

Invoking the typing mechanism (e.g. with the command `xcerpt -t` <*program file*> <*type specification*>) starts the process of type inference for the program. 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 *Top* (which can be seen as a default type of a resource). After the type of results for each query rule of the program has been inferred, type checking is performed for each query rule for which output type specification is provided and for which the inferred result type is not empty. Type checking for a rule includes an inclusion check: it is checked whether the inferred result type is included in the corresponding output type (specified by the output type specification). If the check fails an intersection emptiness check is performed: it is checked whether the intersection of the inferred result type and the specified output type is empty. Thus there are three possible results of type checking for a rule:

- *OK* - the inferred result type is included in the specified type i.e. the rule is correct wrt. the specified type,
- *Failed* - the intersection of the inferred result type and the specified type is empty i.e. there is a weak type error for the rule,
- *Unsuccessful* - the inferred result type is not included in the specified type but the intersection of both types is not empty i.e. the rule may be incorrect wrt. the specified type.

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 that for each query rule of the program, contains

- the inferred result type for the rule (0 stands for empty result type), for example, `Rule 2: Person`,
- if there is a result type specified for the rule and if the inferred result type is not empty, the result of type checking, for example, `Type checking: Failed`,
- variable-type mappings for variables occurring in the rule ($* \rightarrow Top$ stands for the mapping $\top$, cf. Section 4.2.1)

Moreover the printout contains a Type Definition defining all the inferred types and the types of the queried resources.

For the types being intersection of other types their content model is provided by a DFA instead of a regular type expression[2]. A DFA is presented by descriptions of all its states. Each such a description is of the form $\mathtt{S_i} => \mathtt{a_1} > \mathtt{S_{k_{i1}}} \ldots \mathtt{a_n} > \mathtt{S_{k_{in}}}$, 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 $S_{k_{ij}}$ 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  => A>0 F>0
>1  => A>2 F>0
 2! => 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^T_2. The type checker also invents type names for the newly

[2]For any regular expression a DFA representing the same language may be constructed [38].

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 and 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 39. *Here we present an output of the type system prototype for the following Xcerpt program:*

```
GOAL
  authors-editors[ all var X ]
FROM
  books[[
    book{{
      title[ var Y ],
      author[ var X ],
      editor[ var X ]
    }}
  ]]
END

CONSTRUCT
  books[ all var X ]
FROM
  in{ resource{ "file:publications.xml" },
    desc var X -> book{{ }}
  }
END
```

A type specification for the program is given by publications.xts *file from the previous example. The obtained output is:*

```
==========================================================
Rule 1: authors-editors
Type checking: Failed (no results of type AuthorsEditors)
----------------------------------------------------------
Y->Text, X->P^P'
==========================================================
Rule 2: books
----------------------------------------------------------
X->Book
==========================================================
==========================================================
Type Definition:
----------------------------------------------------------
authors-editors -> authors-editors[ P^P'+ ]
books -> books[ Book+ ]
Publications -> publications[ Book* Article* ]
Article -> article[ Title Author+ Proceedings ]
```

```
Proceedings -> proceedings[ Title Editor+ ]
Book -> book[ Title Author+ Editor+ ]
Title -> title[ Text ]
Author -> author[ P ]
Editor -> editor[ P' ]
P -> person[ S ]
P' -> person[ F? S? ]
Person -> person[ F+ S ]
F -> firstname[ Text ]
S -> surname[ Text ]
AuthorsEditors -> authors-editors[ Person+ ]
P^P' -> person[
 0  => S>0
>1  => S>2
 2! => S>0
 ]
=========================================================
```

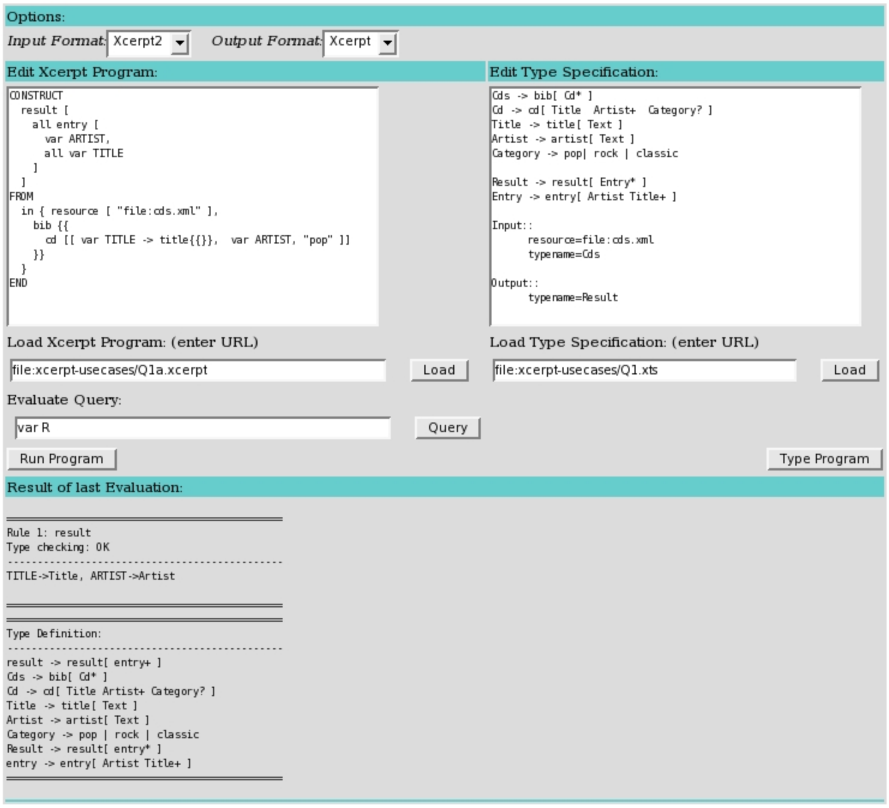
The printout contains the inferred result types for the first and the second rule, which are respectively, authors-editors *and* books*. It also contains information of the inferred types for the particular variables occurring in the rules. All the types are defined by the Type Definition from the bottom of the printout. As an output type specification is provided for the first rule the printout contains the result of type checking for the rule.*

Figure 5.1 presents a screenshot of the online type system interface.

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 5.2. Most of the modules shown there are discussed in the description of Xcerpt prototype in [50]. 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
  - `TypeInter.hs` containing an implementation of the Type Intersection algorithm.
- `Xcerpt.RegExp` implements regular expressions and automata using a Haskell library[53]. The library has been modified to support lists


Options:

Input Format: Xcerpt2 Output Format: Xcerpt

Edit Xcerpt Program:

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

Edit Type Specification:

```
Cds -> bib[ Cd* ]
Cd -> cd[ Title  Artist+  Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> pop| rock | classic

Result -> result[ Entry* ]
Entry -> entry[ Artist Title+ ]

Input::
      resource=file:cds.xml
      typename=Cds

Output::
      typename=Result
```

Load Xcerpt Program: (enter URL)

file:xcerpt-usecases/Q1a.xcerpt

Load Type Specification: (enter URL)

file:xcerpt-usecases/Q1.xts

Evaluate Query:

var R

Result of last Evaluation:

```
=============================================
Rule 1: result
Type checking: OK
---------------------------------------------
TITLE->Title, ARTIST->Artist

=============================================
=============================================
Type Definition:
---------------------------------------------
result -> result[ entry+ ]
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> pop | rock | classic
Result -> result[ entry* ]
entry -> entry[ Artist Title+ ]
=============================================
```

Figure 5.1: Type system prototype interface.

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
  - `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.

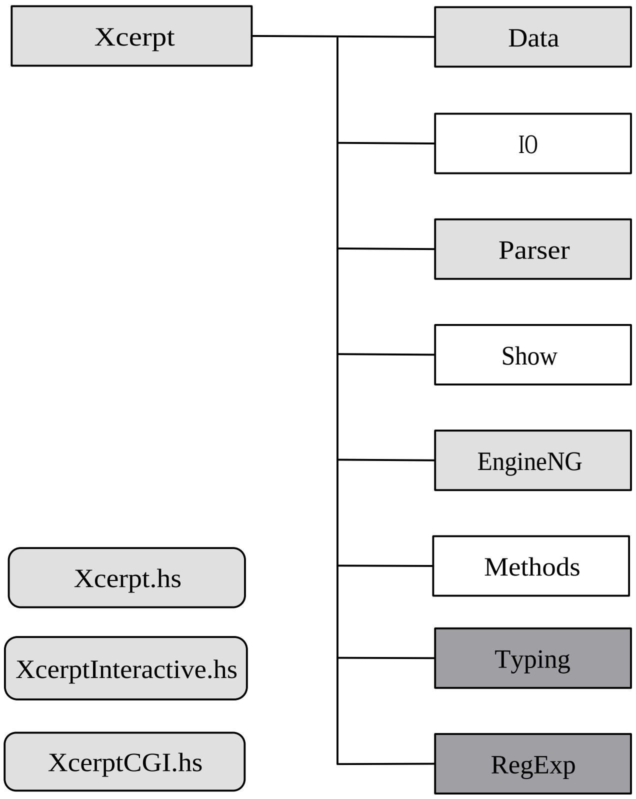


Figure 5.2: Overall module and file structure of the Xcerpt runtime system together with the type system; modules denoted by rectangles, files by rounded rectangles, added modules related to the type system in grey, modified modules or files in light grey.

- `Xcerpt/EngineNG` has been extended with the file `Typing.hs` containing the main functions controlling the type system.
- `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 for checking correctness of Xcerpt programs. The examples show the way the type system can facilitate finding errors in programs. The programs presented in this chapter, except the last one (as the prototype is not operational for recursive programs), have been type checked by our prototype and the corresponding printouts are presented in Appendix B.

6.1 CD Store

Consider a Type Definition:

$$
\begin{array}{lll}
\mathit{Cds} & \rightarrow & \mathit{bib}\,[\,\mathit{Cd}^{*}\,] \\
\mathit{Cd} & \rightarrow & \mathit{cd}\,[\,\mathit{Title}\ \mathit{Artist}^{+}\ \mathit{Category}^{?}\,] \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathit{Text}\,] \\
\mathit{Artist} & \rightarrow & \mathit{artist}\,[\,\mathit{Text}\,] \\
\mathit{Category} & \rightarrow & \text{"}\mathit{pop}\text{"} \mid \text{"}\mathit{rock}\text{"} \mid \text{"}\mathit{classic}\text{"}
\end{array}
$$

The query rule below queries a document *cds.xml* of the type *Cds* defined above. The intention of the query rule is to collect artists together with all the titles of the CD's of the category "pop".

```
CONSTRUCT
  pop-entries[
    all entry[
      var ARTIST,
      all var TITLE
    ]
  ]
FROM
  in{ resource[ "file:cds.xml" ],
    bib{{
```

```
                cd[[ var TITLE,  var ARTIST, "pop" ]]
              }}
            }
          END
```

First, we assume that no result type specification is given for the rule.

Printout CDSTORE.1 in the appendix is the result of typing the query rule by the typechecker. We assume that the intention of an author of the query rule is that the variable TITLE will be bound to data terms *title*[...] and the variable ARTIST will be bound to data terms *artist*[...]. The type system infers the types of variables used in the query rule. They are given by the variable-type mappings: $[TITLE \mapsto Title,\ ARTIST \mapsto Artist]$, $[TITLE \mapsto Artist,\ ARTIST \mapsto Artist]$. As the variable TITLE is intended (by the programmer) to take values only of the type *Title*, the inferred types for variables suggest that the query rule is incorrect with respect to the programmer's expectations.

Based on the inferred types of variables the query rule result type is inferred. The inferred result type is *pop-entries* defined as

$$\begin{array}{lcl} \textit{pop-entries} & \rightarrow & \textit{pop-entries}\,[\, \textit{entry}^{+} \,] \\ \textit{entry} & \rightarrow & \textit{entry}\,[\, \textit{Artist}\ (\textit{Title} \mid \textit{Artist})^{+} \,] \\ \textit{Artist} & \rightarrow & \textit{artist}\,[\, \textit{Text} \,] \\ \textit{Title} & \rightarrow & \textit{title}\,[\, \textit{Text} \,] \end{array}$$

Looking at this definition of the inferred result type of the rule, the programmer can also realize that the results of the rule may be different from expected ones (as an entry should not contain more than one artist).

Now, let us assume that a result type specification is provided for the rule and the specified result type is *Entries* as defined below:

$$\begin{array}{lcl} \textit{Entries} & \rightarrow & \textit{pop-entries}\,[\, \textit{Entry}^{*} \,] \\ \textit{Entry} & \rightarrow & \textit{entry}\,[\, \textit{Artist}\ \textit{Title}^{+} \,] \\ \textit{Artist} & \rightarrow & \textit{artist}\,[\, \textit{Text} \,] \\ \textit{Title} & \rightarrow & \textit{title}\,[\, \textit{Text} \,] \end{array}$$

Printout CDSTORE.2 in the appendix corresponds to this case. Now the system can automatically check that the inferred result type *pop-entries* is not included in the type *Entries* (as the type *entry* is not a subtype of the type *Entry*). This information suggests 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 *pop-entries* is not exact[1] (as the query rule uses the construct `all`). There is no weak type error for the rule as the intersection

[1] The inferred result type *pop-entries* is a superset of the actual set of possible results which is $[\![Entries']\!]$ defined as

$$\begin{array}{l} \textit{Entries}' \rightarrow \textit{pop-entries}[\textit{Entry}'^{+}] \\ \textit{Entry}' \rightarrow \textit{entry}[\textit{Artist}\ (\textit{Title} \mid \textit{Artist})^{*}\ \textit{Title}\ (\textit{Title} \mid \textit{Artist})^{*}] \end{array}$$

of the types *Entries* and *pop-entries* is not empty. Nevertheless, there is a type error for the rule. The 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, for example:

```
pop-entries[
  entry[ artist[ "artist1" ], title[ "title1" ] ],
  entry[ artist[ "artist2" ], title[ "title1" ], artist[ "artist1" ] ]
]
```

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

```
bib[
  cd[
    title[ "title1" ],
    artist[ "artist1" ],
    artist[ "artist2" ],
    "pop"
  ]
]
```

6.2 Bibliography

Consider the following Type Definition:

$$\begin{array}{lcl}
\mathit{Bibliography} & \rightarrow & \mathit{bib}\,[\,(\mathit{Book} \mid \mathit{Article} \mid \mathit{InProceedings})^{*}\,] \\
\mathit{Book} & \rightarrow & \mathit{book}\,\{\,\mathit{Title}\ \mathit{Authors}\ \mathit{Editors}\ \mathit{Publisher}^{?}\,\} \\
\mathit{Article} & \rightarrow & \mathit{article}\,\{\,\mathit{Title}\ \mathit{Authors}\ \mathit{Journal}^{?}\,\} \\
\mathit{InProceedings} & \rightarrow & \mathit{inproc}\,\{\,\mathit{Title}\ \mathit{Authors}\ \mathit{Book}\,\} \\
\mathit{Title} & \rightarrow & \mathit{title}\,[\,\mathrm{Text}\,] \\
\mathit{Authors} & \rightarrow & \mathit{authors}\,[\,\mathit{Person}^{*}\,] \\
\mathit{Editors} & \rightarrow & \mathit{editors}\,[\,\mathit{Person}^{*}\,] \\
\mathit{Publisher} & \rightarrow & \mathit{publisher}\,[\,\mathrm{Text}\,] \\
\mathit{Journal} & \rightarrow & \mathit{journal}\,\{\,\mathit{Title}\ \mathit{Editors}\,\} \\
\mathit{Person} & \rightarrow & \mathit{person}\,[\,\mathit{FirstName}\ \mathit{LastName}\,] \\
\mathit{FirstName} & \rightarrow & \mathit{first}\,[\,\mathrm{Text}\,] \\
\mathit{LastName} & \rightarrow & \mathit{last}\,[\,\mathrm{Text}\,]
\end{array}$$

6.2.1 No Result Type Specified

The query rules from this section query a document *bibliography.xml* of the type *Bibliography* defined above.

```
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 corresponding printout in the appendix is BIBLIOGRAPHY.1. The query rule returns no results when it is applied to a document of type *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. We get an emptiness error for the rule.

This is another example of a query rule with an emptiness error.

```
   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 corresponding printout in the appendix is also BIBLIOGRAPHY.1. 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 document because of the square brackets used to match data terms *book*{...}. According to the type of the document direct subterms of *book*{...} are unordered and cannot be matched with a query term being an ordered pattern.

```
      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 corresponding Printout in the appendix is also Bibliography.1.

An emptiness error is obtained also for the next query rule. 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.

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

The corresponding printout in the appendix is Bibliography.1.

6.2.2 Result Type Specified

The next example of a query rule illustrates a transformation of an XML document to a format similar to HTML. The format is defined by the following Type Definition which specifies the result type *TextBook* [2]:

$$\begin{array}{lll}
\textit{TextBook} & \rightarrow & \textit{book}\,[\,\textit{Cover Body}\,] \\
\textit{Cover} & \rightarrow & \textit{cover}\,[\,\textit{Title Author}^{*}\ \textit{Publisher}^{?}\,] \\
\textit{Body} & \rightarrow & \textit{body}\,[\,\textit{Abstract}^{?}\ \textit{Chapter}^{*}\,] \\
\textit{Title} & \rightarrow & \textit{title}\,[\,\textit{Text}\,] \\
\textit{Author} & \rightarrow & \textit{author}\,[\,\textit{Text}\,] \\
\textit{Publisher} & \rightarrow & \textit{publisher}\,[\,\textit{Text}\,] \\
\textit{Abstract} & \rightarrow & \textit{abstract}\,[\,\textit{Text}\,]
\end{array}$$

[2] The Type Definition and the two following examples of query rules were devised by Sacha Berger.

$\mathit{Chapter}$	$\rightarrow$	$\mathit{chapter}\,[\,\mathit{Title}\ \mathit{Section}^*\,]$
$\mathit{InlineContent}$	$\rightarrow$	$\mathit{inline}\,[\,\mathit{Text} \mid \mathit{Bf} \mid \mathit{Em}\,]$
$\mathit{Section}$	$\rightarrow$	$\mathit{section}\,[\,\mathit{Title}^?\ (\mathit{Paragraph} \mid \mathit{Table} \mid \mathit{List})^*\,]$
Em	$\rightarrow$	$\mathit{em}\,[\,\mathit{InlineContent}\,]$
Bf	$\rightarrow$	$\mathit{bf}\,[\,\mathit{InlineContent}\,]$
$\mathit{Paragraph}$	$\rightarrow$	$\mathit{p}\,[\,\mathit{InlineContent}^*\,]$
Table	$\rightarrow$	$\mathit{table}\,[\,\mathit{TableRow}^+\,]$
List	$\rightarrow$	$\mathit{list}\,[\,\mathit{ListItem}\,]$
$\mathit{TableRow}$	$\rightarrow$	$\mathit{tr}\,[\,\mathit{TableCell}^*\,]$
$\mathit{ListItem}$	$\rightarrow$	$\mathit{item}\,[\,\mathit{InlineContent}^*\,]$
$\mathit{TableCell}$	$\rightarrow$	$\mathit{td}\,[\mathit{InlineContent}^*\,]$

Consider a query rule which queries a document *bibliography.xml*:

```
CONSTRUCT
  book[
    cover[ title[ "List_of_Books" ] ],
    body[
      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
```

Let us assume that no type specification for the document *bibliography.xml* is provided. In such case the system infers a very rough approximation of the set of results of the rule (printout BIBLIOGRAPHY.2) and the inferred result type *book* is not included in the specified result type *TextBook*. Thus a type error is possible. To make sure about that it is checked whether the intersection of the inferred result type and the specified result type is empty. Indeed the intersection is empty. A weak type error, which we get, implies that the rule will not return any results of the type *TextBook*. Notice that the weak type error has been obtained without any type specification for the queried data.

The weak type error 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 a data term *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.

Consider another query rule which queries the document *bibliography.xml*. This time we assume that both type specifications are given i.e. type specification for the document *bibliography.xml* (the type *Bibliography*) and a result type specification for the rule (the type *TextBook*).

```
CONSTRUCT
  book[
    cover[ title [ "Books" ] ],
    body[
      chapter[
        title[ "List_of_Books_and_Authors"],
        section[
          table[
            all tr[
              td[ inline [ var TITLE ] ],
              td[ inline [ var NAME ] ]
            ]
          ]
        ]
      ]
    ]
  ]
FROM
  in{ resource[ "file:bibliography.xml" ],
    bib{{
      book{{
        title[ var TITLE ],
        desc var NAME
      }}
    }}
  }
END
```

A type error is possible for the query rule as the inferred result type *book* is not included in the type *TextBook* (printout BIBLIOGRAPHY.3). We are not sure about type incorrectness of the rule as the inferred result type is not exact (due to a construct `all`). This time there is no weak type error discovered for the rule, which means that the rule may produce results of the specified result type. Thus the structure of the head of the rule conforms to the specified result type. The type inclusion check failure is due to the variables which get wrong values i.e. not of the types required by the result type specification. The variable NAME used in the body of the query rule can be bound to any data term which is a direct or an indirect subterm of *book*[...] (except a data term *title*[...]). Thus, the variable NAME may be mapped to the types: *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 *inline*[...] which are *Text*, *Bf* and *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 *Text*, *Bf* and *Em*.

The next rule is almost the same as the previous one. The difference is in the usage of the variable NAME in the body of the rule, which now can be bound only to the direct subterm of an element *last*[...].

```
CONSTRUCT
  book[
    cover[ title [ "Books" ] ],
    body[
      chapter[
        title[ "List_of_Books_and_Authors"],
        section[
          table[
            all tr[
              td[ inline [ var TITLE ] ],
              td[ inline [ var NAME ] ]
            ]
          ]
        ]
      ]
    ]
  ]
FROM
  in{ resource[ "file:bibliography.xml" ],
    bib{{
      book{{
        title[ var TITLE ],
        desc last[ var NAME ]
      }}
    }}
  }
END
```

The result of type checking performed by the type system for the rule is positive. Thus the rule is correct wrt. the specified result type *TextBook*. The corresponding typechecker printout is BIBLIOGRAPHY.4.

6.3 Bookstore

Here we present an example of an Xcerpt program being one of the use cases for Xcerpt presented in [50]. As no result type specification is given for the program the type system is only able to perform type inference and check emptiness of the inferred types. This results in a specification of the inferred result type for the program. 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 use case is similar to one from *XQuery Use Cases* (*XMP-Q5* in [21]). 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[3]). 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 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 [50] using the Relax NG notation and can be expressed by the following Type Definition. The definition of type *Bib* is expressed also by the DTD from Example 10 (Section 2.2.1).

$$
\begin{array}{lll}
Bib & \rightarrow & bib\,[\,Book^*\,] \\
Book & \rightarrow & book\,[\,Book\_attr\ \ Title\ \ (Authors \mid Editor)\ \ Publisher\ \ Price\,] \\
Book\_attr & \rightarrow & attr\,\{\,Book\_year\,\} \\
Book\_year & \rightarrow & year\,[\,Text\,] \\
Title & \rightarrow & title\,[\,Text\,] \\
Authors & \rightarrow & authors\,[\,Author^*\,] \\
Author & \rightarrow & author\,[\,Last\ \ First\,] \\
Editor & \rightarrow & editor\,[\,Last\ \ First\ \ Affil\,] \\
Last & \rightarrow & last\,[\,Text\,] \\
First & \rightarrow & first\,[\,Text\,] \\
Affil & \rightarrow & affiliation\,[\,Text\,] \\
Publisher & \rightarrow & publisher\,[\,Text\,] \\
Price & \rightarrow & price\,[\,Text\,] \\
Reviews & \rightarrow & reviews\,[\,Entry^*\,]
\end{array}
$$

[3]`http://www.wapforum.org/DTD/wml_1.1.xml`

$$\begin{array}{lcl} \textit{Entry} & \rightarrow & \textit{entry}\,[\,\textit{Title Price Review}\,] \\ \textit{Review} & \rightarrow & \textit{review}\,[\,\textit{Text}\,] \end{array}$$

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{
   resource[ "file:prices.wml" , "xml" ],
   wml[
     all card[
       "Title: " , var Title ,
       "Price A: " , var PriceA,
       "Price B: " , var PriceB
     ]
   ]
 }
FROM
 books-with-prices[[
   book-with-prices[[
     title[[ var Title ]],
     price-a[[ var PriceA ]],
     price-b[[ var PriceB ]]
```

```
    ]]
  ]]
END

CONSTRUCT
 books-with-prices[
   all book-with-prices[
     title[ var T ],
     price-a[ var Pa ],
     price-b[ var Pb ]
   ]
 ]
FROM
 and{
   in{ resource [ "file:bib.xml" ],
     bib[[
       book[[
         title[ var T ],
         price[ var Pa ]
       ]]
     ]]
   },
   in{
     resource[ "file:reviews.xml" ],
     reviews[[
       entry[[
         title[ var T ],
         price[ var Pb ]
       ]]
     ]]
   }
 }
END
```

The type system infers results types of the rules. The inferred result type for the third query rule is *books-with-prices*. The inferred result types for the first and the second goal are respectively *html* and *wml*. These types are defined by the following Type Definition:

$$\begin{array}{lll}
\textit{books-with-prices} & \rightarrow & \textit{books-with-prices}\,[\,\textit{book-with-prices}^{+}\,] \\
\textit{book-with-prices} & \rightarrow & \textit{book-with-prices}\,[\,\textit{title price-a price-b}\,] \\
\textit{price-a} & \rightarrow & \textit{price-a}\,[\,\text{Text}\,] \\
\textit{price-b} & \rightarrow & \textit{price-b}\,[\,\text{Text}\,] \\
\textit{title} & \rightarrow & \textit{title}\,[\,\text{Text}\,] \\
 & & \\
\textit{html} & \rightarrow & \textit{html}\,[\,\textit{head body}\,] \\
\textit{head} & \rightarrow & \textit{head}\,[\,\textit{title}_1\,] \\
\textit{title}_1 & \rightarrow & \textit{title}\,[\,\textit{Text}_1\,] \\
\textit{Text}_1 & \rightarrow & \text{"}\textit{Price Overview}\text{"}
\end{array}$$

$$\begin{array}{lll}
\mathit{body} & \rightarrow & \mathit{body}\,[\,\mathit{table}\,] \\
\mathit{table} & \rightarrow & \mathit{table}\,[\,\mathit{tr}\ \mathit{tr}_1^+\,] \\
\mathit{tr} & \rightarrow & \mathit{tr}\,[\,\mathit{td}\ \mathit{td}_1\ \mathit{td}_2\,] \\
\mathit{td} & \rightarrow & \mathit{td}\,[\,\mathit{Text}_2\,] \\
\mathit{Text}_2 & \rightarrow & \text{"}\mathit{Title}\text{"} \\
\mathit{td}_1 & \rightarrow & \mathit{td}\,[\,\mathit{Text}_3\,] \\
\mathit{Text}_3 & \rightarrow & \text{"}\mathit{Price\ at\ A}\text{"} \\
\mathit{td}_2 & \rightarrow & \mathit{td}\,[\,\mathit{Text}_4\,] \\
\mathit{Text}_4 & \rightarrow & \text{"}\mathit{Price\ at\ B}\text{"} \\
\mathit{tr}_1 & \rightarrow & \mathit{tr}\,[\,\mathit{td}_3\ \mathit{td}_3\ \mathit{td}_3\,] \\
\mathit{td}_3 & \rightarrow & \mathit{td}\,[\,\mathrm{Text}\,] \\
\\
\mathit{wml} & \rightarrow & \mathit{wml}\,[\,\mathit{card}^+\,] \\
\mathit{card} & \rightarrow & \mathit{card}\ [\,\mathit{Text}_5\ \mathrm{Text}\ \mathit{Text}_6\ \mathrm{Text}\ \mathit{Text}_7\ \mathrm{Text}\,] \\
\mathit{Text}_5 & \rightarrow & \text{"}\mathit{Title}:\ \text{"} \\
\mathit{Text}_6 & \rightarrow & \text{"}\mathit{PriceA}:\ \text{"} \\
\mathit{Text}_7 & \rightarrow & \text{"}\mathit{PriceB}:\ \text{"}
\end{array}$$

The corresponding printout in the appendix is BOOKSTORE. Since no result type specification is given the type system checks only if the inferred result type for each rule is not empty. If a result type specification was given, the type system could check whether the data produced by the program conforms to HTML and WML formats.

6.4 Clique of Friends

Consider the program from Examples 6,26 consisting of three query rules, p_1, p_2, g, respectively:

```
CONSTRUCT
 fo[ var X, var Y ]
FROM
 in[ "file:addrBooks.xml",
   addr-books{{
     addr-book{{
       owner[ var X ],
       entry{{ name[ var Y ], relation[ "friend" ] }}
     }}
   }}
 ]
END

CONSTRUCT
 foaf[ var X, var Y ]
FROM
 or[
   fo[ var X,  var Y ],
   and[ fo[ var X, var Z ], foaf[ var Z, var Y ] ]
 ]
END

GOAL
 clique-of-friends[ all foaf[ var X, var Y ] ]
FROM
 foaf[ var X, var Y ]
END
```

The program queries the document *addrBooks.xml* of type *AddrBs* defined by the following Type Definition:

AddrBs	$\rightarrow$ *addr-books*[*AddrB**]
AddrB	$\rightarrow$ *addr-book*[*Owner Entry**]
Owner	$\rightarrow$ *owner*[*Name*]
Entry	$\rightarrow$ *entry*[*Name Rel PhNo** *Address*$^?$]
Name	$\rightarrow$ *name*[Text]
Rel	$\rightarrow$ *relation*[*RelCat*]
Address	$\rightarrow$ *address*[*Street ZipC*$^?$ *City Country*$^?$]
PhNo	$\rightarrow$ *phoneNo*[Text]
Street	$\rightarrow$ *street*[Text]
ZipC	$\rightarrow$ *zip-code*[Text]
Country	$\rightarrow$ *country*[Text]
City	$\rightarrow$ *city*[Text]
RelCat	$\rightarrow$ "*friend*" | "*family*" | "*colleague*" | "*acquaintance*"

As the program is recursive it cannot be checked by the current version of our type system prototype. However the presented algorithm for type inference (Section 4.2.3) allows to derive types for the rules of the program. As the second rule w-depends on itself the recursion in the program can be broken by approximating its result type. The approximation can be provided by the programmer. Let us assume that the programmer expects that all results of the second rule are of type *Foaf* defined as *Foaf* $\rightarrow$ *foaf*[Text Text]. Thus *Foaf* is a specified result type of p_2. Using the algorithm from Section 4.2.3, a fixed point of the operator $\widehat{T}_{\{p_1,p_2\}}$ can be computed. The fixed point is a set of type names $U^\infty = T_{\{p_1\}}(\emptyset) \cup \{Foaf\} = resType(p_1, \emptyset) \cup \{Foaf\} = \{Foaf, Fo\}$, where type *Fo* is defined as *Fo* $\rightarrow$ *fo*[*Name* Text].

According to Theorem 3, $[\![U^\infty]\!]$ includes all the results of p_1, p_2, provided that $[\![resType(p_2, U^\infty)]\!] \subseteq [\![Foaf]\!]$. To check the latter condition we obtain $resType(p_2, U^\infty) = \{Foaf'\}$, where type *Foaf'* is defined as *Foaf'* $\rightarrow$ *foaf*[*Name* Text]. As $[\![Foaf']\!] \not\subseteq [\![Foaf]\!]$, the condition does not hold. This suggests an error in the program. (We expect that for a correct program the assumptions of Theorem 3 are satisfied.) A more detailed look at how U^∞ has been obtained shows that $Fo = resType(p_1, \emptyset)$ and *Fo* is the inferred results type of rule p_1 (as $_1$ does not w-depend on any rule in the program). This is incompatible with the intention of the programmer, that both arguments of fo are of type *Text*. In this way she finds that p_1 is incorrect.

Chapter 7

Semantic Types

7.1 Ontology Classes in Type Definitions

We have defined a type system for Xcerpt which is based on syntactic types such as types provided by XML schemata. Such a type system is useful for structure-based querying of XML data which refers to syntax of the data. It can be used e.g. to check correctness of queries and to find syntactic type errors. However, XML data may be associated with concepts defined by ontologies. Thus a type system might be also useful for checking semantic correctness of queries. For example, an inconsistency such as a requirement that an XML element represents an individual of a class male and female could be discovered by a system. A simple way of extending the type system from previous chapters by means to check some kind of semantic correctness of Xcerpt queries is to introduce type constants which are names of classes defined by some ontology. When some operations on types representing classes are needed an ontology reasoner can be employed, for example, for computing the intersection of classes.

Example 40. *Consider a query rule:*

```
CONSTRUCT
  list[ var X, var Y ]
FROM
  in[ "book.xml",
    book[ title[ var X ],
      author[ var Y ],
      publisher[ var Y ]
  ]
END
```

The file "book.xml" is of type Book *defined as*

$$\begin{array}{lll} Book & \rightarrow & book\,[\,Title\ Author^{+}\ Publisher\,] \\ Title & \rightarrow & title\,[\,\mathrm{Text}\,] \\ Author & \rightarrow & author\,[\,\mathrm{Text}\,] \\ Publisher & \rightarrow & publisher\,[\,\mathrm{Text}\,] \end{array}$$

Assume that the data in the document "book.xml" is related to some ontology defining classes Person *and* Company*. Moreover, given a data term representing the document "book.xml", the direct subterm of a term* author *is an individual of a class* Person *and the direct subterm of a term* publisher *is an individual of a class* Company*. When we do type inference for variables in the query rule, all variables are mapped to the type* Text*. Thus, there is no type error in the rule if we are taking into account only the syntax of queried data. However the query rule does not have much sense if the authors of books are persons and the publishers are companies (as as the rule requires that the direct subterms of data terms author*[...] *and publisher*[...] *are the same).*

Assume that the Type Definition is extended by two type constants class:Company and class:Person which correspond, respectively, to the classes Company and Person. Assume also that the types Author *and* Publisher *are defined as:*

$$\begin{array}{lll} Author & \rightarrow & author\,[\,class{:}Person\,] \\ Publisher & \rightarrow & publisher\,[\,class{:}Company\,] \end{array}$$

When performing type inference for the variable Y in the rule the system has to find intersection between types class:Company and class:Person. As the types represent ontology classes an ontology reasoner is employed which states that the intersection of the classes Person and Company is empty. Thus, the rule will not return any results if the queried data is correct wrt. the ontology. □

7.2 DigXcerpt: Ontology Queries in Xcerpt

In this chapter we show how structure-based querying of XML data can be combined with ontology reasoning. We present an extension of an XML query language Xcerpt which also allows to query an ontology reasoner. The extension can be seen as a kind of an interface between XML query language and an ontology reasoner. Thus, the extended language, called DigXcerpt, can use ontological information to query XML data. For instance, it can be used to filter XML data returned by a structural query by reasoning on an ontology to which the data is related. This can be illustrated by the following example. Assume that an XML database of culinary recipes is given. Each recipe indicates ingredients (like flour, salt, sugar etc.). We

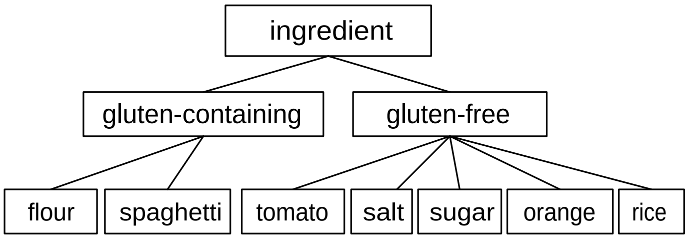


Figure 7.1: Ingredient ontology graph

assume that the names of the ingredients are defined by a standard ontology, accessible separately on the Web and providing also some classification. For example, the ontology may specify disjoint classes of gluten-containing and gluten-free ingredients (see Figure 7.1). To find a gluten-free recipe we would query the XML database for recipes, and query the ontology to check if the ingredients are gluten-free.

To communicate with an ontology reasoner we use DIG interface, so DigXcerpt can query any ontology reasoner supporting DIG (such as RacerPro[1] and and Pellet[2]). DigXcerpt augments Xcerpt rules with ontology queries. The extended language is easy to implement on the top of an Xcerpt implementation and a reasoner with DIG interface, without any need of modifying them.

Related work. The problem of combination of XML queries with ontology queries seems to be important for the Semantic Web. There exist many approaches combining a Description Logic and a language with logical semantics, like Datalog. For references see e.g. the articles of Eiter et al. and of Rosati in [7]. In contrast to these approaches, we add ontology queries to a language whose semantics is operational. Moreover we re-use an existing ontology reasoning system and a query language implementation, which is impossible for most of the approaches mentioned above.

An intermediate approach is presented in [6]. Ontology queries which cannot be solved are accumulated during rule computation. The fixed point semantics specifies the formulae (built out of delayed queries) with which the reasoner is eventually queried. This makes possible reasoning by cases. In contrast, our operational semantics makes the programmer decide when an ontology query is evaluated. The approach of [6] allows only Boolean ontology queries. It is applicable to a certain (negation-free) subset of Xcerpt. Our approach imposes no restriction on ontology queries and is applicable to full Xcerpt.

A different approach is that of [48], where XQuery is used both to query data and to perform (restricted kinds of) reasoning.

The approach described here is based on the paper [30] and its extended version [31]. Preliminary versions of the current approach were presented in

[1]`http://www.racer-systems.com/`

[2]`http://www.mindswap.org/2003/pellet/`

the papers [52] and [29].

7.2.1 Syntax and Semantics

This section presents an extension of Xcerpt, called DigXcerpt, allowing attaching ontology queries to Xcerpt rules. A DigXcerpt program is a set of Xcerpt rules and extended rules. The syntax of an extended construct rule is

```
CONSTRUCT
   head
WHERE
   dig [ digResponseQuery, digAskConstruct ]
FROM
   body
END
```

Analogical syntax can be used for extended goal rules (with the keyword GOAL instead of CONSTRUCT). Sometimes the rule will be denoted as

$$head \leftarrow (digResponseQuery, digAskConstruct), body$$

(without distinguishing between a construct and goal rule). *digAskConstruct* is a construct term intended to produce DIG ask statements which are sent to the reasoner. *digResponseQuery* is a query term that is applied to the response statements returned by the reasoner. As in an Xcerpt rule, *head* is a construct term and *body* is a query to the results of other rules and/or to external resources.

In what follows we assume existence of a fixed ontology and an ontology reasoner to which the ontology queries refer. We also assume fixed data terms $\delta(r_i)$ associated with external resources occurring in programs. This is a formal definition of extended query rule.

Definition 24. *An* **extended query rule***(shortly,* extended rule*) is an expression of the form $c \leftarrow (q, c'), Q$, where Q is a query, q is a query term, c' is a construct term without grouping constructs and c is a construct term not of the form* `all` *c'' or* `some` *k c''. Let* `or`$(Q_1, \ldots, Q_n)$ *be a disjunctive normal form of Q. The variables occurring in c, c', q must satisfy the following conditions:*

- *every variable of c' occurs in each Q_i, for $i = 1, \ldots, n$,*
- *every variable of c which does not occur in q 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 extended rule.*

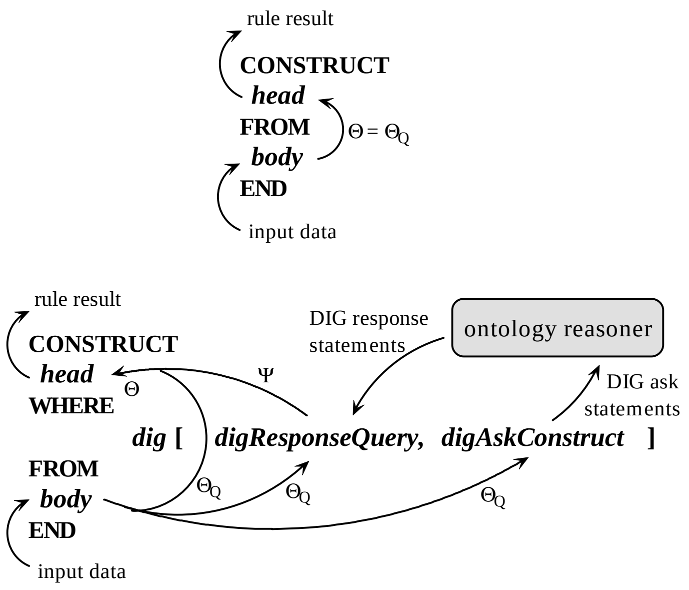


Figure 7.2: Data flow in an Xcerpt rule and in an extended rule. $\Theta, \Theta_Q, \Psi_\theta$ are sets of substitutions as described below and $\Psi = \bigcup_{\theta \in \Theta_Q} \Psi_\theta$.

Now we describe the semantics of an extended construct rule $c \leftarrow (q, c'), Q$. First, we assume that the rule does not contain grouping constructs. Let Θ_Q be the set of non redundant answer substitutions obtained by evaluation of the body Q of the rule. Each $\theta \in \Theta_Q$ is applied to the construct term c'; this produces a DIG ask statement $c'\theta$ to be sent to the reasoner. For each $c'\theta$ the reasoner returns a DIG response statement d_θ. To each d_θ the query term $q\theta$ is applied, producing a set Ψ_θ of substitutions. (The domain of the substitutions are those variables that occur in q and do not occur in Q.) A set of substitutions $\Theta = \{\, \theta \cup \sigma \mid \theta \in \Theta_Q,\ \sigma \in \Psi_\theta \,\}$ is constructed. (Informally: the substitutions bind the rule variables according to the results of Q and of DIG querying.) Now the set of results of the whole rule is $\{\, c\theta \mid \theta \in \Theta \,\}$ (the substitutions from Θ are applied to the head of the rule). Figure 7.2 presents the data flow in an extended rule.

The semantics above has to be generalized for the case where a construct term c of an extended rule $c \leftarrow (q, c'), Q$ contains a grouping construct. This is expressed by the following definition which extends Definition 8.

Definition 25 (Result of an extended query rule and a set of data terms). *Let $p = c \leftarrow (q, c'), Q$ be an extended query rule, Z be a finite set of data terms, $\Theta_Q = \{\theta_1, \ldots, \theta_n\}$ be the set of non redundant answer substitutions for Q and Z. Let $a_1, \ldots, a_n$ be data terms representing DIG ask statements such that $a_i = c'\theta_i$ and let $r_1, \ldots, r_n$ be the corresponding DIG response* *statements. Let $q_1, \ldots, q_n$ be query terms such that $q_i = q\theta_i$. Let Ψ_i be the set of non redundant answer substitutions for q_i and r_i, and $\Theta = \{\, \theta_i \cup \sigma \mid \theta_i \in \Theta_Q,\, \sigma \in \Psi_i \,\}$.*

If $\Theta' \in \Theta/_{\simeq_{FV(c)}}$ then $\Theta'(c)$ is a **result** *of an extended rule $c \leftarrow (q, c'), Q$. The* **set of results** *of $p = c \leftarrow (q, c'), Q$ and Z is denoted as $res(p, Z)$.*

The semantics of an extended goal rule $c \leftarrow (q, c'), Q$ is similar to that of the extended construct rule $c \leftarrow (q, c'), Q$. The difference is — as in Xcerpt — that the goal rule produces only one answer (from the set of answers of the construct rule).

The new *WHERE* part in the extended rule allows to ask an ontology arbitrary queries expressible in DIG. One category of such queries are Boolean queries for which answer *true* or *false* (or *error*) can be obtained. This kind of queries can be used to filter out some data from the XML document based on the ontological information. For example, an extended rule can be used to filter out recipes which are gluten-free. In such case, the rule would have a query term *true*[[ ]] as the *digResponseQuery*, thus it would filter out those answer substitutions for the variables in the *body* for which the corresponding reasoner answer was not *true*. It seems that a need for such filtering is relatively common. Hence, to simplify syntax, we assume that the *digResponseQuery* in the *WHERE* part is optional and by default it is a query term *true*[[ ]].

The semantics of DigXcerpt programs is defined in terms of the semantics of single rules in the same way as for Xcerpt. Formally this is done by Definitions 11,15. The difference is that in the case of DigXcerpt they refer not only to Definition 8 (semantics of an Xcerpt rule) but also to Definition 25 (semantics of an extended rule). If we want to deal with the features of Xcerpt not considered in Section 2.1.2, like negation, then the semantics from [50] applies.

Example 41. (Boolean ontology query (1)) *Consider the XML document* recipes.xml *from Example 2 and the culinary ingredients ontology from Figure 7.1. We assume that the ontology is loaded into an ontology reasoner with which we can communicate using DIG. We also assume that the names of the ingredients used in the XML document are defined by the ontology. We want to find all the recipes in the XML document which are not gluten-free. This can be achieved using a rule:*

```
CONSTRUCT
 bad-recipes[ all name[ var R ] ]
WHERE
 dig[ subsumes[
        catom[ attr{ name[ "gluten-containing" ]}],
        catom[ attr{ name[ var I ] } ] ] ]
FROM
 in[ resource[ "file:recipes.xml" ],
     desc recipe[[
              name[ var R ], ingredient[[ name[ var I ] ]] ]]  ]
        END
```

The body of the rule (the FROM part) extracts the names of recipes together with their ingredients and assigns respective substitutions to the variables R, I. This results in an answer substitution set Θ_Q. Based on Θ_Q the digAskConstruct `subsumes[...]` *constructs DIG ask statements asking whether particular ingredients (values of the variable I) are* gluten-containing. digResponseQuery *is omitted in the* WHERE *part which means that its default value* `true[[ ]]`*is used. The final set Θ of answer substitutions which are applied to the head of the rule contains those substitutions from Θ_Q for which the reasoner answer for the corresponding DIG ask statement was* true *i.e. the substitutions where the variable I is bound to data terms representing* gluten-containing *ingredients:* flour *and* spaghetti*. As these ingredients occur in* Recipe2 *and* Recipe3 *the final result of the rule is*

```
        bad-recipes[ name[ "Recipe2" ], name[ "Recipe3" ] ]
```
□

Example 42. (Boolean ontology query (2)) *Let us now construct a query producing a list of gluten-free recipes, instead of those containing gluten. This may be less obvious, as we have to make sure that none of the ingredients of a recipe contains gluten. One way to find gluten-free recipes is by using the rule from Example 41, and extract from* recipes.xml *all the recipes not found by that program to contain gluten. Thus the program producing a list of gluten-free recipes consists of the rule from the previous example and the following rule:*

```
CONSTRUCT
 good-recipes[ all name[ var R ] ]
FROM
 and[
   in[ resource[ "file:recipes.xml" ],
     desc recipe[[ name[ var R ] ]] ],
   not bad-recipes[[ name[ var R ] ]]
 ]
END
```

The result of the program is

```
good-recipes[ name[ "Recipe1"] ]
```
□

The previous examples illustrate usage of a filter where Boolean queries are sent to an ontology reasoner. The next one presents a more general query.

Example 43. (Non boolean ontology query) *Consider the ingredients ontology (Figure 7.1) extended with a class* vitamin*, its three subclasses:* A,B,C*, and a property* contained_in*. The extended ontology contains also axioms which indicate in which ingredients a particular vitamin is contained.*

The axioms state that the vitamin A is contained in tomato, vitamin B in tomato, orange, flour and spaghetti, and vitamin C in orange and tomato. For example, using description logics syntax, one of the axioms can be expressed as $A \sqsubseteq \exists contained\_in.tomato$.

The following DigXcerpt rule queries the document recipes.xml *and the ontology to provide a list of vitamins for each recipe in the document. The* WHERE *part of the rule contains a construct term* `descendants[...]` *producing ontology queries which ask about vitamins included in a particular ingredient. The reasoner answers are queried by the query term* `conceptSet[[...]]` *from the* WHERE *part.*

```
CONSTRUCT
 vit-recipes[ all recipe[ var R, all var V ] ]
WHERE
 dig[
   conceptSet [[ synonyms[[ catom[ attr{ name[var V] } ] ]] ]],
   descendants[
     some[ ratom[ attr{ name["contained_in"] } ],
     catom[ attr{ name[var I] } ] ]   ]   ]
FROM
 in[ resource[ "file:recipes.xml" ],
     desc recipe[[
       name[var R], ingredient[[ name[var I] ]] ]]  ]
END
```

The result of the rule is:

```
vit-recipes[ recipe[ "Recipe1", "B", "C" ],
             recipe[ "Recipe2", "B" ],
             recipe[ "Recipe3", "A", "B", "C" ] ]
```

□

7.2.2 Implementation

This section presents a way DigXcerpt can be implemented on the top of Xcerpt engine i.e. without any modification of Xcerpt implementation. Evaluation of a DigXcerpt program can be organized as a sequence of executions of Xcerpt programs and ontology queries. This can be implemented in a rather simple way; an implementation iteratively invokes an Xcerpt system and an ontology reasoner with a DIG interface.

We begin with discussing implementation of DigXcerpt programs where there is no recursion over extended rules i.e. no extended rule statically depends on itself. We call such programs non *DIG recursive programs*.

Let $\mathcal{P} = (P, G)$ be a non DIG recursive DigXcerpt program and $e_1, \ldots, e_n$ be the extended rules from P such that each rule e_i does not statically depend on any rule $e_{i+1}, \ldots, e_n$. The ordering $e_1, \ldots, e_n$ can be obtained by topological sorting of the dependency graph for the extended rules in P.

The first step is to compile the extended rules of P into pairs of rules. Each extended rule e_i of P of the form $c \leftarrow (q, c'), Q$ is translated into a pair of rules:

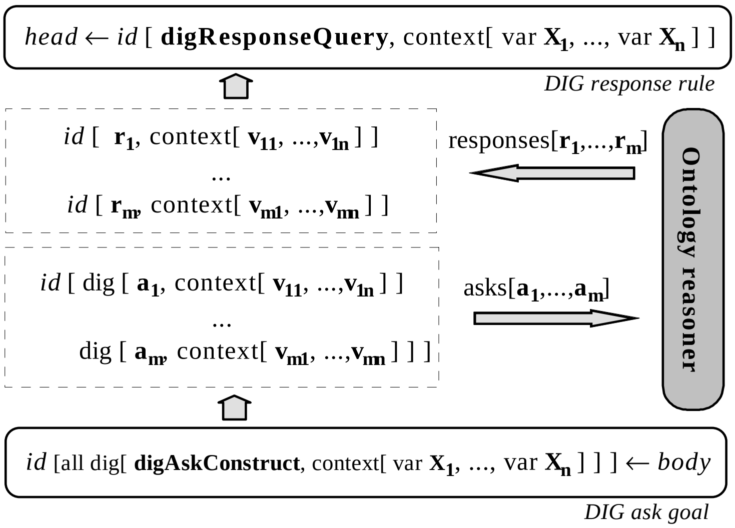

head ← id [digResponseQuery, context[var X1, ..., var Xn]]
DIG response rule
id [r1, context[v11, ...,v1n]]
...
id [rm, context[vm1, ...,vmn]]
responses[r1,...,rm]
Ontology reasoner
id [dig [a1, context[v11, ...,v1n]]
...
dig [am, context[vm1, ...,vmn]]]
asks[a1,...,am]
id [all dig[digAskConstruct, context[var X1, ..., var Xn]]] ← body
DIG ask goal

Figure 7.3: Evaluation of DIG rules: DIG ask goal produces DIG ask statements which are sent to a reasoner; DIG response rule queries data terms constructed out of the reasoner's responses.

- a DIG ask goal dg_i:
  $id_i[$ `all` $dig[c', context[X_1, \ldots, X_l]]] \leftarrow Q$,
- and a DIG response rule dr_i:
  $c \leftarrow id_i[q, context[X_1, \ldots, X_l]]$.

The DIG ask goal is used to produce DIG ask statements to be sent to the reasoner and the DIG response rule is used to capture the reasoner responses. $X_1, \ldots, X_l$ are the variables occurring in the query Q. The term $context[\ldots]$ is used here to pass the values of the variables from the body Q of the DIG ask goal to the head c of the response rule. The construct `all` in the DIG ask goal is added to collect all the results of the query Q.

The purpose of the labels $id_1, \ldots, id_n$ is 1. to associate DIG ask goals with the corresponding DIG response rules, and 2. to distinguish the data produced by the rules of P from the data related to the implementation of extended rules. So it is required that $id_1, \ldots, id_n$ are distinct and that no id_i occurs in P as the label of the head of a non goal rule of P. (Moreover, if the head of some non goal rule of P is a variable then no id_i can occur in the data to which P is applied.) Figure 7.3 shows how DIG ask and response rules are evaluated.

Out of the sets of DigXcerpt rules P, G we construct new sets of Xcerpt rules P', G', which are the sets P, G with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $P'' = P' \setminus G'$. Then a sequence of Xcerpt programs $\mathcal{P}_0, \ldots, \mathcal{P}_n$ is constructed, where $\mathcal{P}_0 = (P'' \cup \{dg_1\}, \{dg_1\})$ (and $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are described later on). For $i = 1, \ldots, n$ we proceed as follows. For a set of data terms R, $r(R)$ denotes rules with empty bodies representing the data terms from R. (We do not distinguish between XML elements and their representation as data terms.)

- Program $\mathcal{P}_{i-1}$ is executed by Xcerpt. A data term $id_i[dig[a_1, c_1], \ldots, dig[a_m, c_m]\,]$ is obtained (it is produced by a goal rule dg_i), where $a_1, \ldots, a_m$ are DIG ask statements. Out of $a_1, \ldots, a_m$ a DIG ask request is built. (The DIG ask request is an XML document additionally containing a header with DIG namespace declarations, and unique identifiers for the elements corresponding to $a_1, \ldots, a_m$.)

- The DIG ask request is sent to the DIG reasoner. The reasoner replies with a response that (after removing its attributes) is $responses[r_1, \ldots, r_m]$, where each r_i is an answer for a_i. A set of Xcerpt facts $R_i = \{\, id_i[r_1, c_1], \ldots, id_i[r_m, c_m]\,\}$ is constructed. (The set contains the results from the reasoner together with the corresponding context information, to be queried by the DIG response rule dr_i. The results of executing the rule dr_i in an Xcerpt program containing the rules $\{dr_i\} \cup r(R_i)$, (and no other rules producing data terms with the label id_i) are the same as the results of executing e_i in program $\mathcal{P}$ according the the semantics described in the previous section.)

- If $1 \leq i < n$ then $\mathcal{P}_i = (P'' \cup \bigcup_{j=1}^{i} r(R_j) \cup \{dg_{i+1}\}, \{dg_{i+1}\})$. (Program $\mathcal{P}_i$ contains the reasoner results obtained up to now. They are to be queried by the DIG response rules $dr_1, \ldots, dr_i$. The goal of $\mathcal{P}_i$ is dg_{i+1} in order to produce the next query to the reasoner.)

  If $i = n$ then $\mathcal{P}_n$ is an Xcerpt program $(P'' \cup \bigcup_{j=1}^{n} r(R_j) \cup G', G')$.

As the last step, $\mathcal{P}_n$ is executed by Xcerpt, producing the final results of $\mathcal{P}$.

The results are the same as those described by the DigXcerpt semantics of Section 7.2.1. (We skip a formal justification of this fact.) As an additional consequence we obtain that the results do not depend of the ordering of $e_1, \ldots, e_n$ (which may be not unique).

Example 44. *Here we illustrate an evaluation of a simple DigXcerpt program. Consider a program $\mathcal{P}$ consisting of the extended rule from Example 41, changed into a goal rule:*

```
GOAL
 bad-recipes[ all name[ var R ] ]
WHERE
 dig[ subsumes[
        catom[ attr{ name["gluten-containing"] } ],
        catom[ attr{ name[ var I ] } ]   ]   ]
FROM
 in[ resource[ "file:recipes.xml" ],
     desc recipe[[
       name[ var R ], ingredient[[ name[ var I ] ]] ]] ]
END
```

First, the rule is translated into the corresponding DIG response rule (which is a goal in this case) and a DIG ask goal:

```
GOAL
 bad-recipes[ all name[ var R ] ]
FROM
 #g1[ true[[ ]], context[ var I, var R ] ]
END

GOAL
 #g1[ all dig[
        subsumes[
          catom[ attr{ name["gluten-containing"] } ],
          catom[ attr{ name[ var I ] } ]  ],
        context[ var I, var R ]  ]  ]
FROM
 in[ resource[ "file:recipes.xml" ],
   desc recipe[[ name[ var R ],
                 ingredient[[ name[var I] ]] ]] ]
END
```

Now a sequence of programs $\mathcal{P}_0, \mathcal{P}_1$ is constructed. The program $\mathcal{P}_0$ contains only the DIG ask goal which produces the following data term:

```
#g1[ dig[ subsumes[ catom[ attr{ name["gluten-containing"] } ],
                    catom[ attr{ name["sugar"] } ]  ],
          context[ "sugar", "Recipe1" ]   ],
     dig[ subsumes[ catom[ attr{ name["gluten-containing"] } ],
                    catom[ attr{ name["orange"] } ]  ],
          context[ "orange", "Recipe1" ]   ],
     dig[ subsumes[ catom[ attr{ name["gluten-containing"] } ],
                    catom[ attr{ name["flour"] } ]  ],
          context[ "flour", "Recipe2" ]   ],
     dig[ subsumes[ catom[ attr{ name["gluten-containing"] } ],
                    catom[ attr{ name["salt"] } ]  ],
          context[ "salt", "Recipe2" ]   ],
     dig[ subsumes[ catom[ attr{ name["gluten-containing"] } ],
                    catom[ attr{ name["spaghetti"] } ]  ],
          context[ "spaghetti", "Recipe3" ]   ],
     dig[ subsumes[ catom[ attr{ name["gluten-containing"] } ],
                    catom[ attr{ name["tomato"] } ]  ],
          context[ "tomato", "Recipe3" ]   ]   ]
```

The data term contains DIG ask statements asking whether particular ingredients are gluten-containing. *The additional information attached to each ask statement (its context) is the name of the ingredient queried about and the corresponding name of the recipe.*

Out of these data terms a DIG ask request (which is an XML document) is built. The request contains six DIG ask statements which according to the DIG syntax are augmented by unique identifiers, here $1, \ldots, 6$*:*

```
<asks ...>
   <subsumes id="1">
     <catom name="gluten-containing"/>
     <catom name="sugar"/>
   </subsumes>
   ...
   <subsumes id="6">
     <catom name="gluten-containing"/>
     <catom name="tomato"/>
   </subsumes>
</asks>
```

The DIG ask request is sent to the ontology reasoner. Its XML answer represented by a data term is (the attributes of the element responses *are removed):*

```
responses[ false[ attr{ id["1"] } ], false[ attr{ id["2"] } ],
           true [ attr{ id["3"] } ], false[ attr{ id["4"] } ],
           true [ attr{ id["5"] } ], false[ attr{ id["6"] } ] ]
```

Based on the answer the following set R_1 of data terms is constructed:

```
#g1[ false[ attr{ id["1"] } ], context[ "sugar", "Recipe1" ] ]
#g1[ false[ attr{ id["2"] } ], context[ "orange", "Recipe1" ] ]
#g1[ true [ attr{ id["3"] } ], context[ "flour", "Recipe2" ] ]
#g1[ false[ attr{ id["4"] } ], context[ "salt", "Recipe2" ] ]
#g1[ true [ attr{ id["5"] } ], context[ "spaghetti", "Recipe3" ] ]
#g1[ false[ attr{ id["6"] } ], context[ "tomato", "Recipe3" ] ]
```

The final program $\mathcal{P}_1$ to be evaluated by Xcerpt consists of the DIG response goal and rules $r(R_1)$:

```
GOAL
  bad-recipes[ all name[ var R ] ]
FROM
  #g1[ true[[ ]], context[ var I, var R ] ]
END

CONSTRUCT
  #g1[ false[ attr{ id["1"] } ], context[ "sugar", "Recipe1" ] ]
END
...
CONSTRUCT
  #g1[ false[ attr{ id["6"] } ], context[ "tomato", "Recipe3" ] ]
END
```

The result of $\mathcal{P}_1$ is the result of the initial program $\mathcal{P}$:

```
bad-recipes[ name[ "Recipe2" ], name[ "Recipe3" ] ]
```
□

DIG recursive programs. In the presented algorithm we assumed that $\mathcal{P}$ is a non DIG recursive program. DIG recursive programs, without grouping constructs can be dealt with as follows. Let $\mathcal{P}$ be an arbitrary DigXcerpt program and $e_1, \ldots, e_n$ be the extended rules from $\mathcal{P}$ not necessarily satisfying the previous condition on their mutual dependencies. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$, and P', G', P'' be as defined earlier. We construct a sequence of Xcerpt programs $\mathcal{P}^0, \mathcal{P}^1, \mathcal{P}^2, \ldots$, where $P^0 = (P'' \cup E, E)$, $P^j = (P'' \cup E \cup r(R^j), E)$ for $j > 0$, and R^j is defined below. (R^j represents the reasoner responses for the ask statements produced by the program $\mathcal{P}^{j-1}$.)

Each goal dg_i of $\mathcal{P}^{j-1}$ produces a result $id_i[dig[a_1, c_1], \ldots, dig[a_m, c_m]\,]$. As in the previous approach, a DIG ask request is constructed out of the result. The corresponding response of the reasoner is represented, as previously, by a set of Xcerpt facts $R^j_i = \{\, id_i[r_1, c_1], \ldots, id_i[r_m, c_m]\,\}$. Now $R^j = R^j_1 \cup \ldots \cup R^j_n$. It holds that $R^j \subseteq R^{j+1}$ for $j = 1, 2, \ldots$.

The programs $\mathcal{P}^0, \mathcal{P}^1, \ldots$ are executed (by Xcerpt) until the results of the program $\mathcal{P}^k$ are the same as the results of $\mathcal{P}^{k-1}$. Finally, a program $\mathcal{Q} = (P'' \cup r(R^k) \cup G', G')$ is constructed. The results of $\mathcal{Q}$ are the same as the results of $\mathcal{P}$ described by the semantics of DigXcerpt. A formal justification can be found in Appendix A.3 (Theorem 4).

Notice that for non DIG recursive programs this method may be less efficient than the previous one, as it sends more ask requests to the reasoner.

Applying this method to a program with grouping constructs may lead to incorrect results. This is because an evaluation of an Xcerpt program with such constructs is a sequence of evaluations of its *strata* (cf. Section 2.1.2), in a particular order. In the non DIG recursive case the stratification did not pose any problems. The order of extended rules $e_1, \ldots, e_n$ coincides with the order of strata. Thus the results produced by a goal dg_i added to a program $\mathcal{P}_j$, $j > i$, are the same as the results of dg_i in $\mathcal{P}_{i-1}$. This is not the case for a program that both is DIG recursive and requires stratification. For such programs, the method for handling DIG recursive programs, described above, applies to each stratum separately. The division of a DigXcerpt program into strata and the way of sequential evaluation of strata is the same as in Xcerpt (cf. Section 2.1.2). We omit the details of the algorithm for this case.

The presented algorithm for evaluation of DigXcerpt programs may not terminate for recursive programs. However this is also the case in standard Xcerpt. It is up to the programmer to make sure that a recursive program will terminate.

7.2.3 Discussion

We believe that the examples we presented illustrate practical usability of the proposed approach. The examples use arbitrary ontology queries, not only Boolean ones. We put no restriction on usage of DIG. For instance with a query term *error*[[ ]] used as a *digResponseQuery* a DigXcerpt program can check for which data the reasoner returns an error. Ability to modify ontologies could be added to DigXcerpt, by using *DigAskConstruct* that sends DIG tell statements to the reasoner.

Our approach abstracts from a way XML data is related to an ontology. It is left to a programmer. In our examples XML data is associated with ontology concepts by using common names i.e. XML element names are the same as class names. However, our approach is not restricted to this way of associating. For example, the association may be defined through element attributes.

The semantics of DigXcerpt imposes certain implicit type requirements on programs. The data terms produced by a *digAskConstruct* should be DIG ask statements (represented as data terms). *digResponseQuery* should match data terms being DIG response statements. It is better to check such conditions statically, instead of facing run-time errors. For this purpose the descriptive type system for Xcerpt from the previous chapters can be used.

DigXcerpt in its current form requires the programmer to use the verbose syntax of DIG ask and response statements. On the other hand, the tedious details of constructing DIG requests out of DIG ask statements and extracting DIG response statements out of DIG responses are done automatically. Still, one has to specify a construct term *digAskConstruct* for construction of a DIG ask statement, and a query term *digResponseQuery* to match DIG response statements. Dealing with details of DIG syntax may be considered cumbersome and too low level. It may be useful to introduce simpler and more concise syntax for both *digAskConstruct* and *digResponseQuery*. For example, the WHERE part from the rule in Example 43 could be abbreviated as `WHERE V in descendants[ some["contained_in", var I] ]`.

We expect that the approach discussed here can be applied to composing some other XML query languages (such as XQuery) with ontology querying.

The work on implementing DigXcerpt is in progress. A prototype implementation of the Xcerpt extension from our previous work [29] is available on-line[3]. Implementation of DigXcerpt requires only slight modification of that prototype.

[3]`http://www.ida.liu.se/digxcerpt/`

Chapter 8

Conclusions

This work provides a type system for the XML query language Xcerpt. The type system is descriptive; this means that types approximate the semantics of programs. In our approach types are sets of data terms. For specifying types we adapted and slightly generalized the formalism of Type Definitions [15].

In general, Type Definitions are not closed under intersection. This is due to simple treatment of unordered content models in this formalism. We present a class of intersectable Type Definitions (Section 3.2.2) which is closed under intersection. It seems that in practice most of Type Definitions are intersectable. For instance all definitions without unordered content models are. We provide an algorithm of computing type intersection for such definitions, and an algorithm of approximating a type by its intersectable superset. The latter makes the type system able to deal with non intersectable content models. We also suggest a generalization of Type Definitions by introducing types representing ontology classes (Section 7.1). This should make possible adding semantic types to our approach.

In practice, when the available type information is given in some schema language then, to apply our type system, the schema has to be transformed into a Type Definition describing the same type. (Formally, a set defined by the schema is a set of XML documents, and the Type Definition describes the corresponding set of data terms). Our prototype performs such transformation for DTDs, and can be extended to deal with XML Schema or RELAX NG. In some cases a type described using XML Schema or RELAX NG cannot be defined by a Type Definition (see Section 3.3). Then a Type Definition describing a superset of the given type can be used.

The type system presented in this thesis makes possible:

- Type inference for a single Xcerpt rule. We show how to compute types containing all the results of a single Xcerpt rule, provided that the rule is applied to data of a given type. The algorithm is introduced in an abstract form of derivation rules (Section 4.2.2). This makes possible a formal proof of its correctness (Appendix A.1.1), and deriving a concrete algorithm (Section 4.2.5). The algorithm is of exponential complexity. However experiments show that it is sufficiently efficient in practice. (Section 4.2.5 shows in which cases we can expect problems). In Section 4.2.6 we propose how to generalize the abstract algorithm for most of the Xcerpt constructs not formally dealt with in this work.

- Type inference for an Xcerpt program. Section 4.2.3 presents a method of computing types containing the results of a program, provided that the program is applied to data of a given type. The method employs type inference for a single Xcerpt rule. It can be seen as an instance of the abstract interpretation paradigm. A special feature is that in computing a fixed point the number of iterations is known in advance. This makes it possible to avoid an expensive test for a fixed point (i.e. type inclusion with non proper Type Definitions). We provide a formal proof of correctness and termination of the method.

  In Section 4.2.4 we discuss conditions (on the program and on the type specification), under which the type inference is exact. Part of the discussion is formal, and the corresponding proofs are given in Appendix A.2.

- Type checking for Xcerpt programs i.e. proving correctness of an Xcerpt program (or a query rule) wrt. a given type specification. Type correctness means that whenever the program (the rule) is applied to data from a given type (of the database) the result is from a given type (of expected results). Type correctness can be proved by successful checking whether the inferred result type is included in the specified one.

  If the type inference is exact then failure of type checking implies type incorrectness of the program (rule). Otherwise such failure may be interpreted only as a hint of possible incorrectness. Section 4.4 further discusses relations between errors in the program and the results of type inference and type checking.

- Determining dependencies between rules in Xcerpt programs. Understanding rule dependencies and finding them effectively is necessary for efficient evaluation of Xcerpt programs [35]. Previously, static dependency [50] has been used; it is a rather rough approximation of the actual, dynamic dependency. We show how a more precise approximation can be effectively obtained by employing type analysis (Section 4.3).

The theoretical part of this work is complemented by a prototype implementation (Chapter 5). The prototype is written in Haskell and is an extension of the prototype implementation of Xcerpt. The prototype is available on-line. Most of the techniques described in this thesis are implemented, a main exception is type inference for recursive programs. We presented example scenarios demonstrating usefulness of the proposed approach for indicating errors in Xcerpt programs (Chapter 6). The prototype has been applied in most of these examples.

We also present an extension of Xcerpt, called DigXcerpt, that allows to query ontologies in addition to XML data (Section 7.2). Extended programs communicate with an ontology reasoner using DIG interface. No restrictions are imposed on Xcerpt and on the DIG ask statements used. In particular, ontologies can be queried with arbitrary, not only Boolean, queries. We present a way of implementation of DigXcerpt by employing an existing Xcerpt implementation and an existing ontology reasoner; they are treated as "black boxes" (no modifications to the Xcerpt system or the reasoner are needed). Appendix A.3 provides a soundness proof of the implementation approach wrt. the semantics of DigXcerpt.

Related work. The type system of XQuery [25] differs substantially from our approach (see Section 4.5). A main difference is that we add types to an untyped language, while in XQuery types are an essential part of the language. In the semantics of XQuery, data values are augmented with type information, and the results of some constructs depend on the type annotation. Our typing approach is descriptive, types are sets used to approximate the semantics of programs. The type system of XQuery is prescriptive; types are expressions, and they are indispensable part of the semantics. Treating types as sets is a secondary notion – to each type there corresponds a certain set of data objects (defined by the relation *matches* [25]). Type correctness of programs is defined by numerous conditions, related to particular constructs of XQuery. The conditions can be checked dynamically or – maybe not all of them – statically. In contrast, type correctness in our approach is specified by a single condition that the results of a program are members of the specified result type (if the input data queried by the program are within specified types). Another difference is that data terms used in our approach are more abstract than the XQuery data model. As a consequence, types in XQuery deal with more details of XML documents than the types considered in this approach.

XDuce [37] is an XML processing language, in which types play important role. Types in XDuce are (expressions denoting) sets. This is different from XQuery, and similar to our approach. As in this work, a data object in XDuce may be a member of many types. XDuce is statically typed, the typechecking is performed at compile time. XDuce is based on pattern matching, like Xcerpt, not on path expressions, like XQuery. The pattern matching mechanism is less sophisticated than that of Xcerpt. For example, there are no unordered patterns in XDuce. An interesting feature is that patterns are closely related to type expressions. A specific property of pattern matching is that for any input value it results in exactly one variable binding. The type system of XDuce can be classified as prescriptive, as its semantics is defined only for well-typed programs.

Although XDuce formalism for type specification is similar to Type Definitions there are some differences. Elements of types are sequences of terms, in contrast to our approach and similarly to XQuery. Thus the formalism allows to assign type names to sets of sequences of data terms expressible by regular expressions, while Type Definitions allow to assign type names to sets of data terms of one kind, ordered or unordered, and with the same label. In contrast to Type Definitions, XDuce and its type formalism do not deal with unordered trees. Another difference is that XDuce, similarly to XQuery, does not impose efficiency related restrictions on the formalism for specifying types. Thus restrictions like those in our proper Type Definitions or in XML Schema (*Element Declarations Consistent* and *Unique Particle Attribution*) are absent here. As a result, the class of sets defined by the XDuce formalism is closed under union and complement. Inefficiency issues related to this choice are accepted in return for a clean and powerful language design.

Substantial theoretical work on types for XML transformations has been done from decidability and complexity point of view (see e.g. [43] and the references therein). In that work, XML query and transformation languages are usually abstracted as unranked tree transducers. Also, mainly exact typing algorithms are considered.

Future work. An obvious subject of future work is completion of the prototype by implementing its missing features. Another topic is generalization of the presented approach to full Xcerpt. Adding most of the missing Xcerpt constructs is discussed in Section 4.2.6. However extending the system for data terms representing graphs has not been addressed in this work.

Another subject of future work is improvement of the presented algorithms. It is possible that in some cases accuracy should be better traded for efficiency, and the most general type *Top* should be used instead of computing a more precise approximation.

The treatment of unordered content models in the formalism of Type Definitions is rather simple. It would be useful to improve it, in order to make the formalism closed under intersection, and to obtain better approximation of sets of data terms with unordered arguments. Possible suggestions may be found in [46, 60].

An interesting issue has been purposefully left out of the scope of this work. Namely, data terms or XML documents can be used to represent non-tree data structures, for instance RDF graphs. Actually, in full Xcerpt data terms may include a kind of pointers; such terms are treated as representation of graphs. Such representation of data structures as data terms is not unique. A given graph can be represented by various data terms, which differ substantially. This imposes an equivalence relation: data terms representing the same data structure are equivalent. Type systems, like the one presented here, do not respect this equivalence. A type may include a data term d but not include a data term d' which is equivalent to d. A subject of future work is construction of type systems which consider not only data terms but also the data structures they represent.

Appendix A

Proofs

A.1 Type System Correctness

The section presents proofs of Theorem 1, Theorem 2, Proposition 2, and Theorem 3, essential for the correctness of the type system.

A.1.1 Type Inference for Rules

Here we prove Theorem 1 which is similar to Theorem 20 of [12]. Its proof is also similar and to prove Theorem 1 we use lemmata and propositions from [12]. However, the set of rules to which the lemmata and propositions refer, is slightly different in this thesis than in [12].

One difference is in the rule (Targeted Query Term), which instead of the condition $\delta(r) \in [\![T]\!]$ has a condition $\mathit{type}(r) = T$. The rule (Pattern) for query terms is augmented with conditions related to newly introduced type *Top*. The conditions for rule (Pattern) for construct terms are corrected. Additionally we provide another typing rule for construct terms, namely (Var Approx), which was absent in [12]. The last difference concerns the rule (Query Rule) which now has an additional condition saying that $\mathit{substitutions}(\Gamma) \neq \emptyset$ for each $\Gamma \in \Psi$.

Moreover, introduction of a type *Top* in this thesis allows to generalize Lemma 27 of [12] (by removing the requirement that $\Gamma(X) \neq \mathit{Top}$). As the mentioned modifications require only small and simple changes in lemmata, propositions and proofs recalled from [12], we do not describe them here. However we introduce Proposition 7 which is a new version of Proposition 23 of [12]. The new version of the proposition is necessary due to the fact that we have introduced in this thesis the notion of an answer for a query and a set of data terms.

First, we need to recall an auxiliary definition of Γ_θ (defined earlier in [12]).

Definition 26. *Given a Type Definition D and a substitution θ assigning data terms to variables, the mapping Γ_θ is defined as:*

$$\Gamma_\theta(X) = T_1 \cap \ldots \cap T_n,$$

where $X \in dom(\theta)$ and $T_1, \ldots, T_n$ are type names of D such that $\{T_1, \ldots, T_n\} = \{T \mid X\theta \in [\![T]\!]_D\}$.

By definition $\theta \in substitutions(\Gamma_\theta)$.

This is a new version of Proposition 23 of [12]:

Proposition 7. *Let D be a Type Definition, U a set of type names and Q a query such that for each targeted query term $\mathtt{in}(r, q)$ in Q there is a type name $T = type(r)$ defined in D. Let Z be a set of data terms such that $Z \subseteq [\![U]\!]_D$. If θ is an answer for Q and Z 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 $U \neq \emptyset$. θ is an answer for Q and $d \in Z$, such that $d \in [\![T_i]\!]$ for some $T_i \in U$. By Lemma 22 of [12], $D \vdash Q : T_i \triangleright \Gamma_\theta$. By rule (Query Term), we obtain $D \vdash Q : U \triangleright \Gamma_\theta$.

- If Q is a targeted query term $\mathtt{in}(r, q)$, θ is an answer for q and $\delta(r)$. Let $T = type(r)$. Then $\delta(r) \in [\![T]\!]$. By Lemma 22 of [12], $D \vdash q : T \triangleright \Gamma_\theta$. By rule (Targeted Query Term), we obtain $D \vdash Q : U \triangleright \Gamma_\theta$.

- Let Q be of the form $\mathtt{and}(Q_1, \ldots, Q_p)$. By Definition 4, for each $i \in \{1, \ldots, p\}$, θ is an answer for Q_i and Z. By the inductive assumption, 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 $\mathtt{or}(Q_1, \ldots, Q_p)$. By Definition 4, for some $i \in \{1, \ldots, p\}$, θ is an answer Q_i and Z. By the inductive assumption, 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 p be a query rule, where for each targeted query term $\mathtt{in}(r, q)$ in p there is a type name $T = type(r)$ 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 p and Z exists then there exist s and D' such that $D' \supseteq D$ and $D' \vdash p : U \triangleright s$.

If $D \vdash p : U \triangleright s$ and d is a result for p and Z, then $d \in [\![s]\!]$.

Proof. Let $p = c \leftarrow Q$. Assume that there exists a result for $(c \leftarrow Q)$ and Z. Let Θ be the set of all answers for Q and Z. By Definition 8, $\Theta \neq \emptyset$. Let $\theta \in \Theta$. By Proposition 7, $D \vdash Q : U \rhd \Gamma_\theta$. Thus, by Lemma 26 of [12], there is a set Ψ of variable type mappings that Ψ is complete for Q and U.

Let $\{\Psi_1, \dots, \Psi_n\} = \Psi/_{\overset{*}{\sim}_{FV(c)}}$. By induction from Lemma 27 of [12], we obtain that there exist $D_n \supseteq \cdots \supseteq D_1 \supseteq D$ and $s_1, \dots, s_n$ such that $D_i \vdash c : \Psi_i \rhd s_i$ (and by Lemma 28 of [12], $D_n \vdash c : \Psi_i \rhd s_i$) for each $i \in \{1, \dots, n\}$. By Lemma 29 of [12], Ψ is also complete for Q and U wrt. D_n. By rule (Query Rule), we obtain $D_n \vdash (c \leftarrow Q) : U \rhd s_1 \mid \cdots \mid s_n$.

Now assume that there exists s and D such that $D \vdash c \leftarrow Q : U \rhd s$. Let d' be a result for $(c \leftarrow Q)$ and Z. Let Θ be the set of all answers for Q and Z and let $\theta \in \Theta$. By Proposition 7, $D \vdash Q : U \rhd \Gamma_\theta$. Since Ψ used in the rule (Query Rule) is complete for Q and U wrt. D, there exists $\Gamma \in \Psi$ such that $\Gamma_\theta \sqsubseteq \Gamma$. Since $\theta \in \mathit{substitutions}(\Gamma_\theta)$, we obtain $\theta \in \mathit{substitutions}(\Psi)$. Thus $\Theta \subseteq \mathit{substitutions}(\Psi)$.

We have $d' = \Theta'(c)$ for some $\Theta' \in \Theta/_{\simeq_{FV(c)}}$. By Lemma 30 of [12], there exists $\Psi' \in \Psi/_{\overset{*}{\sim}_{FV(c)}}$ such that $\Theta' \subseteq \mathit{substitutions}(\Psi')$. Since $D \vdash (c \leftarrow Q) : U \rhd s_1 \mid \cdots \mid s_n$, then for some $i \in \{1, \dots, n\}$, we have $\Psi_i = \Psi'$ and $D \vdash c : \Psi_i \rhd s_i$. By Proposition 31 of [12], $d = \Theta'(c) \in [\![s_i]\!] \subseteq [\![s_1 \mid \cdots \mid s_n]\!]$. □

A.1.2 Type Inference for Programs

Here we provide proofs for Theorems 2, 3 and Proposition 2. The proofs in this section use properties of *resType* expressed by Lemmata 1, 2. We omit their formal proofs.

Lemma 1. *Let p be a query rule and U, U' be sets of type names from a Type Definition D. Let the type of each external resource occurring in p be defined by D. Provided that*

- *there are no grouping constructs in p,*
- *if the head of p contains a construct term $l\{c_1, \dots, c_n\}$ then $c_1, \dots, c_n$ are rooted construct terms,*
- *all multiplicity lists occurring in D are intersectable,*

if $[\![U]\!] \subseteq [\![U']\!]$ then $[\![resType(p, U)]\!] \subseteq [\![resType(p, U')]\!]$.

If D from the lemma above contains non intersectable multiplicity lists, computation of *resType*(p, U) may require finding intersectable multiplicity lists approximating non intersectable ones (see the typing rule (Var Approx)). For this an algorithm for computing a multiplicity list approximating a union of multiplicity lists is employed. This operation does not retain monotonicity and as a consequence *resType* is not monotonic for this case.

In rule (Pattern) for construct terms, approximating $s_1 \cdots s_n$ by a multiplicity list is not monotonic in general. The condition on $c_1, \ldots, c_n$ above implies that a permutation of $s_1 \cdots s_n$ is a multiplicity list, and $s_1 \cdots s_n$ is approximated by this permutation. Also, *resType* may be not monotonic wrt. its second argument if p contains grouping constructs.

Lemma 2. *Let p, p' be a pair of rules, U, U'' be sets of type names and $U' = resType(p', U'')$. If $p \not\leadsto_w p'$ then $[\![resType(p, U \cup U')]\!] = [\![resType(p, U)]\!]$.*

Corollary 1 (Monotonicity of T_P)**.** *Let P be a set of rules and U, U' be sets of type names from a Type Definition D. Let the type of each external resource occurring in rules from P be defined by D. Assume that*

- *there are no grouping constructs in the rules of P,*
- *if the head of a rule from P contains a construct term $l\{c_1, \ldots, c_n\}$ then $c_1, \ldots, c_n$ are rooted construct terms with distinct labels,*
- *all multiplicity lists occurring in D are intersectable.*

Let D' be the obtained Type Definition specifying the type names in the sets $T_P(U), T_P(U')$. Then

- *all the multiplicity lists in D' are intersectable,*
- *$[\![U]\!]_D \subseteq [\![U']\!]_D$ implies $[\![T_P(U)]\!]_{D'} \subseteq [\![T_P(U')]\!]_{D'}$.*

Proof. For each construct term $l\{c_1, \ldots, c_n\}$ in P the labels of $c_1, \ldots, c_n$ are distinct. Thus whenever rule (Pattern) for construct terms is applied, $s_1, \ldots, s_n$ are type variables with distinct labels, and the obtained multiplicity list $s_1 \cdots s_n$ is intersectable. The last conclusion follows from Lemma 1. □

Lemma 3. *Let P, P' be sets of rules such that $P' \subseteq P$. Let $U = T_P^i(\emptyset)$ for some $i > 0$ and Z be a set of data terms. Assume that T_P is monotonic. If $Z \subseteq [\![U]\!]$ then $R_{P'}^j(Z) \subseteq [\![T_P^j(U)]\!]$ for each $j = 0, 1, \ldots$.*

Proof. By Theorem 1, for each rule $p \in P'$, $res(p, Z) \subseteq [\![resType(p, U)]\!] \subseteq [\![\bigcup_{p\in P} resType(p, U)]\!] = [\![T_P(U)]\!]$. This implies that $\bigcup_{p\in P'} res(p, Z) \subseteq [\![T_P(U)]\!]$. Also $Z \subseteq [\![U]\!] \subseteq [\![T_P(U)]\!]$, by Corollary 1. Thus $R_{P'}(Z) = Z \cup \bigcup_{p\in P'} res(p, Z) \subseteq [\![T_P(U)]\!]$. Hence $R_{P'}^j(Z) \subseteq [\![T_P^j(U)]\!]$, by induction on j. □

Proposition 8. *Let $\mathcal{P} = (P', G)$ be an Xcerpt program, $P_1, \ldots, P_n, G$ be a stratification of $\mathcal{P}$ and $P = P' \backslash G$. Let $Z_0 = \emptyset$ and, for $j = 1, \ldots, n$, let $Z_j = R_{P_j}^{l_j}(Z_{j-1})$ for such $l_j > 0$ that $R_{P_j}^{l_j}(Z_{j-1}) = R_{P_j}^{l_j+1}(Z_{j-1})$. Assume that T_P is monotonic. Then, for $j = 1, \ldots, n$,*

$$Z_j \subseteq [\![T_P^{k_j}(\emptyset)]\!] \qquad \textit{where} \qquad k_j = \sum_{m=1}^{j} l_m.$$

Proof. By induction on j. For $j = 0$ the conclusion trivially holds. Assume $Z_{j-1} \subseteq [\![T_P^{k_{j-1}}(\emptyset)]\!]$. By Lemma 3, $R_{P_j}^{l_j}(Z_{j-1}) \subseteq [\![T_P^{l_j}(\,T_P^{k_{j-1}}(\emptyset)\,)]\!] = [\![T_P^{l_j+k_{j-1}}(\emptyset)\,)]\!] = [\![T_P^{k_j}(\emptyset)\,)]\!]$. So $Z_j \subseteq [\![T_P^{k_j}(\emptyset)]\!]$. □

Theorem 2. *Let $\mathcal{P} = (P', G)$ be an Xcerpt program and $P = P' \backslash G$. Assume that T_P is monotonic. If d is a result of a rule p in P' then there exists $i > 0$ such that*

$$d \in [\![resType(p, T_P^i(\emptyset))]\!] \subseteq [\![T_{P'}(T_P^i(\emptyset))]\!].$$

If $[\![T_P^{j+1}(\emptyset)]\!] = [\![T_P^j(\emptyset)]\!]$ for some $j > 0$ then the above holds for $i = j$.

Proof. Let $P_1, \ldots, P_n, G$ be a stratification of $\mathcal{P}$. Let $Z_0 = \emptyset$ and for $j = 1, \ldots, n$, $Z_j = R_{P_j}^{l_j}(Z_{j-1})$ for such $l_j > 0$ that $R_{P_j}^{l_j}(Z_{j-1}) = R_{P_j}^{l_j+1}(Z_{j-1})$.

By Proposition 8, $Z_n \subseteq [\![T_P^{k_n}(\emptyset)]\!]$, where $k_n = \sum_{m=1}^{n} l_m$. Let $i = k_n$. By Definition 15 and Theorem 1, $d \in res(p, Z_n) \subseteq [\![resType(p, T_P^{k_n}(\emptyset))]\!]$.

By Definition 21, $[\![resType(p, U)]\!] \subseteq [\![T_{P'}(U)]\!]$ for any U. Thus the first conclusion of the Theorem holds.

If $[\![T_P^{j+1}(\emptyset)]\!] = [\![T_P^j(\emptyset)]\!]$ then $[\![T_P^i(\emptyset)]\!] \subseteq [\![T_P^j(\emptyset)]\!]$ for any $i \geq 0$. Thus $d \in [\![resType(p, T_P^i(\emptyset))]\!] \subseteq [\![resType(p, T_P^j(\emptyset))]\!]$ and the conclusion holds with i replaced by j. □

Proposition 2. *Let P be a set of rules and $n > 0$. If $[\![T_P^{n-1}(\emptyset)]\!] \neq [\![T_P^n(\emptyset)]\!]$ then there exist $p_1, \ldots, p_n \in P$ such that $p_n \succ_w \cdots \succ_w p_1$.*

Proof. For $n = 1$ the proposition holds trivially, as $\emptyset \neq [\![T_P(\emptyset)]\!]$ implies that P is nonempty. So assume that $n > 1$. Let $U^i = T_P^i(\emptyset)$. We have $T_P(U) = \bigcup_{p \in P} resType(p, U)$ by the definition of T_P, and

$$resType(p, U^i) = resType\Big(\,p, \bigcup_{\substack{p' \in P \\ p \succ_w p'}} resType(p', U^{i-1})\Big) \qquad \text{(A.1)}$$

for $i > 0$, by Lemma 2. From $[\![U^{n-1}]\!] \neq [\![U^n]\!]$ it follows that $T_P(U^{n-2}) \neq T_P(U^{n-1})$ and then $resType(p_n, U^{n-2}) \neq resType(p_n, U^{n-1})$ for some $p_n \in P$.

By (A.1), if $resType(p, U^{i-1}) \neq resType(p, U^i)$ then there exists a rule $p' \in P$ such that $p \succ_w p'$ and if $i > 1$ then $resType(p', U^{i-2}) \neq resType(p', U^{i-1})$. From this by induction we obtain that if $resType(p_n, U^{n-2}) \neq resType(p_n, U^{n-1})$ then there exist $p_1, \ldots, p_n \in P$ such that $p_n \succ_w \cdots \succ_w p_1$. □

Theorem 3. *Let $\mathcal{P} = (P', G)$ be an Xcerpt program, $P = P' \backslash G$, and $P_0 \subseteq P$ such that $P \setminus P_0$ is not w-recursive. Assume that T_P is monotonic. Let W be a set of type names, and let $\widehat{T}_P(U) = T_{P \setminus P_0}(U) \cup W$ for any set U of type* *names. Let $U^\infty = \widehat{T}_P^k(\emptyset)$ be a fixed point of $\widehat{T}_P$ (i.e. $[\![\widehat{T}_P(U^\infty)]\!] = [\![U^\infty]\!]$). If $[\![resType(p, U^\infty)]\!] \subseteq [\![W]\!]$ for each $p \in P_0$ then*

$$\begin{array}{ll} d \in [\![resType(p, U^\infty)]\!] \subseteq [\![U^\infty]\!] & \textit{for any result } d \textit{ of a rule } p \in P, \\ d \in [\![resType(p, U^\infty)]\!] \subseteq [\![T_{P'}(U^\infty)]\!] & \textit{for any result } d \textit{ of a rule } p \in P', \\ [\![T_P(U^\infty)]\!] \subseteq [\![U^\infty]\!] \quad \textit{and} \quad [\![T_P^j(\emptyset)]\!] \subseteq [\![U^\infty]\!] \textit{ for any } j > 0. & \end{array}$$

Moreover, U^∞ in the last three lines may be replaced by $T_P^j(U^\infty)$, for any $j > 0$.

Proof. Notice that $[\![T_{P\setminus P_0}(U^\infty)]\!] \subseteq [\![U^\infty]\!]$, as $[\![T_{P\setminus P_0}(U^\infty)]\!] \cup [\![W]\!] \subseteq [\![U^\infty]\!]$. Notice also that $[\![resType(p, U^\infty)]\!] \subseteq [\![U^\infty]\!]$ for $p \in P_0$. Hence $[\![T_P(U^\infty)]\!] \subseteq [\![U^\infty]\!]$, as $T_P(U) = T_{P\setminus P_0}(U) \cup \bigcup_{p\in P_0} [\![resType(p, U)]\!]$. From monotonicity of T_P we obtain by induction that $[\![U^\infty]\!] \supseteq [\![T_P(U^\infty)]\!] \supseteq [\![T_P^2(U^\infty)]\!] \supseteq \cdots$, and that $[\![T_P^i(U^\infty)]\!] \supseteq [\![T_P^i(\emptyset)]\!]$ for $i \geq 0$. Hence $[\![T_P^i(\emptyset)]\!] \subseteq [\![U^\infty]\!]$ for each $i \geq 0$. Thus, by Theorem 2 any result of a rule p of P' is in $[\![resType(p, U^\infty)]\!]$. The latter is a subset of $[\![T_{P'}(U^\infty)]\!]$ and a subset of $[\![U^\infty]\!]$ if $p \in P$. □

A.2 Exactness of Inferred Type

Proposition 3. *Let D be a Type Definition without nullable type names, and whose content models do not contain useless type names. Let q be a query term, T a type name from D, and $\Theta = \{\, \theta \,|\, D \vdash q : T \triangleright \Gamma, \theta \in substitutions_D(\Gamma) \,\}$. If q does not contain $\rightsquigarrow$ then each $\theta \in \Theta$ is an answer for q and some $d \in [\![T]\!]_D$.*

Proof. Notice, that for any type name T occurring in D, $[\![T]\!]_D \neq \emptyset$ (as D does not contain nullable type names). Assume that $D \vdash q : T \triangleright \Gamma$ and $\theta \in substitutions(\Gamma)$. We will show that θ is an answer for q and some $d \in [\![T]\!]$. By induction on the derivation tree of $D \vdash q : T \triangleright \Gamma$.

- If q is a basic constant then an arbitrary substitution θ 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) $[\![\Gamma(X)]\!] \subseteq [\![T]\!]$. As $\theta \in substitutions_D(\Gamma)$ we obtain $X\theta \in [\![\Gamma(X)]\!]$. Hence $X\theta \in [\![T]\!]$. Thus, θ is an answer for X and $d = X\theta$.

- Let q be of the form $l[q_1, \cdots, q_n]$ and the rule for T in D be of the form $T \to l[r]$. 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 $T_1 \cdots T_n \in L(r)$.

  As $[\![T_i]\!] \neq \emptyset$, by induction hypothesis there exist data terms $d_i \in [\![T_i]\!]$ $(i = 1, \ldots, n)$ such that θ is an answer for each q_i and d_i. By Definition 3, θ is an answer for q and $l[\, d_1, \ldots, d_n \,] \in [\![T]\!]$.

- Let q be of the form $l\{q_1, \cdots, q_n\}$ and the rule for T in D be of the form $T \to l[r]$. 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 $T_1 \cdots T_n \in \mathit{perm}(L(r))$. As $[\![T_i]\!] \neq \emptyset$, by induction hypothesis there exist data terms $d_i \in [\![T_i]\!]$ $(i = 1, \ldots, n)$ such that θ is an answer for each q_i and d_i. Let $t_1, \ldots, t_n$ be a permutation of $d_1, \cdots, d_n$ such that $l[t_1, \ldots, t_m] \in [\![T]\!]$. By Definition 3, θ is an answer for q and $l[t_1, \ldots, t_n] \in [\![T]\!]$.

- Let q be of the form $l[[q_1, \cdots, q_n]]$ and the rule for T in D be of the form $T \to l[r]$. 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 $T_1 \cdots T_n \in L(s)$, where s is r with every type name U replaced by $U|\epsilon$. As $[\![T_i]\!] \neq \emptyset$, by induction hypothesis there exist data terms $d_i \in [\![T_i]\!]$ $(i = 1, \ldots, n)$ such that θ is an answer for each q_i and d_i. Let $t_1, \ldots, t_m$ be a sequence of data terms containing subsequence $d_1, \cdots, d_n$ such that $l[t_1, \ldots, t_m] \in [\![T]\!]$. By Definition 3, θ is an answer for q and $l[\, t_1, \ldots, t_m \,] \in [\![T]\!]$.

- Let q be of the form $l\{\{q_1, \cdots, q_n\}\}$ and the rule for T in D be of the form $T \to l[r]$. 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 $T_1 \cdots T_n \in \mathit{perm}(L(s))$, where s is r with every type name U replaced by $U|\epsilon$. As $[\![T_i]\!] \neq \emptyset$, by an induction hypothesis there exist data terms $d_i \in [\![T_i]\!]$ $(i = 1, \ldots, n)$ such that θ is an answer for each q_i and d_i. Let $t_1, \ldots, t_m$ be a sequence of data terms containing subsequence being a permutation of $d_1, \cdots, d_n$ such that $l[t_1, \ldots, t_m] \in [\![T]\!]$. By Definition 3, θ is an answer for q and $l[\, t_1, \ldots, t_m \,] \in [\![T]\!]$.

- Let q be of the form $l\{q_1, \cdots, q_n\}$ and the rule for T in D be of the form $T \to l\{r\}$. 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 $T_1 \cdots T_n \in \mathit{perm}(L(r))$. As $[\![T_i]\!] \neq \emptyset$, by induction hypothesis there exist data terms $d_i \in [\![T_i]\!]$ $(i = 1, \ldots, n)$ such that θ is an answer for each q_i and d_i. By Definition 3, θ is an answer for q and $l\{d_1, \ldots, d_n\} \in [\![T]\!]$.

- Let q be of the form $l\{\{q_1, \cdots, q_n\}\}$ and the rule for T in D be of the form $T \to l\{r\}$. 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 $T_1 \cdots T_n \in \mathit{perm}(L(s))$, where s is r with every type name U replaced by $U|\epsilon$. As $[\![T_i]\!] \neq \emptyset$, by induction hypothesis there exist data terms $d_i \in [\![T_i]\!]$ $(i = 1, \ldots, n)$ such that θ is an answer for each q_i and d_i. Let $t_1, \ldots, t_m$ be a sequence of data terms containing subsequence $d_1, \cdots, d_n$ such that $l\{t_1, \ldots, t_m\} \in [\![T]\!]$. By Definition 3, θ is an answer for q and $l\{\, t_1, \ldots, t_m \,\} \in [\![T]\!]$.

- Let q be of the form `desc` q'. Given $D \vdash q : T \triangleright \Gamma$ a premise of the rule (Descendant) or of the rule (Descendant Rec) must hold.

First, assume that the premise of (Descendant) holds i.e. $D \vdash q' : T \triangleright \Gamma$. As $[\![T]\!] \neq \emptyset$, by induction hypothesis there exist a data term $d \in [\![T]\!]$ such that θ is an answer for q' and d. By Definition 3, θ is an answer for q and d.

Now, assume that the premise of (Descendant Rec) holds, i.e. $D \vdash \texttt{desc}\, q : T' \triangleright \Gamma$, for $T' \in \mathit{types}(r)$, where r is the content model of T. As $[\![T']\!] \neq \emptyset$, by induction hypothesis there exists a data term $d \in [\![T']\!]$ such that θ is an answer for q and d. As T' is not a useless type name, $T' \in \mathit{types}(r)$ and $[\![T]\!] \neq \emptyset$ there exists a data term $d' \in [\![T]\!]$ such that d is a subterm of d'. By the Definition 3, θ is an answer for q and d'.

□

Proposition 4. *Let D be a Type Definition without nullable type names, and whose content models do not contain useless type names. Let U be a set of type names from D, Q be a query and $\Theta = \{\, \theta \,|\, D \vdash Q : U \triangleright \Gamma,\, \theta \in \mathit{substitutions}_D(\Gamma) \,\}$. Let $T_1, \ldots, T_n$ be type names in D such that $\mathit{type}(r_i) = T_i$ for each targeted query term $\texttt{in}(r_i, q_i)$ in Q $(i = 1, \ldots, n)$. If Q does not contain $\rightsquigarrow$ and multiple occurrences of the same resource (as an argument of a construct $\texttt{in}(\ldots)$) under the scope of a construct $\texttt{and}(\ldots)$ then for each $\theta \in \Theta$ there exist*

- *data terms $d_1, \ldots, d_n$ of types $T_1, \ldots, T_n$, respectively,*
- *a set $Z \subseteq [\![U]\!]_D$ of data terms*

such that θ is an answer for Q' and Z, where Q' is Q with each targeted query term $\texttt{in}(r_i, q_i)$ replaced by a targeted query term $\texttt{in}(r'_i, q_i)$, such that $\delta(r'_i) = d_i$.

Proof. We assume that $D \vdash Q : U \triangleright \Gamma$ and $\theta \in \mathit{substitutions}_D(\Gamma)$. Let $T_1, \ldots, T_n$ be type names such that $\mathit{type}(r_i) = T_i$ for each targeted query term $\texttt{in}(r_i, q_i)$ 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 3, there exists a data term $d \in [\![T]\!]$ such that θ is an answer substitution for Q' and d. Hence, there exists $Z = \{d\} \subseteq [\![U]\!]$ such that θ is an answer substitution for Q' and Z.

  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 $\texttt{in}(r_i, q_i)$. By the query typing rule (Targeted Query Term), $D \vdash Q : U \triangleright \Gamma$ implies $D \vdash q_i : T_i \triangleright \Gamma$

for $T_i = type(r_i)$. As $D \vdash q_i : T_i \rhd \Gamma$, by Proposition 3, there exists $d_i \in [\![T_i]\!]$ such that θ is an answer for q_i and d_i. Let Q' be $\texttt{in}(r'_i, q_i)$, where $\delta(r'_i) = d_i$.

Let $Z \subseteq [\![U]\!]$ be a set of data terms. By Definition 4, θ is an answer for Q' and Z.

- Let Q be of the form $\texttt{or}(Q_1, \ldots, Q_n)$. By the typing rule (Or Query) $D \vdash Q : U \rhd \Gamma$ implies $D \vdash Q_j : U \rhd \Gamma$ for some j $(1 \leq j \leq n)$. By induction hypothesis there exist

  - data terms $d_1, \ldots, d_n$ of types $T_1, \ldots, T_n$, respectively,
  - a set of data terms $Z \subseteq [\![U]\!]_D$

  such that θ is an answer for some Q'_j and Z, where Q'_j is Q_j with each targeted query term $\texttt{in}(r_p, q_p)$ replaced by a targeted query term $\texttt{in}(r'_p, q_p)$, such that $\delta(r'_p) = d_p$. Let $Q' = \texttt{or}(Q_1, \ldots, Q'_j, \ldots, Q_n)$. By Definition 4, θ is an answer for Q' and Z.

- Let Q be of the form $\texttt{and}(Q_1, \ldots, Q_n)$. By the typing rule (And Query) $D \vdash Q : U \rhd \Gamma$ implies $D \vdash Q_j : U \rhd \Gamma$ for $j = 1, \ldots, n$. Thus for each $\theta \in \Theta$, $\theta \in \{\, \theta \mid D \vdash Q_j : U \rhd \Gamma,\, \theta \in substitutions_D(\Gamma) \,\}$ for $j = 1, \ldots, n$. By induction hypothesis there exist

  - data terms $d_1, \ldots, d_n$ of types $T_1, \ldots, T_n$, respectively,
  - sets of data terms $Z_1, \ldots, Z_n$

  such that for $j = 1, \ldots, n$, $Z_j \subseteq [\![U]\!]_D$ and θ is an answer for Q'_j and Z_j, where each Q'_j is Q_j with each targeted query term $\texttt{in}(r_p, q_p)$ replaced by a targeted query term $\texttt{in}(r'_p, q_p)$, such that $\delta(r'_p) = d_p$. Let $Q' = \texttt{and}(Q'_1, \ldots, Q'_n)$. By Definition 4, θ is an answer for Q' and $Z = Z_1 \cup \ldots \cup Z_n$.

□

A.3 Soundness of DigXcerpt Implementation

The section presents a proof of Theorem 4 which expresses soundness of the implementation algorithm for DigXcerpt described in Section 7.2.2. First we introduce a notation used in this section.

Let d be a data term of the form $id[dig[a_1, c_1], \ldots, dig[a_m, c_m]\,]$, where id is one of the unique labels introduced by translation of a DigXcerpt program into an Xcerpt program. The data term d contains DIG ask statements $a_1, \ldots, a_m$. The corresponding set of reasoner responses, denoted as $RR(d)$, is $\{\, id[r_1, c_1], \ldots, id[r_m, c_m] \,\}$, where $r_1, \ldots, r_n$ are the reasoner responses (DIG response statements) for $a_1, \ldots, a_n$, respectively. The data terms of the form $id[r_i, c_i]$ are called DIG response terms. For a set of data terms Z, $RR(Z) = \bigcup_{d \in Z} RR(d)$.

For a set of rules P, $res(P, Z)$ is defined as $\bigcup_{p \in P} res(p, Z)$ and $R_P(Z)$ is defined as in Definition 10: $R_P(Z) = Z \cup res(P, Z)$. The set $R^k_P(\emptyset)$ such that $R^k_P(\emptyset) = R^{k+1}_P(\emptyset)$ (for some $k \geq 0$) will be denoted as $R^\infty_P(\emptyset)$.

For a set of rules P, a (length m) *computation* of P is a sequence: $Z_0, p_1, Z_1, \ldots, Z_{m-1}, p_m, Z_m$, where $Z_0 = \emptyset$, $Z_j = Z_{j-1} \cup res(p_j, Z_{j-1})$ and $p_j \in P$, for $j = 1, \ldots, m$. Notice that the sets $Z_0, \ldots, Z_m$ are finite, and that $Z_j \subseteq R^j_P(\emptyset)$ for $j = 0, 1, \ldots$, provided that there are no grouping constructs in the rules from P. (The latter is due to $res(p, Z) \subseteq R_P(Z) \subseteq R_P(Z')$ for any $p \in P$ and $Z \subseteq Z'$, by Lemma 5.) Sometimes the computation will be abbreviated as $Z_0, P_1, Z_1, \ldots, Z_{m'-1}, P_{m'}, Z_{m'}$, where each $P_i \subseteq P$ is a set of rules not pairwise dependent (thus the order of execution of rules from P_i is irrelevant) and $Z_0 = \emptyset$, $Z_i = res(P_i, Z_{i-1})$, for $i = 1, \ldots, m'$.

Given a computation $\emptyset, \ldots, Z$, the set Z is called the result of the computation. A computation $\emptyset, \ldots, Z$ of P is called final if $Z = R^\infty_P(\emptyset)$. Thus an existence of a final computation of P guarantees that there is no infinite loop in P and that $R^\infty_P(\emptyset)$ exists. Also existence of $R^\infty_P(\emptyset)$ guarantees that a final computation of P exists.

We will use the fact that $res(p, \cdot)$ is monotone for any rule p without grouping constructs:

Lemma 4. *Let p be a DigXcerpt rule and Z, Z' be sets of data terms such that $Z \subseteq Z'$. If there is no grouping constructs in p then $res(p, Z) \subseteq res(p, Z')$.*

As its consequence we obtain:

Lemma 5. *Let P be a set of query rules without grouping constructs and let Z, Z' be sets of data terms. If $Z \subseteq Z'$ then $R_P(Z) \subseteq R_P(Z')$, $R^j_P(Z) \subseteq R^j_P(Z')$, and $R^j_P(Z) \subseteq R^k_P(Z)$ for $0 \leq j \leq k$.*

Proposition 9. *Let P be a set of query rules without grouping constructs and $\emptyset, \ldots, Z$, be a computation of P. If $res(p, Z) \subseteq Z$ for each $p \in P$ then $Z = R^\infty_P(\emptyset)$.*

Proof. As $res(p, Z) \subseteq Z$ for each $p \in P$, $\bigcup_{p \in P} res(p, Z) \subseteq Z$. Thus $R_P(Z) = Z \cup \bigcup_{p \in P} res(p, Z) \subseteq Z$. Hence by monotonicity of R^i_P (Lemma 5), $R^{i+1}_P(Z) \subseteq R^i_P(Z)$ for $i \geq 0$ and $R^l_P(Z) \subseteq Z$ for any $l \geq 0$. As Z is finite, $R^\infty_P(\emptyset)$ exists. Thus $R^\infty_P(\emptyset) \subseteq Z$ and by Lemma 5, $Z \subseteq R^\infty_P(\emptyset)$. Hence $Z = R^\infty_P(\emptyset)$.

□

In what follows we will use the following Lemmata; rather obvious proofs of the first four of them we skip.

Lemma 6. *Let Z be a set of data terms, e be an extended DigXcerpt rule and dg, dr be the corresponding DIG ask goal and DIG response rules. Then $res(e, Z) = res(dr, RR(res(dg, Z)))$.*

Lemma 7. *Let dg be a DIG goal rule without grouping constructs in* digAskConstruct *and Z, Z' be sets of data terms. If $Z \subseteq Z'$ then $RR(res(dg, Z)) \subseteq RR(res(dg, Z'))$.*

Lemma 8. *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program and $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and Z be a set of data terms. Then $res(dr_i, RR(res(E, Z))) = res(dr_i, RR(res(dg_i, Z)))$.*

Lemma 9. *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program and $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and Z, Z' be sets of data terms. Let $R = RR(res(E, Z)))$ and $p \in P$. Then $res(p, Z' \cup R) = res(p, Z')$ and $res(E, Z' \cup R) = res(E, Z')$. For Z'' being a set of data terms without DIG response terms $res(dr_i, Z'' \cup R) = res(dr_i, R)$.*

Lemma 10. *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without grouping constructs. Let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P' \backslash G'$. Let $\mathcal{P}^0, \ldots, \mathcal{P}^{k+1}$ be a sequence of Xcerpt programs such that for $j = 0, \ldots, k+1$,*

- *$\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, if $j \leq k$ then there exists a final computation for P^j, and $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$,*
- *$R^{k+1} = R^k$.*

Let $S = \emptyset, p_1, Z_1, p_2, Z_2, \ldots, p_m, Z_m$ be a computation of $P \backslash G$. There exists a computation $S' = \emptyset, q_1, W_1, q_2, W_2, \ldots, q_{m'}, W_{m'}$ of P^k such that $Z_m \subseteq W_{m'}$.

Proof. Proof by induction on m. For $m = 0$, $S = S' = \emptyset$, thus $Z_m = W_{m'} = \emptyset$.

Induction step. Let $S = S_{-}, p_m, Z_m$ be a length m computation of $P \backslash G$, where $S_{-} = \emptyset, \ldots, Z_{m-1}$ is a computation of length $m-1$. By the inductive assumption, there exists a computation $S'_{-} = \emptyset, \ldots, W_l$ of P^k, such that $Z_{m-1} \subseteq W_l \subseteq R^{\infty}_{P^k}(\emptyset)$.

Let p_m be an Xcerpt rule p. Then $S' = S'_{-}, p, W_{m'}$. By Lemma 4, as $Z_{m-1} \subseteq W_l$, $Z_m = Z_{m-1} \cup res(p, Z_{m-1}) \subseteq W_l \cup res(p, W_l) = W_{m'}$.

Let p_m be an extended rule e and dr and dg be a DIG response rule and

DIG ask goal, respectively, corresponding to e.

$$\begin{array}{ll}
res(e, Z_{m-1}) & \\
= res(dr, RR(res(dg, Z_{m-1}))) & \text{by Lemma 6,} \\
\subseteq res(dr, RR(res(dg, R^{\infty}_{P^k}(\emptyset)))) & \text{by Lemmata 4, 7, as } Z_{m-1} \subseteq R^{\infty}_{P^k}(\emptyset), \\
\subseteq res(dr, RR(res(E, R^{\infty}_{P^k}(\emptyset)))) & \text{by Lemma 4} \\
 & \text{and the definition of } res(E, Z), \\
= res(dr, R^{k+1}) & \text{by the definition of } R^{k+1}, \\
= res(dr, R^k) & \text{as } R^{k+1} = R^k, \\
\subseteq res(dr, W_l \cup R^k) & \text{by Lemma 4.}
\end{array}$$

We construct $S' = S'_{\_}, r(R^k), W_l \cup R^k, dr, W_{m'}$, where $W_{m'} = W_l \cup R^k \cup res(dr, R^k \cup W_l)$.

The result of S is $Z_m = Z_{m-1} \cup res(e, Z_{m-1})$. As $Z_{m-1} \subseteq W_l$ and $res(e, Z_{m-1}) \subseteq res(dr, W_l \cup R^k)$, we have $Z_m \subseteq W_l \cup res(dr, W_l \cup R^k) \subseteq W_l \cup R^k \cup res(dr, W_l \cup R^k) = W_{m'}$.

□

Corollary 2. *$R^{\infty}_{P\backslash G}(\emptyset) \subseteq R^{\infty}_{P^k}(\emptyset)\backslash R^k$.*

Proof. Let S be a computation such that $Z_m = R^{\infty}_{P\backslash G}(\emptyset)$. By Lemma 10, we have $Z_m \subseteq W_{m'} \subseteq R^{\infty}_{P^k}(\emptyset)$. Thus $R^{\infty}_{P\backslash G}(\emptyset) \subseteq R^{\infty}_{P^k}(\emptyset)$. As the labels from R^k are unique identifiers $R^{\infty}_{P\backslash G}(\emptyset) = R^{\infty}_{P\backslash G}(\emptyset)\backslash R^k$. Hence $R^{\infty}_{P\backslash G}(\emptyset) \subseteq R^{\infty}_{P^k}(\emptyset)\backslash R^k$.

□

Lemma 11. *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without grouping constructs and let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P'\backslash G'$. Let $\mathcal{P}^0, \ldots, \mathcal{P}^k$ ($k \geq 0$), be a sequence of Xcerpt programs such that for $j = 0, \ldots, k$,*

- *$\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, if $j < k$ then there exists a final computation for P^j, and $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$.*

Let $S = \emptyset, r(R^j), R^j, p_1, W_1, \ldots, p_m, W_m$ be a computation of P^j. There exists a computation $S' = \emptyset, \ldots, Z_{m'}$ of $P\backslash G$ such that $W_m\backslash R^k \subseteq Z_{m'}$.

Proof. Proof by induction on j.

$j = 0$.

Let $S = \emptyset, p_1, W_1, \ldots, p_m, W_m$ be a computation of P^0. As none of $W_0, \ldots, W_m$ contain DIG response terms, by Lemma 9, if p_i ($i = 1, \ldots, m$) is a DIG response rule then $W_{i-1} = W_i$. Thus, by removing DIG response rules from S we obtain a computation $S'' = \emptyset, \ldots, W_m$ of P^0 and of $P\backslash G$ with the same result as S. Thus $S' = S''$ and $W_m = Z_{m'}$. Hence $W_m\backslash R^k \subseteq Z_{m'}$.

$j > 0$.

Let $S_{j-1} = \emptyset, r(R^{j-1}), R^{j-1}, p_1, W'_1, \ldots, p_s, W'_s$ be a computation of P^{j-1} such that $W'_s = R^{\infty}_{P^{j-1}}(\emptyset)$. By the inductive assumption there exists a computation $S'_{j-1} = \emptyset, \ldots, Z'_{s'}$ of $P \backslash G$ such that $W'_s \backslash R^k \subseteq Z'_{s'}$.

Proof by induction on the length m of S. We construct a computation $S' = \emptyset, \ldots, Z_{m'}$ such that $W_m \backslash R^k \subseteq Z_{m'}$ and $R^{\infty}_{P^{j-1}}(\emptyset) \backslash R^k \subseteq Z_{m'}$. For $m = 0$, $S' = S_{j-1}$.

By the inductive assumption there exists a computation $S'_{-} = \emptyset, \ldots, Z_{m''}$ of $P \backslash G$ such that $W_{m-1} \backslash R^k \subseteq Z_{m''}$ and $R^{\infty}_{P^{j-1}}(\emptyset) \backslash R^k \subseteq Z_{m''}$.

Let p_m be not a DIG response rule. Then $W_m = W_{m-1} \cup res(p_m, W_{m-1}) = W_{m-1} \cup res(p_m, W_{m-1} \backslash R^k)$. Now $S' = S'_{-}, p_m, Z_{m'}$, where $Z_{m'} = Z_{m''} \cup res(p_m, Z_{m''})$. As $W_{m-1} \backslash R^k \subseteq Z_{m''}$, by Lemma 4, $W_m \backslash R^k \subseteq Z_{m'}$.

Let p_m be a DIG response rule dr and e be the corresponding extended rule from $P \backslash G$. Then $W_m = W_{m-1} \cup res(dr, W_{m-1})$. $S' = S'_{-}, e, Z_{m'}$ where $Z_{m'} = Z_{m''} \cup res(e, Z_{m''})$.

$$
\begin{array}{ll}
res(dr, W_{m-1}) & \\
= res(dr, R^j) & \text{by Lemma 9,} \\
= res(dr, RR(res(E, R^{\infty}_{P^{j-1}}(\emptyset)))) & \text{by the definition of } R^j, \\
= res(dr, RR(res(dg, R^{\infty}_{P^{j-1}}(\emptyset)))) & \text{by Lemma 8,} \\
= res(dr, RR(res(dg, R^{\infty}_{P^{j-1}}(\emptyset) \backslash R^k))) & \text{by Lemma 9,} \\
\subseteq res(dr, RR(res(dg, Z_{m''}))) & \text{by Lemmata 4, 7,} \\
 & \text{as } R^{\infty}_{P^{j-1}}(\emptyset) \backslash R^k \subseteq Z_{m''}, \\
= res(e, Z_{m''}) & \text{by Lemma 6.}
\end{array}
$$

Thus, by the inductive assumption, $W_m \backslash R^k \subseteq Z_{m'}$. □

Corollary 3. $R^{\infty}_{P \backslash G}(\emptyset) \supseteq R^{\infty}_{P^k}(\emptyset) \backslash R^k$.

Proof. Let S be a computation such that $W_m = R^{\infty}_{P^k}(\emptyset)$. As $W_m \backslash R^k \subseteq Z_{m'} \subseteq R^{\infty}_{P \backslash G}(\emptyset)$, $R^{\infty}_{P^k}(\emptyset) \backslash R^k \subseteq R^{\infty}_{P \backslash G}(\emptyset)$. □

Lemma 12. *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without grouping constructs such that there exists a final computation for $P \backslash G$. Let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P' \backslash G'$. Then*

1. *there exists a sequence of Xcerpt programs $\mathcal{P}^0, \mathcal{P}^1, \ldots$, such that, for $j \geq 0$, $\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$, and there exists a final computation for P^j,*

2. *$R^i \subseteq R^{i+1}$ for $i \geq 0$,*

3. *there exists $k \geq 0$ such that $R^k = R^{k+1}$.*

Proof. Let $S = \emptyset \ldots, W$ be a final computation of $P \backslash G$.

1. Notice that, for any $m > 0$, a sequence $\mathcal{P}^0, \mathcal{P}^1, \ldots, \mathcal{P}^m$ of programs exists iff there exist final computations for programs $\mathcal{P}^0, \mathcal{P}^1, \ldots, \mathcal{P}^{m-1}$. Proof by contradiction. Assume that there is no final computation for some P^j and take the first P^j for which there is no final computation. Thus, there exist final computations for the sets $P^0, P^1, \ldots, P^{j-1}$. As there is no final computation for P^j we can construct an infinite computation $\emptyset, \ldots, W'_t, \ldots, W'_{t+1}, \ldots$ such that $W'_t \subset W'_{t+1} \subset \ldots$. However, by Lemma 11, for any computation $S' = \emptyset, \ldots, W'$ of P^j, $W' \backslash R^j \subseteq W$. As W is finite we get a contradiction.

2. By induction: $\emptyset = R^0 \subseteq R^1$. If $R^j \subseteq R^{j+1}$ then $P^j \subseteq P^{j+1}$, hence $res(\mathcal{P}^j) \subseteq res(\mathcal{P}^{j+1})$ and $R^{j+1} \subseteq R^{j+2}$.

3. Proof by contradiction. Assume that $R^k \neq R^{k+1}$ for any $k \geq 0$. Thus $RR(res(\mathcal{P}^{k-1})) \neq RR(res(\mathcal{P}^k))$ for any $k \geq 1$ and then $res(\mathcal{P}^{k-1}) \neq res(\mathcal{P}^k)$. By the definition of $res(\mathcal{P}^k)$, $res(E, R^\infty_{P^{k-1}}(\emptyset)) \neq res(E, R^\infty_{P^k}(\emptyset))$. By Lemma 9, $res(E, R^\infty_{P^{k-1}}(\emptyset)) = res(E, R^\infty_{P^{k-1}}(\emptyset) \backslash R^k)$ and $res(E, R^\infty_{P^k}(\emptyset)) = res(E, R^\infty_{P^k}(\emptyset) \backslash R^k)$. Thus $R^\infty_{P^{k-1}}(\emptyset) \backslash R^k \neq R^\infty_{P^k}(\emptyset) \backslash R^k$.

   As $R^{k-1} \subseteq R^k$, $P^{k-1} \subseteq P^k$ and then $R^\infty_{P^{k-1}}(\emptyset) \subseteq R^\infty_{P^k}(\emptyset)$ (from Lemma 5 and the definitions of R_P and of $res(P, Z)$, by induction). Hence $R^\infty_{P^{k-1}}(\emptyset) \backslash R^k \subseteq R^\infty_{P^k}(\emptyset) \backslash R^k$ and, as $R^\infty_{P^{k-1}} \backslash R^k \neq R^\infty_{P^k}(\emptyset) \backslash R^k$, we have $R^\infty_{P^{k-1}}(\emptyset) \backslash R^k \subset R^\infty_{P^k}(\emptyset) \backslash R^k$. However, by Lemma 11, $R^\infty_{P^k}(\emptyset) \backslash R^k \subseteq W$ for any $k \geq 1$. So W is infinite. Contradiction. Hence there exists k such that $R^k = R^{k+1}$.

□

Theorem 4. *Let $\mathcal{P} = (P, G)$ be a DigXcerpt program without grouping constructs and such that there exists $R^\infty_{P \backslash G}(\emptyset)$. Let $\mathcal{P}' = (P', G')$ be $\mathcal{P}$ with each extended rule e_i replaced by the corresponding DIG response rule dr_i. Let $E = \{dg_1, \ldots, dg_n\}$ be the set of DIG ask goals corresponding to $e_1, \ldots, e_n$ and P'' be $P' \backslash G'$. Then there exists $k \geq 0$, a sequence of Xcerpt programs $\mathcal{P}^0, \ldots, \mathcal{P}^{k+1}$, and a sequence of sets $R^0, \ldots, R^{k+1}$ such that*

- *for $j = 0, \ldots, k+1$: $\mathcal{P}^j = (P^j \cup E, E)$, $P^j = P'' \cup r(R^j)$, $R^0 = \emptyset$, there exists $R^\infty_{P^j}(\emptyset)$, and $R^j = RR(res(\mathcal{P}^{j-1}))$ if $j > 0$,*
- *$R^k = R^{k+1}$.*

Let $\mathcal{Q} = (P^k \cup G', G')$. Then $R^\infty_{P \backslash G}(\emptyset) = R^\infty_{P^k}(\emptyset) \backslash R^k$ and $res(\mathcal{P}) = res(\mathcal{Q})$.

Proof. As there exists $R^\infty_{P \backslash G}(\emptyset)$ there exists a final computation for $P \backslash G$. By Lemma 12, there exist a sequence $P^0, \ldots, P^k$ and a sequence $R^0, \ldots, R^{k+1}$ such that $R^k = R^{k+1}$.

By Lemma 12, there exist final computations for $P^0, \ldots, P^k$. By Corollary 2, $R^\infty_{P \backslash G}(\emptyset) \subseteq R^\infty_{P^k}(\emptyset) \backslash R^k$. By Corollary 3, $R^\infty_{P \backslash G}(\emptyset) \supseteq R^\infty_{P^k}(\emptyset) \backslash R^k$. Hence $R^\infty_{P \backslash G}(\emptyset) = R^\infty_{P^k}(\emptyset) \backslash R^k$.

Let $Z = R^{\infty}_{P^k}(\emptyset) \backslash R^k = R^{\infty}_{P\backslash G}(\emptyset)$. As $R^k \subseteq R^{\infty}_{P^k}(\emptyset)$, $R^{\infty}_{P^k}(\emptyset) = Z \cup R^k$.

Let $A = \{eg_1, \ldots, eg_f\} \subseteq G$ be the set of those goals of G which are extended rules and let $B = G\backslash A$. Let $A' = \{dr_1, \ldots, dr_f\}$ be the set of DIG response rules (goals) corresponding to the rules from A and $A'' = \{dg'_1, \ldots, dg'_f\}$ be the set of DIG ask goals corresponding to the rules from A. We have $G = A \cup B$ and $G' = A' \cup B$.

$res(\mathcal{Q})$	
$= res(G', R^{\infty}_{P^k}(\emptyset))$	by the definition of $res(\mathcal{Q})$,
$= res(A' \cup B, Z \cup R^k)$	as $R^{\infty}_{P^k}(\emptyset) = Z \cup R^k$ and $G' = A' \cup B$,
$= res(A', Z \cup R^k) \cup res(B, Z \cup R^k)$	by the definition of $res(P, Z)$,
$= res(A', R^k) \cup res(B, Z)$	by Lemma 9,
$= res(A', R^{k+1}) \cup res(B, Z)$	as $R^{k+1} = R^k$,
$= res(A', RR(res(\mathcal{P}^k))) \cup res(B, Z)$	by the definition of R^{k+1},
$= res(A', RR(res(E, Z \cup R^k))) \cup res(B, Z)$	by the definition of $res(\mathcal{P}^k)$,
$= res(A', RR(res(E, Z))) \cup res(B, Z)$	by Lemma 9,
$= \bigcup_{i=1}^{f} res(dr_i, RR(res(dg'_i, Z))) \cup res(B, Z)$	by the definition of $res(P, Z)$ and Lemma 8,
$= \bigcup_{e \in A} res(e, Z) \cup res(B, Z)$	by Lemma 6,
$= res(A, Z) \cup res(B, Z)$	by the definition of $res(P, Z)$,
$= res(A \cup B, Z)$	by the definition of $res(P, Z)$,
$= res(G, Z)$	as $G = A \cup B$,
$= res(\mathcal{P})$	by the definition of $res(\mathcal{P})$.

$\square$

Appendix B

Typechecker Results

This chapter presents printouts from the typechecker prototype. The printouts are results of typing the program examples from Chapter 6. The way how the obtained results should be interpreted is explained in Chapter 5.

CDstore.1

```
===================================================================
Rule 1: pop-entries
-------------------------------------------------------------------
TITLE->Artist, ARTIST->Artist
TITLE->Title, ARTIST->Artist
===================================================================
===================================================================
Type Definition:
-------------------------------------------------------------------
pop-entries -> pop-entries[ entry+ ]
entry -> entry[ Artist (Artist|Title)+ ]
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> "pop" | "rock" | "classic"
===================================================================
```

CDstore.2

```
===================================================================
Rule 1: pop-entries
Type checking: Unsuccessful (results not of type Entries possible)
-------------------------------------------------------------------
TITLE->Artist, ARTIST->Artist
```

```
TITLE->Title, ARTIST->Artist
==================================================================
==================================================================
Type Definition:
------------------------------------------------------------------
pop-entries -> pop-entries[ entry+ ]
entry -> entry[ Artist (Artist|Title)+ ]
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> "pop" | "rock" | "classic"
Entries -> pop-entries[ Entry ]
Entry -> entry[ Artist Title+ ]
==================================================================
```

Bibliography.1

```
==================================================================
Rule 1: 0
------------------------------------------------------------------
0
==================================================================
==================================================================
Type Definition:
------------------------------------------------------------------
TextBook -> book[ Cover Body ]
Cover -> cover[ Title Author* Publisher? ]
Body -> body[ Abstract? Chapter* ]
Title -> title[ InlineContent ]
Author -> author[ Text ]
Publisher -> publisher[ Text ]
Abstract -> abstract[ Text ]
Chapter -> chapter[ Title Section* ]
InlineContent -> inline[ Text|Bf|Em ]
Section -> section[ Title (Paragraph|Table|List)* ]
Em -> em[ InlineContent ]
Bf -> bf[ InlineContent ]
Paragraph -> p[ InlineContent* ]
Table -> table[ TableRow+ ]
List -> list[ ListItem ]
TableRow -> tr[ TableCell* ]
ListItem -> item[ InlineContent* ]
TableCell -> td[ InlineContent* ]
Bibliography -> bib[ (Book|Article|InProceedings)* ]
Book -> book{ Publisher? Editors Authors Title1 }
Article -> article{ Journal? Authors Title1 }
InProceedings -> inproc{ Book Authors Title1 }
```

```
Title1 -> title[ Text ]
Authors -> authors[ Person* ]
Editors -> editors[ Person* ]
Journal -> journal{ Editors Title1 }
Person -> person[ FirstName LastName ]
FirstName -> first[ Text ]
LastName -> last[ Text ]
==================================================================
```

Bibliography.2

```
==================================================================
Rule 1: book
Type checking: Failed (no results of type TextBook)
------------------------------------------------------------------
* -> Top
==================================================================
==================================================================
Type Definition:
------------------------------------------------------------------
book -> book[ cover body ]
body -> body[ table ]
table -> table[ tr+ ]
tr -> tr[ td td_1 ]
td_1 -> td[ em+ ]
em -> em[ Top Top ]
td -> td[ Top ]
cover -> cover[ title ]
title -> title[ Text_1 ]
Text_1 -> "List_of_Books"
==================================================================
```

Bibliography.3

```
==================================================================
Rule 1: book
Type checking: Unsuccessful (results not of type TextBook possible)
------------------------------------------------------------------
TITLE->Text, NAME->Publisher
TITLE->Text, NAME->Text
TITLE->Text, NAME->Editors
TITLE->Text, NAME->Person
TITLE->Text, NAME->FirstName
TITLE->Text, NAME->LastName
TITLE->Text, NAME->Authors
==================================================================
==================================================================
Type Definition:
------------------------------------------------------------------
book -> book[ cover body ]
body -> body[ chapter ]
chapter -> chapter[ title_1 section ]
section -> section[ table ]
table -> table[ (tr|tr_1|tr_2|tr_3|tr_4|tr_5|tr_6)+ ]
tr_6 -> tr[ td td_13 ]
td_13 -> td[ inline_13 ]
inline_13 -> inline[ Authors ]
tr_5 -> tr[ td td_11 ]
td_11 -> td[ inline_11 ]
inline_11 -> inline[ LastName ]
tr_4 -> tr[ td td_9 ]
td_9 -> td[ inline_9 ]
inline_9 -> inline[ FirstName ]
tr_3 -> tr[ td td_7 ]
td_7 -> td[ inline_7 ]
inline_7 -> inline[ Person ]
tr_2 -> tr[ td td_5 ]
td_5 -> td[ inline_5 ]
inline_5 -> inline[ Editors ]
tr_1 -> tr[ td td ]
tr -> tr[ td td_1 ]
td_1 -> td[ inline_1 ]
inline_1 -> inline[ Publisher ]
td -> td[ inline ]
inline -> inline[ Text ]
title_1 -> title[ Text_2 ]
Text_2 -> "List_of_Books_and_Authors"
cover -> cover[ title ]
title -> title[ Text_1 ]
Text_1 -> "Books"
TextBook -> book[ Cover Body ]
Body -> body[ Abstract? Chapter* ]
Chapter -> chapter[ Title Section* ]
InlineContent -> inline[ Text|Bf|Em ]
Section -> section[ Title? (Paragraph|Table|List)* ]
Em -> em[ InlineContent ]
Bf -> bf[ InlineContent ]
Paragraph -> p[ InlineContent* ]
Table -> table[ TableRow+ ]
List -> list[ ListItem ]
TableRow -> tr[ TableCell* ]
ListItem -> item[ InlineContent* ]
TableCell -> td[ InlineContent* ]
TextBook_1 -> book[ Cover Body_1 ]
Cover -> cover[ Title Author* Publisher? ]
Body_1 -> body[ Abstract? Chapter_1* ]
```

```
Author -> author[ Text ]
Publisher -> publisher[ Text ]
Abstract -> abstract[ Text ]
Chapter_1 -> chapter[ Title Section_1* ]
InlineContent_1 -> inline[ Text|Bf_1|Em_1 ]
Section_1 -> section[ Title? (Paragraph_1|Table_1|List_1)* ]
Em_1 -> em[ InlineContent_1 ]
Bf_1 -> bf[ InlineContent_1 ]
Paragraph_1 -> p[ InlineContent_1* ]
Table_1 -> table[ TableRow_1+ ]
List_1 -> list[ ListItem_1 ]
TableRow_1 -> tr[ TableCell_1* ]
ListItem_1 -> item[ InlineContent_1* ]
TableCell_1 -> td[ InlineContent_1* ]
Bibliography -> bib[ (Book|Article|InProceedings)* ]
Book -> book{ Title Authors Editors Publisher? }
Article -> article{ Title Authors Journal? }
InProceedings -> inproc{ Title Authors Book }
Title -> title[ Text ]
Authors -> authors[ Person* ]
Editors -> editors[ Person* ]
Journal -> journal{ Title Editors }
Person -> person[ FirstName LastName ]
FirstName -> first[ Text ]
LastName -> last[ Text ]
==================================================================
```

Bibliography.4

```
==================================================================
Rule 1: book
Type checking: OK
------------------------------------------------------------------
TITLE->Text, NAME->Text
==================================================================
==================================================================
Type Definition:
------------------------------------------------------------------
book -> book[ cover body ]
body -> body[ chapter ]
chapter -> chapter[ title_1 section ]
section -> section[ table ]
table -> table[ tr+ ]
tr -> tr[ td td ]
td -> td[ inline ]
inline -> inline[ Text ]
title_1 -> title[ Text_2 ]
Text_2 -> "List_of_Books_and_Authors"
```

```
cover -> cover[ title ]
title -> title[ Text_1 ]
Text_1 -> "Books"
TextBook -> book[ Cover Body ]
Body -> body[ Abstract? Chapter* ]
Chapter -> chapter[ Title Section* ]
InlineContent -> inline[ Text|Bf|Em ]
Section -> section[ Title? (Paragraph|Table|List)* ]
Em -> em[ InlineContent ]
Bf -> bf[ InlineContent ]
Paragraph -> p[ InlineContent* ]
Table -> table[ TableRow+ ]
List -> list[ ListItem ]
TableRow -> tr[ TableCell* ]
ListItem -> item[ InlineContent* ]
TableCell -> td[ InlineContent* ]
TextBook_1 -> book[ Cover Body_1 ]
Cover -> cover[ Title Author* Publisher? ]
Body_1 -> body[ Abstract? Chapter_1* ]
Author -> author[ Text ]
Publisher -> publisher[ Text ]
Abstract -> abstract[ Text ]
Chapter_1 -> chapter[ Title Section_1* ]
InlineContent_1 -> inline[ Text|Bf_1|Em_1 ]
Section_1 -> section[ Title? (Paragraph_1|Table_1|List_1)* ]
Em_1 -> em[ InlineContent_1 ]
Bf_1 -> bf[ InlineContent_1 ]
Paragraph_1 -> p[ InlineContent_1* ]
Table_1 -> table[ TableRow_1+ ]
List_1 -> list[ ListItem_1 ]
TableRow_1 -> tr[ TableCell_1* ]
ListItem_1 -> item[ InlineContent_1* ]
TableCell_1 -> td[ InlineContent_1* ]
Bibliography -> bib[ (Book|Article|InProceedings)* ]
Book -> book{ Title Authors Editors Publisher? }
Article -> article{ Title Authors Journal? }
InProceedings -> inproc{ Title Authors Book }
Title -> title[ Text ]
Authors -> authors[ Person* ]
Editors -> editors[ Person* ]
Journal -> journal{ Title Editors }
Person -> person[ FirstName LastName ]
FirstName -> first[ Text ]
LastName -> last[ Text ]
==================================================================
```

Bookstore

```
==================================================================
Rule 1: html
------------------------------------------------------------------
Title->Text, PriceA->Text, PriceB->Text

==================================================================
Rule 2: wml
------------------------------------------------------------------
Title->Text, PriceA->Text, PriceB->Text

==================================================================
Rule 3: books-with-prices
------------------------------------------------------------------
T->Text, Pa->Text, Pb->Text

==================================================================
==================================================================
Type Definition:
------------------------------------------------------------------
wml -> wml[ card+ ]
card -> card[ Text_5 Text Text_6 Text Text_7 Text ]
Text_7 -> "Price B:"
Text_6 -> "Price A:"
Text_5 -> "Title:"
html -> html[ head body ]
body -> body[ table ]
table -> table[ tr tr_1+ ]
tr_1 -> tr[ td_3 td_3 td_3 ]
td_3 -> td[ Text ]
tr -> tr[ td td_1 td_2 ]
td_2 -> td[ Text_4 ]
Text_4 -> "Price at B"
td_1 -> td[ Text_3 ]
Text_3 -> "Price at A"
td -> td[ Text_2 ]
Text_2 -> "Title"
head -> head[ title_1 ]
title_1 -> title[ Text_1 ]
Text_1 -> "Price Overview"
books-with-prices -> books-with-prices[ book-with-prices+ ]
book-with-prices -> book-with-prices[ title price-a price-b ]
price-b -> price-b[ Text ]
price-a -> price-a[ Text ]
Bib -> bib[ Book* ]
Book -> book[ Book_attr title (Authors|Editor) Publisher Price ]
Book_attr -> attr{ Book_year }
Book_year -> year[ Text ]
title -> title[ Text ]
Authors -> authors[ Author* ]
Author -> author[ Last First ]
```

```
Editor -> editor[ Last First Affil ]
Last -> last[ Text ]
First -> first[ Text ]
Affil -> affiliation[ Text ]
Publisher -> publisher[ Text ]
Price -> price[ Text ]
Reviews -> reviews[ Entry* ]
Entry -> entry[ title Price Review ]
Review -> review[ Text ]
===================================================================
```

Bibliography

[1] XML Schema Part 1: Structures Second Edition. October 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

[2] XML Schema Part 2: Datatypes Second Edition. October 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-2/`.

[3] XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation. `http://www.w3.org/TR/xpath-datamodel/`.

[4] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[5] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. *Computer and System Sciences*, 66(4):688–727, 2003.

[6] U. Aßmann, J. Henriksson, and J. Małuszyński. Combining Safe Rules and Ontologies by Interfacing of Reasoners. In *Principles and Practice of Semantic Web Reasoning, International Workshop (PPSWR 2006)*, number 4187 in LNCS, pages 31–43. Springer Verlag.

[7] P. Barahona, F. Bry, E. Franconi, N. Henze, and U. Sattler. *Reasoning Web 2006. Second International Summer School. Tutorial Lectures.* Springer.

[8] S. Bechhofer. The DIG Description Logic Interface: DIG/1.1. In *Proceedings of DL2003 Workshop*, Rome, 2003.

[9] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ICFP 2003.* ACM Press.

[10] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From pattern-based to visual querying of XML and semistructured data. In *Proceedings of the 29th Intl. Conference on Very Large Databases (VLDB03) – Demonstrations Track*, Berlin, Germany, 2003.

[11] S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive typing rules for Xcerpt. In *Principles and Practice of Semantic Web*

Reasoning, International Workshop (PPSWR 2005), number 3703 in LNCS, pages 85–100. Springer Verlag. Errata: `http://www.ida.liu.se/~wlodr/errata.LNCS3703.pdf`.

[12] S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive Typing Rules for Xcerpt and their Soundness. Technical report, REWERSE, 2005. `http://rewerse.net/publications/download/REWERSE-TR-2005-01.pdf`.

[13] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, April 2001.

[14] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, May 1998.

[15] F. Bry, W. Drabent, and J. Maluszynski. On subtyping of tree-structured data: A polynomial approach. In *Principles and Practice of Semantic Web Reasoning, International Workshop (PPSWR 2004)*, number 3208 in LNCS, pages 1–18, 2004.

[16] F. Bry and S. Schaffert. A gentle introduction into Xcerpt, a rule-based query and transformation language for XML. Technical Report PMS-FB-2002-11, Computer Science Institute, Munich, Germany, 2002. Invited article at International Workshop on Rule Markup Languages for Business Rules on the Semantic Web.

[17] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proc. of the International Conference on Logic Programming*, LNCS. Springer-Verlag, 2002.

[18] F. Bry and S. Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics. In *Proc. 2nd Int. Workshop "Web and Databases"*, LNCS 2593, Erfurt, Germany, October 2002. Springer-Verlag.

[19] F. Bry and S. Schaffert. An Entailment Relation for Reasoning on the Web. In *Proceedings of Rules and Rule Markup Languages for the Semantic Web, Sanibel Island (Florida), USA*, LNCS, 2003.

[20] L. Cardelli. Type Systems. In Allen B. Tucker, editor, *The Handbook of Computer Science and Engineering, Second Edition*, chapter 97-1. CRC Press, 2004.

[21] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases W3C Working Draft. `http://www.w3.org/TR/xquery-use-cases/`.

[22] J. Clark and M. Murata (editors). RELAX NG Specification, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[23] J. Coelho and M. Florido. "XCentric: A Logic Programming Language for XML Processing". In *PLAN-X 07*.

[24] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. http://www.grappa.univ-lille3.fr/tata/, 1999.

[25] W3 Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/ .

[26] W. Drabent. Towards More Precise Typing Rules for Xcerpt. In *Principles and Practice of Semantic Web Reasoning, International Workshop (PPSWR 2006)*, number 4187 in LNCS. Springer Verlag.

[27] W. Drabent. Towards Types for Web Rule Languages. In *Reasoning Web, First International Summer School 2005*, volume 3564 of *LNCS*. Springer-Verlag, 2005.

[28] W. Drabent, J. Maluszynski, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–610, 2002.

[29] W. Drabent and A. Wilk. Combining XML querying with ontology reasoning: Xcerpt and DIG, 2006. RuleML-2006 Workshop. Unpublished proceedings http://2006.ruleml.org/group3.html#3.

[30] W. Drabent and A. Wilk. Extending XML Query Language Xcerpt by Ontology Queries. In *IEEE / WIC / ACM International Conference on Web Intelligence (WI 2007)*. IEEE, 2007.

[31] W. Drabent and A. Wilk. Extending XML Query Language Xcerpt by Ontology Queries. Technical report, REWERSE, 2007. http://idefix.pms.ifi.lmu.de:8080/rewerse/index.html#REWERSE-RP-2007-069.

[32] W. Drabent and A. Wilk. Type Inference and Rule Dependencies in Xcerpt. Technical report, Linköping University, 2007. http://www.ida.liu.se/~artwi/typeInference.pdf.

[33] Extensible Markup Language (XML) 1.1, February 2004. W3C Recommendation. http://www.w3.org/TR/2004/REC-xml11-20040204/.

[34] D. C. Fallside (ed.). XML Schema part 0: Primer. W3C Recommendation, http://www.w3.org/TR/xmlschema-0/, 2001.

[35] T. Furche, 2006. Personal communication.

[36] T. Furche, F. Bry, S. Schaffert, R. Orsini, I. Horrocks, M. Krauss, and O. Bolzer. Survey over Existing Query and Transformation Languages. Deliverable I4-D1, Institute for Informatics, Ludwig-Maximilians-Universität München, 2004.

[37] B. C. Pierce H. Hosoya. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[38] J. E. Hopcroft and J. D. Ullmann. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[39] IEEE / The Open Group. POSIX Regular Expressions, The Open Group Base Specifications Issue, 2001. `http://www.opengroup.org/onlinepubs/007904975/basedefs/xbd_chap09.html`.

[40] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler. *XQuery from the Experts. A Guide to the W3C XML Query Language.* Addison-Wesley, 2003.

[41] D. C. Kozen. *Automata and Computability.* Springer, 1997.

[42] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *PODS 2004*, pages 23–34, 2004.

[43] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. *Computer and System Sciences*, 73(3):362–390, 2007.

[44] A. Möller and M. I. Schwartzbach. The XML Revolution. Technologies for the future Web. `http://www.brics.dk/~amoeller/XML/index.html`.

[45] M. Murata, D. Lee, and K. Kawaguchi M. Mani. Taxonomy of XML schema languages using formal language theory. In *ACM Transactions on Internet Technology*, pages 660–704, 2005.

[46] F. Neven and T. Schwentick. XML schemas without order. Unpublished, 1999.

[47] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, 1992.

[48] F. Patel-Schneider and J. Siméon. The Yin/Yang web: A unified model for XML syntax and RDF semantics. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):797–812, 2003.

[49] P. Pietrzak. *A Type-based Framework for Locating Errors in Constraint Logic Programs.* PhD thesis, Linköping University, Sweden, 2002.

[50] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web.* PhD thesis, University of Munich, Germany, 2004.

[51] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.

[52] E. Svensson and A. Wilk. XML Querying Using Ontological Information. In *Principles and Practice of Semantic Web Reasoning, International Workshop (PPSWR 2006)*, number 4187 in LNCS.

[53] S. Thompson. A Haskell Library to Model, Manipulate and Animate Regular Languages, 2000.

[54] E. van der Vlist. *XML Schema. The W3C's Object-Oriented Descriptions for XML.* O'Reilly, 2002.

[55] E. van der Vlist. *RELAX NG.* O'Reilly, 2003.

[56] A. Wilk. Descriptive Types for XML Query Language Xcerpt, 2006. Licentiate Thesis. Linkoping University. `http://www.ida.liu.se/~artwi/lic.pdf`.

[57] A. Wilk and W. Drabent. A prototype of a descriptive type system for Xcerpt. In *Principles and Practice of Semantic Web Reasoning 2006*, LNCS. Springer-Verlag.

[58] A. Wilk and W. Drabent. On types for XML query language Xcerpt. In *Principles and Practice of Semantic Web Reasoning, International Workshop (PPSWR 2003)*, number 2901 in LNCS, pages 128–145. Springer Verlag, 2003.

[59] A. Wilk and W. Drabent. On Types for XML Query Language Xcerpt, Appendix. `http://www.ida.liu.se/~artwi/TypesForXML/appendix.pdf`, 2003.

[60] S. Dal Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. *Appl. Algebra Eng., Commun. Comput.*, 17(5):337–377, 2006.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and

Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-
318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002,
ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented
Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the
Computational Complexity of Temporal
Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for
Enabling Interoperability of Structured and
Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.