I4-D16

Initial Specification of a Language Extension with Types and Type Checking

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D16/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Claude Kirchner and Wlodek Drabent
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	29 February 2008
Actual submission date:	10 March 2008

Abstract

Types are a useful mechanism for early error detection and optimization of any programming language. Semi-structured query evaluation as in Xcerpt needs to be able to cope without schema information or types, in contrast to the relational case. Nevertheless, if available, type information enables query optimization well beyond what is possible in its absence. Furthermore, since semi-structured data such as XML allows diverging types, querying by type becomes a desirable feature of XML query languages, cf. XQuery's typed queries.

In this deliverable, we discuss joint work with I3 on typing in the Xcerpt language. The two working groups have jointly developed two approaches for extending Xcerpt with type information and type checking: The first is guided by the principle of least interference, providing typing for Xcerpt without any changes to the language itself. Type annotations for data and programs is provided by external means. It uses an abstraction of standard XML schema languages such as DTDs and XML schema as tree schemata. The second integrates closely with Xcerpt programs allowing finer granular type annotations. It is also unique in the support for graph schemata rather than tree schemata, allowing type checking also for Xcerpt references. To support type checking with graph schemata, a novel schema language, called R_2G_2 is introduced that allows for the description of graph schemata.

Keyword List

types, constraints, type checking, inference, Xcerpt

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

© REWERSE 2008.

Initial Specification of a Language Extension with Types and Type Checking

Sacha Berger[1]

[1] Institute for Informatics, University of Munich, Germany
`http://pms.ifi.lmu.de/`

10 March 2008

Abstract
Types are a useful mechanism for early error detection and optimization of any programming language. Semi-structured query evaluation as in Xcerpt needs to be able to cope without schema information or types, in contrast to the relational case. Nevertheless, if available, type information enables query optimization well beyond what is possible in its absence. Furthermore, since semi-structured data such as XML allows diverging types, querying by type becomes a desirable feature of XML query languages, cf. XQuery's typed queries.

In this deliverable, we discuss joint work with I3 on typing in the Xcerpt language. The two working groups have jointly developed two approaches for extending Xcerpt with type information and type checking: The first is guided by the principle of least interference, providing typing for Xcerpt without any changes to the language itself. Type annotations for data and programs is provided by external means. It uses an abstraction of standard XML schema languages such as DTDs and XML schema as tree schemata. The second integrates closely with Xcerpt programs allowing finer granular type annotations. It is also unique in the support for graph schemata rather than tree schemata, allowing type checking also for Xcerpt references. To support type checking with graph schemata, a novel schema language, called R_2G_2 is introduced that allows for the description of graph schemata.

Keyword List
types, constraints, type checking, inference, Xcerpt

Contents

1 Introduction **3**
1.1 Scope . . . 3
1.2 Contributions . . . 5
1.3 Outline of the Deliverable . . . 6

2 Preliminaries **7**
2.1 Context of this Research . . . 7
2.1.1 Information vs. Knowledge . . . 7
2.1.2 Documents and Data on The Web . . . 7
2.1.3 Obtaining Knowledge from Information . . . 8
2.2 Semistructured Data . . . 9
2.3 XML—The Extensible Markup Language . . . 9
2.4 From Schema-less Structure to Valid Data . . . 13
2.4.1 DTD—Document Type Declarations . . . 13
2.4.1.1 Limitations . . . 15
2.4.2 XML Schema . . . 18
2.4.2.1 Limitations of XML Schema . . . 22
2.4.3 Relax NG . . . 23
2.5 Querying The Web . . . 27
2.5.1 XPath . . . 27
2.5.2 XSLT . . . 28
2.5.3 XQuery . . . 30
2.5.4 Xcerpt . . . 34

I R_2G_2 **43**

3 R_2G_2—Regular Rooted Graph Grammars **45**
3.1 Regular Tree Grammars . . . 45
3.2 Regular Tree Grammars for Unordered Unranked Trees . . . 47
3.2.0.1 Current Modelling Formalisms for languages of Multisets . . . 48
3.2.0.2 Applications, Shortcomings and Extension of current Approaches . . . 50
3.3 Regular Rooted Graph Grammars . . . 50
3.3.1 Reference Types and Typed References . . . 51
3.3.2 About (Non Tree Structured) Graphs and Tree Grammars . . . 55

3.4 The Syntax of R_2G_2 . . . 56
3.4.1 Core R_2G_2 Syntax . . . 56
3.4.2 Base Data Types . . . 61
3.4.2.1 Definition of the Generic (abstract) Base Data Type System . . . 61
3.4.2.2 A pragmatic set of base types and functions . . . 62
3.4.3 Conversion functions . . . 62
3.4.4 Base functions . . . 63
3.4.4.1 *string* functions . . . 63
3.4.4.2 *rational* functions . . . 64
3.4.4.3 *integer* functions . . . 64
3.4.4.4 *boolean* functions . . . 64
3.4.5 The Non-Core R_2G_2 Constructs . . . 64
3.4.5.1 Namespaces . . . 65
3.5 XML Serialisation of R_2G_2 Valid Data Terms . . . 66
3.5.1 Examples and Explanations of (De-)Reference Serialisation . . . 66
3.6 Semantics of R_2G_2 . . . 69

4 From a Generic Module System to a Module System for R_2G_2 73
4.1 The Purpose of Modules . . . 73
4.2 Modules and XML Name Spaces . . . 74
4.3 Modular R_2G_2 . . . 75
4.3.1 Syntax of Modular R_2G_2 . . . 75
4.4 Realizing the Module System using the *"Divide and Rule"* approach . . . 76

5 Use Cases—Modelling Data and Documents with R_2G_2 81
5.1 Beyond Regular Tree Grammars—The Use of Macros . . . 81
5.2 Design Patterns—R_2G_2 Best Practices . . . 82
5.2.0.1 Design Objectives . . . 82
5.2.1 Global versus Local . . . 83
5.2.2 Composition vs. Sub Classing . . . 85
5.2.3 'eXtreme eXtensibility' . . . 87

II Automata Models 91

6 An Automaton Model for Regular Rooted Graph Languages 93
6.1 Introduction to Regular Tree Automata . . . 93
6.1.1 Handling Ranked Trees . . . 97
6.2 An Automaton Model for Unranked Regular Rooted Graph Languages . . . 98
6.2.1 Labelled Directed Hyper Graphs as Non-Det. Regular Tree Automata . . . 98
6.2.2 Membership Test for a Tree using Hyper Graph Automata . . . 100
6.2.3 Recognition of a Rooted Graph using Hyper Graph Automata . . . 105
6.3 A Calculus Relating Automata and R_2G_2 . . . 107
6.4 Some Set Theoretic Computations on Hyper Graph Automata . . . 110
6.4.1 The Emptiness Test . . . 111
6.4.2 Intersection of Regular Rooted Graph Automata . . . 113
6.4.3 Automata Based Subset Test for two Regular Rooted Graph Languages . . . 116

6.4.3.1 An Algorithm for Subset Graph on Tree Automata 118

7 A Model for Regular Languages of Multisets **121**
7.1 Introduction to Multisets and Multiset Languages 121
7.1.1 Multisets 121
7.1.2 Multiset Languages 122
7.1.2.1 Existing Approaches to Formalize Multiset Languages 122
7.1.2.2 Motivating a new Approach 123
7.2 Counting Constraints 124
7.3 A Calculus for Translation of Regular Expressions to Counting Constraints 125
7.3.1 Example of a Regular Expression Translated to a Counting Constraint 127
7.4 Some Set Theoretic Computations on Counting Constraints 128

III Type Checking 129

8 Type Checking using Regular Rooted Graphs as Data Paradigm **131**
8.1 Types and Query- and Programming Languages 131
8.1.1 Dynamic Typing and Type Checking in Programming Languages 131
8.1.2 Static Typing and Type Checking in Programming Languages 132
8.1.3 Combined Static and Dynamic Typing 132
8.1.4 From Typed Programming languages to typed Query Languages 133
8.2 Type Systems for Xcerpt 134
8.2.1 XcerptT—Descriptive Typing for Xcerpt 134
8.2.2 Prescriptive typing: from CLP to Xcerpt 134
8.2.3 Typing with R_2G_2—Differences to the Former Approaches 135
8.2.4 The Syntax of R_2G_2 typed Xcerpt 136
8.2.4.1 Typed Data Terms 136
8.2.4.2 Typed Xcerpt Construct Terms 137
8.2.4.3 Typed Xcerpt Query Terms 137
8.2.4.4 Typed Xcerpt Queries 138
8.2.4.5 Typed Xcerpt Rules 138
8.2.4.6 Typed Xcerpt Programs 138
8.3 Type checking and Type Inference for R_2G_2 Typed Xcerpt Programs 138
8.3.1 Paying Attention to Type Annotation in Queries 139
8.3.2 Typing Ordered Queries 139
8.3.2.1 Ordered Total Query Patterns 139
8.3.2.2 Ordered Partial Query Patterns 141
8.3.2.3 Descendants 141
8.3.2.4 Variables 142
8.3.3 Typing Unordered Queries 143
8.3.3.1 Unordered Total Query Terms 143
8.3.3.2 Unordered Partial Query Term 144
8.3.4 Typing of Construct Terms 145
8.3.4.1 Ordered Total Construct Terms 146
8.3.4.2 Unordered Construct Terms 148
8.3.5 Typing Rules 149

8.3.6 Typing Programs . . . 150
8.3.7 Coverage of Current Xcerpt Constructs . . . 150

IV Outlook & Conclusion **153**

9 Outlook **155**
9.1 Type Based Querying—an Extension to Simulation Unification . . . 155
9.1.1 Extending Ground Query Term Simulation With Active Type Querying . . . 156
9.1.2 Extending Simulation Unification with Active Type Querying . . . 156
9.2 Optimizing Xcerpt Query Evaluation Based on Type Information . . . 157
9.3 Integrating Types Into visXcerpt . . . 158

10 Conclusion **161**

Overview of this Deliverable

In this deliverable, we discuss joint work with I3 on typing in the Xcerpt language. The two working groups have jointly developed two approaches for extending Xcerpt with type information and type checking: The first is guided by the principle of least interference, providing typing for Xcerpt without any changes to the language itself. Type annotations for data and programs is provided by external means. It uses an abstraction of standard XML schema languages such as DTDs and XML schema as tree schemata. The second integrates closely with Xcerpt programs allowing finer granular type annotations. It is also unique in the support for graph schemata rather than tree schemata, allowing type checking also for Xcerpt references. To support type checking with graph schemata, a novel schema language, called R_2G_2 is introduced that allows for the description of graph schemata.

Both approaches are built on the principles outlined in I3-D4. The first approach is detailed in a companion month 48 deliverable by the I3 working group on typing of Xcerpt. Here, we focus on the second approach which has been developed in particular by Sacha Berger in his thesis [10].

Adding types to Xcerpt is answered in the following in three parts:

1. First, we argue that Xcerpt deserves a schema language tailored to its specific needs, in particular with the ability to describe the schema of graphs and not only trees (as existing XML schema languages). This argument and our proposal to address this need, the graph schema language R_2G_2, is presented in Part I.

2. Second, we show how to add types to Xcerpt and discuss their effect on syntax and semantics in Part III, where we also start the discussion of how to type check an Xcerpt program.

3. For that task, we employ a novel kind of automata introduced in Part automata. These automata allow the type checking of any Xcerpt program against an R_2G_2 schema.

We conclude the deliverable with a discussion of open issues and applications of typed Xcerpt.

Chapter 1

Introduction

The work in this deliverables is about types and schemata for Web query and transformation languages, in particular for the Web query language Xcerpt, that has been developed in the I4 working group of the REWERSE network of excellence.

1.1 Scope

Currently, many new application fields together with their domain specific processing and modelling languages are emerging on the Web. Domain specific modelling languages are often defined by standardisation consortia like W3C, ECMA, Oasis and others. Usually, definitions come along with human language description as well as formal definitions, using grammar and schema languages like EBNF, DTD [61], XML Schema [57] [58] [59] , or Relax NG [31].

Domain specific processing languages are often query languages defined by the same standardisation consortia issuing corresponding modelling languages. An example of such a domain specific language is e.g. the Semantic Web query language SPARQL [62], used to query knowledge modelled in the semantic web modelling language RDF.

Along with domain specific processing languages, some general purpose processing languages, like e.g. XSLT [47], XQuery [63] or Xcerpt [11][43], exist, with the rational to be used for implementation of arbitrary web applications, applicable to any domain specific data—also for writing inter domain applications. The processing languages on the web, either general purpose or domain specific, are usually untyped or dynamically typed.

Modern Programming languages nowadays in general come along with type systems to optimize evaluation or memory representation, check errors or just support programmers while programming when using sophisticated IDEs.[1] Type declarations in most typed languages are usually based on the concept of grammars—while a grammar defines a languages which is a set of words, a type declaration defines a class of objects which is a set of data instances.

Surprisingly, while many formalisms for data on the web are defined using grammars or schemata, their potential to serve as type declaration for processing languages is hardly exploited. The goal of this work was to evaluate the use of web schemata in practise for typing of Web query- or transformation languages.

[1] **I**ntegrated **D**evelopment **E**nvironment—these applications integrate all kind of tools useful for programming in a given programming language, starting from specialized text editors through re-factoring tools up to compilers.

The *typing of web query languages* is defined, such that types in the style of schema information are exploited to (1) find static errors in programs with help of schema knowledge (i.e. find errors prior to execution time, e.g. at compile time), (2) hint possible *odd behaviours* of programs, that could be desired, but maybe are not (i.e. constant results of a query, independent of input data), and (3) exploit automatic optimizations by automatically rewriting queries under given schema constraints for input or output data to more efficient equivalent queries (this is especially applicable under the consideration that syntactically different queries can be found for the same task).

A first step towards typing of web query languages was to evaluate the quality of current schema languages on the web for modelling data, from the point of view of a person willing to use a query language with type support and willing to use the given schema languages as type declarations. Almost all current schema languages on the web are based on so called regular tree grammars [38], with some extensions and also some restrictions. Common extensions, as e.g. found in XML Schema, are modelling facilities from object oriented systems. Restrictions are often very subtle trade offs in favour of simple adaption of legacy systems like SGML DTDs. During this evaluation it turned out that some restrictions where undesirable and some further extensions desirable. As the outcome of this step, a new type and schema language called R_2G_2 (short for **R**egular **R**ooted **G**raph **G**rammars) is presented.

The second step was to actually define, what kind of static properties are of interest and are detectable when the type or schema of the queried data or the type or schema of the desired result is known. In this part of the evaluation, properties that are related to static program analysis are addressed. Static program analysis, e.g. static type checking and type inference for programming languages is vital to ensure secure access of the random access memory representation of data structures. For query languages the necessity of secure memory access is arguably not a topic—query languages are used to *select*, *project* and *construct* data of a fixed data meta model. The meta model for the Web query language XQuery is XML.[2] As query languages operate 'within the bounds' of their meta model, no memory access violation is possible. So, what kind of static properties of query languages could be relevant and what impact could those properties have on the use of query languages? Assuming a query is selecting data from a document or database and the schema of the document is given, then some structural properties of the document are known prior to run time. For a query to data on the Web, it could e.g. be known, that the resources to be queried are HTML [49] documents. Assuming further, that the selection construct used as structured in such a way, that it may match data found in an SVG [54] document, but under no condition in an HTML document, then the query can be considered to never select anything for valid input data—the programmer most likely did a mistake and should be informed about it before run time.

Another static check that could be performed on a query typed with schema information is to check the validity of results if a schema is given as type for the construction parts of the query. A query can be considered to be ill typed, if the data that could result from query evaluation could be invalid with respect to the given schema. While a query never selecting anything may be in the intention of the author, a query constructing invalid data should be considered as an error. In both cases, these are sources of valuable messages to the programmer prior to program execution, possible hints for him, how to improve or correct the query.

The outcome of the second step is a type checking and type inference algorithm for Xcerpt (and Web query languages in general) with type declarations in R_2G_2. For pragmatic reasons the notion of types is defined in such a way, that type checking and inference is still possible in polynomial time.

[2] while the meta model for the query language SQL is the relational database.

The third step was to evaluate and exploit some possibilities of static optimization of typed Xcerpt programs, under the assumption, that the queried data is always valid with respect to the given types or schemata. The static optimizations in mind are related to program or query rewriting, not to modification of the program run-time environment, program translation or query evaluation engine. For given data there are usually different queries producing the same result. It is for many query language implementations likely, that queries with higher selectivity at the beginning of query evaluation evaluate faster as they sieve out irrelevant intermediate results at an earlier state of execution. This means that less candidate elements have to be tested during evaluation. A prerequisite for this assumption is that the programming or query language contains constructs for expression of incompleteness. Such constructs of incompleteness are not only more convenient for the query author, they are in many situations even indispensable due to incompleteness of the structure and shape of data on the web. Under the assumption of more certitude of the queried data due to given schemata or type definitions, it is often possible to reduce incompleteness by replacing corresponding constructs of incompleteness by query constructs selectively restricting the query, such that it focuses early on portions of data containing the desired results.

Achieving an optimal query by rewriting a given query, under type assumption as well as for untyped queries, is an undecidable problem, as the optimal shape of the query depends on the actual shape of the queried data instance. Further on, some query languages may be more efficient under certain forms of incompleteness, as of reducing the checks in concrete evaluation steps. A pragmatic approach to optimization is hence to try to achieve queries with usually better behaviour, based on heuristics about the given query evaluation engine. The outcome of this part of the deliverable is a rewriting rule system, relating typed Xcerpt queries with incompleteness to equivalent Xcerpt queries without incompleteness. The work has been presented on the First International Conference on Web Reasoning and Rule Systems [12]. The task of evaluating possible heuristics has not been addressed, as not sufficient knowledge about the run-time behaviour of current Xcerpt run-time engines exist.

1.2 Contributions

The main contribution of this deliverable can be summarized as

1. a Web Schema language to model (serialisations of) graph structured data, including a new modelling approach for unordered data or multisets, based on unordered interpretations of regular expressions, and

2. a typing approach for web query languages based on unrestricted regular tree grammars with former extension.

The first contribution, the type and schema language R_2G_2 has been implemented for integration, e.g. in Web query or transformation languages, but also for stand alone use as schema language for XML or Xcerpt data terms. The modelling approach for unordered data based on unordered interpretation of regular expressions (2) is implemented as a prototype based on constraint solvers in GNU Prolog [25]. Typing of Web query languages (3) has been implemented based on (1) for the Web query language Xcerpt. Currently two implementations exist, a prototypical one supporting ordered and unordered query terms, and a second implementation for integration into the prototype of Xcerpt currently under development. The second version offers no support for unordered data models by now.

1.3 Outline of the Deliverable

First, the context in which typing is to be applied is introduced. This spans various Web technologies, namely XML as document and data formalism, currently existing schema languages—they will be used as starting point for the proposed type and schema language, as well as various Web query languages.

Next, some shortcomings of the current schema languages are pointed out, and R_2G_2, a new type and schema formalism is presented. R_2G_2 has been conceived to overcome the mentioned shortcomings. A novelty in this languages is the use of regular expressions for the modelling of unordered data. A declarative semantics of R_2G_2 is presented along the syntax. The schema language presented is conceived with modularity in mind to be not only applicable to small examples but also to large scale projects. A generic module system (originally conceived for R_2G_2 and Xcerpt) is used to modularize R_2G_2. The part about R_2G_2 concludes with a use case section which is based on the popular SML Schema authoring style guidelines *"XML Schema best practices"* [23].

As basis for implementations of R_2G_2 as schema language as well as as type formalism, the operational semantics are presented. Those are based on regular tree automata, as e.g. introduced in [22] for ordered content and counting constraints [67] for unordered data. Regular tree grammars are a very general model, specified for reasoning about ranked tree structures. Common techniques for lifting unranked trees—common in the context of the Web and XML—to ranked trees exist, as however the context of R_2G_2 is the world of unranked trees (more precisely rooted graphs based on spanning trees with typed references), a new automaton model dedicated to reasoning about unranked data is presented. Using the new automaton model is arguably easier in the context of Web schema languages, as it aggregates all necessary information used to validate a node in the context of it's graph in one place, as well as a direct representation of a type for an XML node—transitions in this model represent node types. The automaton model is combined with Presburger arithmetic expressions, so called *counting constraints*, where unordered content models are needed. These constraints express relationships between the multiplicities of elements in a multiset—multisets are an appropriate model for unordered content of an element. Presburger arithmetic counting constraints are decidable. A calculus for deriving counting constraints for given regular expressions is presented. Last but not least, for the the automaton model as well as for languages of multisets declared using counting constraints, algorithms for reasoning about some set theoretic properties—namely emptiness test, union, intersection and subset test—are presented. These algorithms are crucial for type checking.

The next part presents a type checking algorithm for Web query languages based on the algorithms and techniques presented in the part about automata and counting constraints. The type checking is presented for the Web and Semantic Web query language Xcerpt, however, it should be easy to adapt the techniques to other Web query languages like e.g. XPath and XQuery, as all Web query languages share the concepts of data selection, variables, data projection and/or result construction. The presented algorithm focuses on data selection, construction and variable consistency check. The algorithm is not only suited to type checking of fully type annotated programs, but also provides type inference for partly type annotated programs.

Chapter 2

Preliminaries

2.1 Context of this Research

> The World Wide Web (known as "WWW', "Web" or "W3") is the universe of network-accessible information, the embodiment of human knowledge.

[W3C, `http://www.w3.org/WWW/`]

The Web is a kind of network of any information anybody around the world considers worth while to be published, almost no boundaries for quality, quantity, style, medium and format exists.

2.1.1 Information vs. Knowledge

A common mistake, when talking about information, is to misconceive information as knowledge. In short, information exists without any context, while knowledge is the result of thinking, understanding and solving problems. Information is useful to obtain knowledge and knowledge can be turned into information. Knowledge is information that is accessible at the right time in the right context to solve the right task.

The tremendous amount of information that forms the current World Wide Web, makes the difference between knowledge and information especially clear: it is usually easy to find information we are precisely able to name by using current search engines, but it is difficult to solve tasks with current web technologies that can not be searched for or of which we do not know the name. The root of the problem is, that machines are not able to understand – to *know* – the information stored on the Web.

Making the web accessible and understandable to machines as well as to humans will arguably be a major task in the future. While the amount of information increases vastly, it becomes intractable for humans such that some sort of "*machine assisted thinking*" is needed. Steps in this direction are taken by initiatives as the *Semantic Web*, that claims that machine understandable and semantically motivated annotation of the web is necessary. Generally speaking, end users of the Web will need general purpose tool support as it is known nowadays to programmers in the form of programming languages, to command their machines to assist them in digging for the right information on the web.

2.1.2 Documents and Data on The Web

Initially, the web was invented as a network of documents, where documents are written text meant to be read and understood by humans. Markups for stylistically beef-up of text and for semantic hints of

text or phrase structure or content type and a reference mechanism to external resources like pictures made the authoring of mostly any content fairly easy on the web. Further on, the Web provided mostly a handy support for citations, that made it possible through use of hyperlinks to access other documents in a seamless way not experienced earlier.

Shortly there after the Web document markup language, namely HTML, was extended to also represent tables and forms.

Later on, increasing numbers of views to traditional databases were made available on the Web, mostly as tables accessible by forms – the Data on the Web [3] was introduced interwoven with the Web of documents. Currently increasing numbers of machine accessible formats and so called Web Services extend the Web landscape with the data oriented aspect. The main difference between data and documents can be summarized as the usually more homogeneous structure used for data than for documents and often the lack of human language phrases – documents are instances of natural languages while data are instances of artificial languages. Prominent examples of database oriented content on the Web are on-line shops like Amazon (e.g. see `http://www.amazon.com/`).

Both aspects of the Web are equally important to human *consumers*, while arguably automatic consumption is mainly focused on data.

Recently traditional database approaches like querying data have been extended to web data and documents as an answer to the rising demand of flexible automated access methods to the web for end users. The interwoven nature of the Web as a Network of (mostly) structured data with (mostly) arbitrary schemata and (mostly) unstructured documents, motivates the so called *Semistructured Data* model as presented in [3] and briefly introduced in section 2.2. Arguably, Semistructured Data are the formal foundation of current technologies around automated processing of data and information on the Web.

Still, the current Web query languages are far from being end user friendly methods to extract information from the Web, even further from being tools to obtain new knowledge from the Web.

2.1.3 Obtaining Knowledge from Information

Former approaches supporting users to gather new knowledge out of information are known under the term *data mining*. Data mining is usually tailored towards experts of a topic, often with heavy statistical background, that dig in a closed and very homogeneous application specific domain database called *data warehouse*. The general nature of data mining in data warehouses can be summarized as: (1) the data warehouse is a (mostly) static materialisation of (mostly) lots of joins between data, (2) a closed world of data is queried, (3) the structure is well known. Application of traditional data warehouse approaches to broad scale web user base is arguably not suited for the task of extracting knowledge on the Web, for various reasons: (1) there is no way in visiting the whole web in a feasible way from an end users point of view, (2) the amount of data is by far too large to materialize any sort of expansion of the whole Web and (3) there is no inherent given structure of web data as a whole, as well as there is arguably no inherent structure of the knowledge of the humanity.

An assumption about the ways to dig for knowledge in the future web is, that **classical web searching** is to be combined by the end user with **automated web crawling**, and **deductive methods**, maybe inspired from artificial intelligence research, and **data querying on the web** as proposed for querying of Semistructured data. Combining all those methods in an end user friendly way is a highly challenging task that is by far not solved by now.

In the following, some foundations of the current and of the future Web as standardized to date are presented. A step towards merging two of those areas—querying and schematizing of data—is motivated as one of many steps in achieving the high goal of the end user friendly access to the future web. This is the step, that is addressed in this deliverable. The benefits of the integration of schematizing

and querying are supporting user friendliness of query languages by

1. extended error detection support,
2. potential optimization of queries and hence increased system reactivity, and
3. potential help for authors using specialized query development environments, e.g. by generating auto-completions or copy-and-paste templates based on schema information.

2.2 Semistructured Data

"Semistructured Data" are self-explanatory (or self-describing) data with or without schema (or data-model). The denomination "Semistructured" refers to the structure conveyed by tags combined with (possibly large) unstructured portions of text. The structure in question is a node labelled graph structure with ordered or unordered sequences of adjacent nodes. The graph structure is serialized using so called *Semistructured expressions* defined as follows:

- A Semistructured expression is either a (quoted) textual item, a data term or a reference to a data term.
- A data term consists of a label l, an optional unique identifier *id* followed by an `@`-sign (e.g. `i1@l{}`, and a (possibly empty) sequence of sub terms $t_1, \ldots, t_n$, that are
  - either enclosed in square brackets (i.e. $[t_1, \ldots, t_n]$ (denoting an ordered sequence of sub terms),
  - or enclosed in curly braces (i.e. $\{t_1, \ldots, t_n\}$ (denoting an unordered multiset of sub terms).
- A reference to a data term uses the identifier of the term to reference prefixed by a 'hat'-sign (e.g. `^id1`). Note, that reference and identifier declaration have to occur at arbitrary depth below a common Semistructured expression.

Considering Semistructured data in databases has been first proposed and investigated in a few research projects in the mid 90th. The book *"Data on the Web"* from Abiteboul et al [3] introduces in this direction of research.

Semistructured data is an interesting abstraction of XML data as presented in section 2.3. The Web and Semantic Web query and transformation language Xcerpt, as introduced in section 2.5.4, is syntactically strongly based on Semistructured expressions and it's data model is graph shaped and node labelled data with ordered or unordered sequences of adjacent nodes—just as Semistructured data.

2.3 XML—The Extensible Markup Language

XML is a generic markup language. Markup languages are traditionally used to mark up or annotate text, generic markup languages allow markup with arbitrary annotation elements. XML has been developed as a simplification of SGML, a former markup language developed in the 70th [1]. The simplification was mainly motivated by simplifying the implementation of efficient XML aware software and in supporting simpler document authoring by reducing the amount of rarely used or obscure features. To some extend the streamlining of SGML to what became XML was arguably inspired by the success of Java—roughly speaking a language near to a streamlined C++ quickly gained success for it's

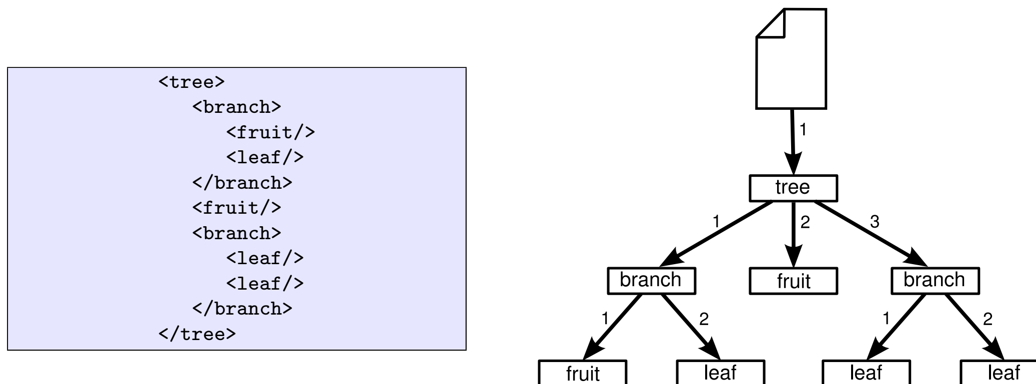


Figure 2.1: As an example of a document with elements, consider the apple tree modelled in this example. Various nodes of different name and/or type can be arranged in a hierarchical manner. The XML serialisation of the tree data structure on the right, is shown on the left.

cleanness and simplicity compared to it's predecessor. XML has been introduced by the W3C and is positioned as a development towards a more generic markup language than HTML. On of the first XML applications was a reformulation of HTML as XML application, called XHTML.

XML is a serialisation format for tree or to some extend graph structured data (see figure 2.1 for an example of a tree structured document and it's XML serialisation—in figure 2.3 a graph shaped example is presented, using so called XML *attributes* which have not been introduced so far). Different kinds of nodes with different properties form an XML instance. An XML instance is called *well formed* when the serialisation is syntactically correct.

XML is a *semi structured data* model, where semi structured means, that no structure is predefined or given for instances like e.g. in structured data models as known from structured programming languages (cf. `struct` in C or `record` in Modula). On the other hand, the data **is** structured by the given structure of the instances. To a certain extend the data is structured in an abstraction of the data model that represents tree structures in general.

Another interpretation of *semi structured* in the context of XML is the possibility to provide a schema for the data, yet the schema is not enforced to be used or validity upon the schema is not enforced – instances **may** be structured according to a schema or they **may** be schema-less. Schemata for XML are currently known under the names DTD, XML Schema and Relax-NG.

Elements are XML nodes, that are always the child of exactly one node and may contain arbitrary many child nodes. See figure 2.1 for an example. The child nodes are organized in an ordered sequence. Elements also have a textual label. In a well formed XML document elements are serialized as an opening tag and a closing tag surrounding the serialisation of the child nodes. An opening tag is serialized as the label in pointy braces and the closing tag as the label preceded by a slash in pointy braces. Empty nodes can be abbreviated as the label followed by a slash in pointy braces. As elements always have a parent node, a kind of root is needed, usually a document node forms that root. A document node may only contain one root element.

Attributes are nodes with a textual name and a textual value which are not members of the child node sequence of an element but of an (unordered) attribute set. Each element has one attribute set

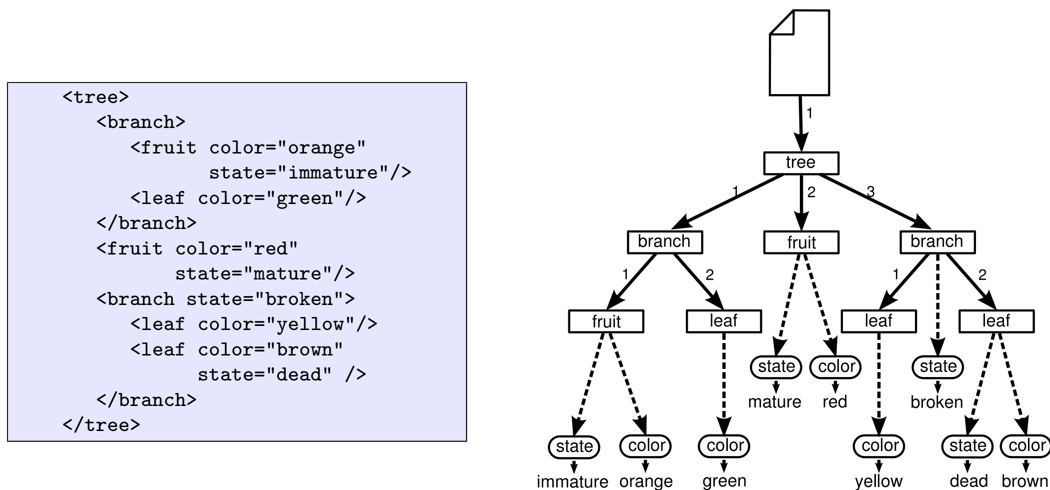


Figure 2.2: This example extends the apple tree example in figure 2.1 such that the elements are annotated using attributes. Elements may contain arbitrarily many attributes, but not two may have the same name. Note, that attributes have no explicit order in which they occur.

that may also be empty. See figure 2.2 for an example of a document with elements and attributes. In a well formed XML document instance two attributes with the same name may not occur in one attribute set. Attribute sets are written within the opening element tag's pointy braces. A well formed attribute is written as it's name followed by an *equal*-sign and the value quoted either in single or double quotes, mutually excluding the possibility to use the other quotes in the value text. Commonly attributes are interpreted as meta information about the element, but there is no formally dictated use for attributes. When modelling graph shaped structures, commonly attributes are used to declare identifiers and references to identifiers, as presented in figure 2.3.

Character Data Character data nodes are nodes without child nodes and attributes, they consist only of textual information. Character data nodes are members of a child node list, i.e. they are members of the ordered nodes. In a well formed XML document instance character data nodes are serialized just as the text they represent, without any quotation. Pointy braces[1] and the ampersand sign may not occur in the well formed serialisation, they have to be substituted by so called character entities, which are themselves general entities. Two character data nodes may not be direct neighbours, as then they would collapse in the serialisation to one node.

Initially, eg. in the context of SGML documents, the textual content was considered to be of major importance, while elements were mere annotation of the textual content. The lack of quotation of character data makes XML especially convenient for document authoring, where the major content is arguably textual content. In the context of data modelled in XML, textual content and elements are arguably of equal importance, yet often character data nodes are ignored in research for pragmatic reason.[2] See figure 2.4 for an example of an XML document with character data.

Further Core Features Further features of XML such as

[1] Indeed, only the opening pointy brace may not occur in character data serialisations.

[2] Conceptually, there is no big difference between a leaf element and a character data node.

```
<cycle>
   <node id="n1" idref="n2" />
   <node id="n2" idref="n1" />
</cycle>
```

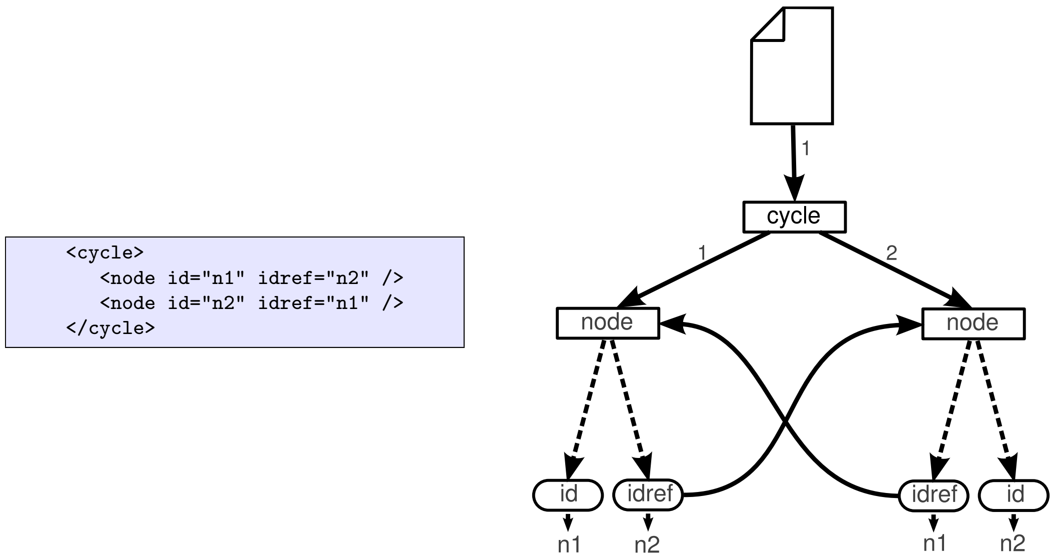


Figure 2.3: This example presents the ability to model graph structures beyond tree shaped data in XML—identifiers and references to identifiers are used in this case. Identifiers and references are given as XML attributes. Actually, the attributes have to be declared being of identifier or reference type, e.g. using a DTD as introduced later in section 2.4.1.

```
<pamphlet>
 <title>
  Document Markup with XML
 </title>
 This pamphlet illustrates the
 use of <acronym>XML</acronym>
 character data, also known as
 <acronym>CDATA</acronym>,
 together with markup.
 <footnote>
  Note, how <bold>semantical</bold>
  markup is used along with
  <bold>stylistical</bold> markup.
 </footnote>
</pamphlet>
```

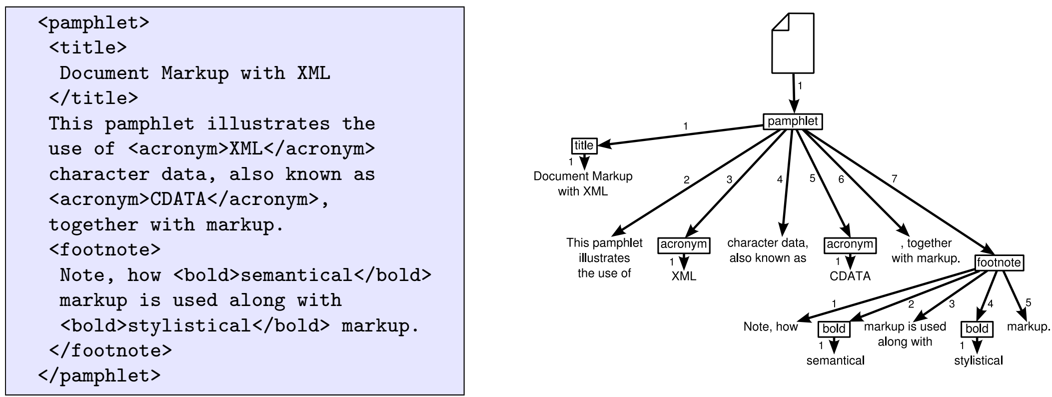


Figure 2.4: This example presents a document centric XML instance. In this case, the serialisation (left) is a more appropriate view on the content than the tree representation on the right, some words or phrases are annotated with markup surrounding some parts of the text.

- **Character encoding** specifying the used character encoding for the serialisation of character data, attributes and elements
- **Processing instructions** are meta information about the document. Processing instructions are information that is considered by the application processing the serialised data. Like character data processing instructions may not be further structured.
- **Comments** are not members of the tree structure neither. Like character data comments may not be further structured.
- **General entities** are a macro expansion or substitution mechanism for abbreviation of XML node sequences or special characters not expressible in the chosen document text encoding—any Unicode character can be generated this way using a so called *numerical character entity* which itself is a general entity

are not mentioned in detail, since they are not needed in this deliverable.

2.4 From Schema-less Structure to Valid Data

Two different interpretations of the term *"Semistructured"* exist for Semistructured Data, of which XML is a representative:

- Data may be created without any notion of a schema declaring a structure, so it is unstructured in the sense of schema-less, while it is structured in the sense of being tree or graph structured data that has the structure of the instance—being structured with respect to the data meta model.
- A schema for data may exist and data can be an instance of the given schema, but data without schema is also tractable as so called well formed data.

In both interpretations the notion of a schema is involved. In between the two extrema a schema can be given and consequently being applied for many sections of a data instance, while exceptions may exist for other sections, in the form of allowing elements of any type for these sections.

For XML different schema formalisms have been introduced, partly ad-hoc or inspired by the predecessor SGML, partly conceived from a theoretical foundation. The widely accepted theoretical foundation of XML schema languages are so called *regular tree grammars*[16], [38]. Regular tree grammars are a sub category of context free languages, that provide a controlled way of modelling the bracket structure inherent to tree serialisations, yet preserving the desirable properties of regular languages being closed under union, intersection and difference. A well formed XML document may be *valid*, if it is an instance of the language generated by the grammar given in the form of a schema instance. In depth explanations of regular tree grammars is given in the chapter about R_2G_2 (see 3).

2.4.1 DTD—Document Type Declarations

DTDs are the oldest schema mechanism for XML, inherited from XML's predecessor, SGML. While an SGML document requires a DTD (and validity to that document), an XML document may or may not have a DTD and may or may not be valid to that DTD. The DTD may be defined in the XML document instance or at an external resource. Note, that every valid XML document has to be well formed while not every well formed document has to be valid.

Element Type A DTD consists of a set of element type declarations, describing the structure of an element. An element type declaration relates an element name to the valid content model of elements with that name. For each element name declared in an element declaration, there is no other element declaration with the same element name, i.e. the element declarations are unambiguous. In a valid document instance, all elements are valid with respect to an element type declaration.

An element type declaration for an element name l and a content model c is written as

<!ELEMENT l c>

Document Type In a valid document not only all elements are valid, additionally the root element has to be of the type given by the document type declaration. The document type declaration is always part of the document instance, while the element type declarations (and also the attribute sets mentioned later) may also be in an external file. The document type declaration also relates a set of element type declarations to a document.

One form of document type declarations relate an external set S of element type (and attribute type) declarations to a document and to its root node with label l written as

<!DOCTYPE l S>

The other form relates the root label l of the document with the element type (and attribute type) declarations R directly in the document as

<!DOCTYPE l [R] >

In both cases the document type is declared after the document encoding (cf. <?xml encoding="..."?>) and before the opening tag of the root element label.

Content Models A content model for an element is given as a regular expression of element names. A valid element with respect to an element type declaration has a sequence of child nodes, that is recognized by the regular expression, i.e. the sequence of element names of the child nodes are recognized by the regular expression and the child nodes are valid with respect to an element declaration for elements of their name. Regular expressions are formed by *sequencing* the element names with comma, *optional* parts get a question mark appended to the regular expression part, *disjunction*s are formed by separating two disjunctive parts with a pipe sign (vertical bar), and *repetition*s are modelled by appending a star or a plus sign (star for zero to many, plus for one to many repetitions) to a regular expression.

The following example motivates the concept of regular expressions in the context of DTD's. A valid `article` element contains first a `title` element, followed by an arbitrary number of `section` or `paragraph` elements and at last an `appendix` element, that may also be omitted. Note, that the repetition of `section` or `paragraph` involves a recursive composition of regular expressions, as the disjunction is also a regular expression on its own.

```
<!ELEMENT article  (title, (section|paragraph)*, appendix?)>
```

Attribute Sets Another part of the content model of an element are the attributes. Attributes are modelled in a different way, as they do not occur in any explicit order, and because regular expressions impose an order on the atom instances occurring in accordance to a regular expression. The attributes of an element are declared with an attribute set. An attribute set relates an element name to some attribute definitions. An attribute definition can be optional – in terms of DTD *implied* – or required. Attribute

values may be arbitrary character data, a fixed value, an (almost) arbitrary string of character data being unique in the context of attribute values of that type, forming an unique identifier or a reference (even a sequence of references) to such unique identifiers.

An attribute set declaration is introduced by the keyword `ATTLIST`. The first token in the attribute set declaration is the name of the containing element. The declaration is followed by a sequence of triples consisting of (1) attribute name token, (2) attribute value type declaration and (3) attribute default declaration. As an example consider the following attribute set declaration for `leaf` elements as they occurred in figure 2.2. The color is a mandatory attribute while the state is optional.

```
<!ATTLIST leaf color CDATA #REQUIRED state CDATA #IMPLIED>
```

Types of Character Data The content of an element may also be declared to be character data. In this case no elements may occur together with the text.

Mixed Content The content of an element may be a mixture of character data and arbitrary elements, so called *mixed content*. In this case any declared element or character data may occur in arbitrary combination.

2.4.1.1 Limitations

The document definition language DTD has some limitations. These led to the development of its successor, the *XML Schema* language.

Alternative Content Models For Equally Named Elements There is no way to specify different content models for an element, according to the context it occurs in. The example 1 of a university schedule illustrates the problem:

```
<university-schedule>
 ...
 <lecture>
  <name>Semi Structured Data and Markup Languages</name>
  <lecture>
   <time>Wednesday, 10:00-13:00</time>
   <location>E101</location>
  </lecture>
  <tutor>
   <name>
    <first>Benedikt</first>
    <last>Linse</last>
   </name>
  </tutor>
  <lecturer>
    <name>
     <title>Prof.</title>
     <first>Francois</first>
     <last>Bry</last>
    <consultation>
     <time>Tuesdays, 9:00-10:00</time>
     <location>D1.03</location>
    </consultation>
  </lecturer>
 </lecture>
 ...
</university-schedule>
```

Code Example 1 An example of an university schedule modelled using XML. The term `name` is reused in different contexts (each time meaningful, but with different meaning), however this can not be modelled in a satisfying way using DTD.

A lecture is presented. A lecture has a name, may have arbitrary many tutors for exercise courses and a lecturer, for the lectures. The ambiguous term *lecture* gives rise to the bizarre fact, that a lecture as an administrative unit contains the information about the lecture as a time and locational bound entity, the event of instructing. In this example the same name has been used for the different meanings of the same term, like it is used in human language. Yet the elements share the same name, they occur in different parent node contexts and they have a different structure. A second example of label ambiguity is the *name* element, which is used once with plain CDATA as content to represent the name of the lecture and once with structure to model the names of the lecturer and the tutors. Again, the parent node content is essential for the choice of the proper content model of equally named elements. The hierarchical context is not generally for all possible XML applications the determining property of the content model of ambiguous elements, the occurrence in the list of siblings or the attributes of an element may also be determining the choice of the right content. When modelling data with DTDs the content model is associated to elements with a certain name not to elements in a certain context. As there is no way to declare concurrent element declarations and especially to refer to one alternative of the concurrent declarations, such documents as the presented university schedule can not be modelled properly. As a 'work around', usually the alternative content models are provided as a disjunctive content model, yet allowing the use of the wrong content model in a given context.

Data Types for Atomic Values An often criticised limitation of DTDs, is the lack of integrated data types other than character data, e.g. integers, enumeration types or similar. For text document centric modelling, the lack of such data types is not a major topic. The first goal of XML is, as stated by the W3C itself, that *"XML is for structuring data"* . For data modelling a rich set of data types is mandatory.

The successor of DTDs, XML Schema, is equipped with a rich and slightly extensible set of data types to model many restrictions of strings, lots of number formats, date information and some other data types.

Namespaces DTDs are not namespace aware, it is not possible to model document types with labels in a specific name space. More or less difficult workarounds exist, but they always rely on modifying the DTD by overriding parts, e.g. special entities simulating namespace expansion, in the document instances.

Deterministic Content Models A less often mentioned limitation of DTDs is the restriction to so called *deterministic content models*. As an example of a non deterministic content model, consider the following element declaration of a chess game with alternating moves of the black and the white figures:

```
<!ELEMENT chessgame  ((white,black)*,white?)>
```

The content model allows the game to end after each move, yet modelling the alternating colors of the moves. Intuitively speaking, when a white move instance occurs, the DTD validator may mot be sure which part of the content model to choose (e.g. is the move valid with respect to the `(white,black)*` or the `white?` sub-expression. It is easy to think of a deterministic automaton representing exactly the given content model: it consists of two final states interlinked using an "*a*" transition from one to the other and a "*b*" transition from the other to the one state[3]. The W3C explains the non determinism of DTD content models in a non normative section of the XML Recommendation (see `http://www.w3.org/TR/REC-xml/#determinism`) as follows:

> [...] a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman]. In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error. [...]

Weak Unordered Content Support Two different notions of disorder can be addressed: (1) The order of elements in a data instance is relevant, but from the point of view of a schema, valid data instances may occur in arbitrary order. (2) The order of elements in data instances is irrelevant, this applies i.e. when modelling a set or multi-set.

To be able to express that certain elements may occur in arbitrary order, in XML DTD the author of the schema is usually forced to give all possible permutations of the order. In addition, the *deterministic content model* property mentioned in the previous paragraph must be retained, making the task even more cumbersome. With the XML predecessor SGML, the DTD formalism had an option to model a sequence of elements with arbitrary order (an *unordered* sequence operator was available), but the feature has been dropped for simplification purpose. When it comes to the necessity of modelling unordered content of elements of different type, XML DTD authors hence tend to simplify and to state that arbitrary many elements of a choice of types are allowed in arbitrary order—this can easily be modelled as a disjunction with a Kleene star, i.e. `(A|B|C)*` for elements of type `A`, `B` and `C` in arbitrary order and arbitrary multiplicity.

[3] Automata are a common way to implement recognition of instances based on regular expressions. See chapter 6 about automata for more information about that topic.

For modelling data without order, i.e. sets, XML provides no more than attributes, their order is irrelevant as their interpretation is usually a set of name/value pairs. No way exist to model multi-sets in XML, except by shifting the unordered interpretation of a sequence to the application level.

Modelling Attributes Concerning the ways of modelling attributes, two limitations are known: First, attributes can only be modelled as a list or a set of valid attributes with or without optionality. It is not possible to model dependant attributes, e.g. subsets that may only occur together, neither choices of attributes nor subsets of attributes can be modelled. Second, attributes and the content model of an element are modelled independently – it is not possible to have e.g. the content of an element to be dependant of an attribute chosen.

Typed References In the context of modelling graph structured data, a new shortcoming appeared: Using ID and ID-Ref typed attributes is suitable for modelling graph shaped data, an identifiable element, annotated with an ID attribute is usually easy to get by de-referencing its ID as found in any ID Ref attribute using common XML tools or application programming interfaces like DOM. However, it is not possible to model data in a schema in such a way, that the reference is a reference to an element of a special type. This virtually renders the reference mechanism a typeless or schema-less reference mechanism. Using DTDs it is neither possible to set or restrict the amount of references in an ID Ref attribute.

2.4.2 XML Schema

XML Schema has been introduced as successor of DTDs to overcome (most) of its limitations. XML Schema is a W3C Recommendation. It is widely accepted, that XML Schema is by far the most complicated recommendation currently available from the W3C. XML Schema is itself an XML application, i.e. it has an XML syntax.

The Schema A Schema is defined in an XML document with root node label `<schema>` in the namespace `http://www.w3.org/2001/XMLSchema`. A Schema contains element and type declarations. There is no way in declaring schema parts directly in the instances, yet there is the possibility to directly annotate instance nodes with schema or type information using the so called [58]. The schema validation is sometimes seen as a transformation of the information set[4] representing the XML document to the so called *Post Schema Validation Infoset* (see the W3C's XML Schema recommendation part 1 [58]), where each information item of the information set is annotated with additional type or schema information deduced from the schema.

Element and Type Declarations An element declaration declares an element with a given name in a given namespace with a given content, where the content is also called *type* in terms of XML Schema. Two different kinds of types exist in XML Schema: so called (1) *simple types* that represent solely character data information, and (2) *complex types* that may be schemata for element sequences, character data and attribute sets. Character data as modelled using simple types may be any XML Schema data type or derived variations. When modelling complex content, different possibilities exist for structuring the schema: (1) Child content of a node can be modelled as nested element declarations inside of the corresponding complex type of an element which in turn is nested inside the element declaration.

[4] The information set is a normative W3C recommendation [56] defining an abstract syntax in the spirit of a parse tree of XML documents.

This paradigm is often referred to as *Russian doll* design – all information is nested. (2) The element declarations for child content can be referred to using an identifier (possibly different to the element label) from inside of the content model declaration located inside of the element declaration. This paradigm is commonly referred to as *salami slices* design – the element declarations are much thinner, the reference (thin as a salami slice) points to the *"beefier"* part. (3) Element declarations occur nested inside of the content type declarations, content type declarations are declared at top level and are referred from inside of the element declarations using unique names. This paradigm is referred to as *venetian blinds* design. The name is due to an arguably obscure XML Schema feature concerning namespace exposure that can best be exploited using this design pattern. With a small change to the schema big amounts of the schema can be exposed or hidden, like venetian blinds open and close with a small trigger action.

Note, that e.g. the reference mechanism for content model references in elements helps in removing one of the limitations of DTDs: alternative content models for equally named elements according to the context they occur in.

Attributes may occur in complex content declarations. Attributes have to be declared *after* the element declarations, they may not be declared inter-winded.

Namespaces Arguably, the most prominent advantage, that XML Schema provides over DTD is the support of name spaces. Elements can be declared to belong to a namespace, as well as it is possible to import and incorporate declarations of elements of another namespace in a type declaration.

Regular Content Models in XML Schema Content models can be modelled using constructs that are mostly inspired by regular expressions. The XML Schema elements `sequence`, `choice` and `all` are used to model content models. They may be nested, roughly representing a structure similar to a parse tree of a regular expression. With `sequence` it is possible to give a list of elements or content model components that have to be matched in a sequence conforming to that content model. With `choice` alternatives may be specified. The `all` construct is a short hand for an unordered sequence of elements that have to be matched, any sequential combination of elements specified in an `all` is represented. The unordered specification is very restricted, just elements may be child nodes of an `all` construct and the multiplicity of the elements always has to be exactly one. Repetitions can be modelled quite accurately using the `minOccurs` and `maxOccurs` attributes of the regular constructs or of the element declarations. To represent unbounded repetition as modelled in DTDs and regular expressions using the Kleene star, the special multiplicity value `unbounded` is used.

Attributes Attributes are defined in a similar way as in DTDs – their name and value type is given, and they may be declared mandatory, optional or as fixed attributes. The value type of attributes, however, may be specified in a much richer way than possible for DTDs. Here any simple type is possible.

Data Types XML Schema provides a fixed yet versatile set of predefined simple types for modelling various number formats, Gregorian calendar data and various forms of strings and tokens. User defined restrictions and extensions of the predefined simple types can be derived. There is also one predefined polymorphic data type that can be user restricted: a polymorphic list of (non list) simple data types.

Clearly this helps to overcome a limitation of DTD: DTD has no data types except strings.

XML Schema Example The example 2 illustrates the presented concepts of XML Schema. An university database is modelled. A university contains a sequence of student elements, at least one and as

many as needed. Student elements contain a *name*, a *matriculation number*, the *semester* information of the student and an element with the results of the *exams* of that student. All those elements must occur, but they may occur in arbitrary order. A name consists of a *first* and a *last* name, each represented by elements containing plain text strings, the *semester* information is an element containing an integer number between 1 and 24. The *exam* elements contain an integer representing the *grade* and a string for the name of the *lecture*, both information in form of an attribute.

The content of the *university* element is modelled outside of the nesting focus of the element in the spirit of the formerly mentioned salami slices paradigm, the *student* is mostly modelled according to the Russian doll paradigm, where all elements are declared inside of the nesting scope of the student element, except the *name* element, which has its content type declared out of scope. The *exam* element as well has the content type defined out of scope in the spirit of the venetian blinds paradigm.

```
<xs:element name="university">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="student" minOccurs="1" maxOccurs="unbound" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="student">
  <xs:complexType>
    <xs:all>
      <xs:element name="name" type="nameType" />
      <xs:element name="matriculation-number" type="xs:integer">
      </xs:element>
      <xs:element name="semester">
        <xs:simpleType>
          <xs:restriction base="xs:integer">
            <xs:minInclusive value="1" />
            <xs:maxInclusive value="24" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      </xs:element>
      <xs:element name="exams">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="exam"
                        minOccurs="0" maxOccurs="unbound"
                        type="examType" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    <xs:all>
  </xs:complexType>
</xs:element>

<xs:complexType name="nameType">
  <xs:element name="first-name" type="xs:string" />
  <xs:element name="last-name" type="xs:string" />
</xs:complexType>

<xs:complexType name="examType">
  <xs:attribute name="grade"   type="xs:integer" />
  <xs:attribute name="lecture" type="xs:string" />
</xs:complexType>
```

Code Example 2 A simple XML Schema modelling some aspects of administrative data found in an university.

Extending and Restricting Declarations An XML Schema content modelling feature orthogonal to the formerly presented is the extension and restriction of types. When modelling simple (data) types, it is possible to restrict a given simple type using type specific facets. This has already been shown in the example above for the restriction of integers to integers greater than 1 and smaller than 24. Strings can be restricted in a very powerful way using regular expressions. For complex types in addition to restriction also extension of the content model is possible. A typical example of extending a content model is to add attributes to an already declared complex type. Restriction and extension of complex types is however a mere integrity constraint on the complex type supplied, i.e. a content type can be

denoted to be an extension or restriction of another content type, but the user (the author of the schema) has to model it according to that constraint, otherwise the schema is invalid.

2.4.2.1 Limitations of XML Schema

As presented, many of DTD's shortcomings have been overcome in XML Schema (e.g. lack of name space support, lack of data types and the lack of alternative content model declarations for different element contexts), yet some remain and some new arise.

The Predefined Set of Data Types While a rich set of data types helps in modelling data types, the predefined data types lack some flexibility. Only predefined ways of restriction exist. For many applications different kinds of atomic data types exist, that can not be modelled using XML Schema. As an example IP-addresses are mentioned: yet difficult to model using e.g. regular expressions (and hence sub-typing them from plain strings) there is no way in supporting the data type itself without further application dependant parsing (e.g. as 4 Byte quadruple). Some kind of modularity or external interface is arguably desirable for simple data types. Another drawback is the fixed constraints on the format of the predefined data types: the date data types are arguably of restricted usefulness, as there is no way to allow localization of the date format and as the calendar model is restricted to Gregorian dates (which is a nuisance e.g. in china).

Deterministic Content Models A limitation of DTDs that has been inherited by the XML Schema language is the content model restriction to deterministic content models. Hence, the chess game use case of the DTD limitations can not be modelled in XML Schema.

Attributes Modelling of attributes of an element has conceptually mostly been inherited from DTDs, thus with the same limitations: attributes are declared independently of each other, it is not possible to declare them such, that e.g. a set of attributes has to occur together or excludes another set of attributes. The independence of content model and attributes has also been inherited from DTDs, as attributes most be declared in the complex content model at the end after all element declarations. Therefore it is not possible to have an element content that either depends e.g. on the occurrence of an attribute or on a certain value of an element.

Typed References DTDs lack of typed references is roughly solved by the possibility to syntactically restrict attribute values yet having them declared as identifiers or references to identifiers. A disadvantage of this approach is, that the typed reference has to be reflected in the way instances of the schema are authored, the values have to fulfill some syntactical properties (e.g. they need a certain prefix) that are conceptually irrelevant to the instances.

Root Element A rather surprising limitation is the lack of an explicit root element declaration when modelling using XML Schema. Each element that is valid to some element declaration of a schema can also be the root element of a valid document. The following example document

```
<bar />
```

is hence perfectly valid with respect to the following schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema";
           elementFormDefault="qualified"
           xmlns="http://www.example.com";
           targetNamespace="http://www.example.com";>
  <xs:element name="foo">
    <xs:complexType/>
  </xs:element>
</xs:schema>
```

The document is valid, as the element "foo" may contain arbitrary content, hence also "*bar*". Arguably, the author of the schema intended to state, that the root element has to be a "foo" element.

Unordered Content Practical applications have proved the relevance of unordered content modelling for XML, especially for data modelling. This reflects grouping and encapsulation of attributes[5] in an object oriented modelling paradigm where it is relevant e.g. to assign the right values to the attributes, yet the order is irrelevant. While SGML DTD has some support for that task, XML DTD lacks that feature. XML Schema reintroduced the SGML feature and thus provides a limited way of modelling unordered content using the *all* construct by specifying a set of elements that has to occur in a valid content instance. When necessary to model unordered content involving optionality, multiplicity or dependant multiplicity of attributes, XML Schema's *all* unordered content modelling facility can not be used. Arguably, the same expressiveness as given by regular expressions for sequences, would be desirable for unordered content modelling.

Syntax and Semantics Thus XML Schema solved some of the shortcomings of DTD, some problems remain. Further on, XML Schema enjoys the questionable fame of being by far the most complicated W3C recommendation available. The recommendation is often expressed in a way how the validation procedure works and how it has to be implemented, rather than explaining in a declarative way how the language has to be used. This is of advantage for XML Schema implementing parties, yet it is very inconvenient for XML Schema users. It requires considerable expertise to be able to understand a W3C XML Schema correctly. Additionally, there is no accepted or official formal description of XML Schema by now, yet a working draft *"XML Schema: Formal Description"* (see `http://www.w3.org/TR/xmlschema-formal/`) last edited in September 2001 exists.

2.4.3 Relax NG

Relax NG (REgular LAnguage for XML Next Generation) is a schema language for XML, based on RELAX [30] and TREX [51]. A Relax NG schema specifies a pattern for structure and content of XML documents. Relax NG has two syntices—an XML syntax and a textual so called *compact syntax*. Compared to e.g. XML Schema, Relax NG is arguably simpler with the same expressive power. It is defined by a committee specification of the OASIS Relax NG technical committee, and also by part two of the international standard ISO/IEC 19757: Document Schema Definition Languages (DSDL) [2].

Relax NG has been defined based on formal language theory, more precisely based on regular tree grammars [22], which are considered to be an appropriate formal model for most current XML type and schema languages [38].

Grammar A schema or type declaration in Relax NG is called a grammar. A grammar consists of one or many grammar rules, of which one is declared to be the starting rule, the rule defining the type of the root of valid documents.

[5] In this context by *'attributes'*, attributes of an object are meant, not forcibly XML attributes.

Element Declarations A Relax NG grammar rule declares the shape of an element. As an example, consider the grammar of a name element consisting of a first- and a last name component (see example 3). In contrast to DTDs, there is a separation of non-terminal and terminal symbols, i.e. it is possible to use a different label for the element instances and for the rule (we see, that an element with label `<first>` is modelled, this element is called *firstName* in the grammar). The separation of label and non-terminal in the grammar is essential to be able to reuse element labels for various element declarations with different structure in e.g. different content models.

For the sake of simplicity, just the compact syntax is used in the Relax NG examples—con- ceptually the XML Syntax and the compact syntax are equivalent.

```
grammar {
        start = element name{ firstName , lastName }
        firstName = element first{ text }
        lastName = element last{ text }
}
```

Code Example 3 A Relax NG example modelling a name element composed of a first- and a last name component.

Regular Content Models in XML Schema The content models of elements are modelled using regular expressions as sen for DTDs. In contrast to the DTD regular expressions, there is no restriction on possible regular expressions to one-unambiguous expressions. It is also possible to model data the *"Russian doll"* way as in XML Schema—by embedding the element declarations anonymously in a content model declaration (see example 6 for nested element declarations and full blown regular expression content models).

```
grammar {
        start = element addressBook{
                element card {
                        ( firstName , lastName ) | ( companyName ) ,
                        element phone{ text } ? ,
                        element address{ text } ?
                } *
        }
        firstName = element first{ text }
        lastName = element last{ text }
        companyName = element company{ text }
}
```

Code Example 4 This example models a versatile address book. Some elements are defined in a nested way in the content model declarations of their parent node, while others are declared in the traditional grammar way—using non terminals and a declaring rule.

Named Patterns What corresponds to a complex type in XML Schema (see section 2.4.2), is called *named pattern* in Relax NG. A named pattern is a regular expression, or a part of a regular expression, which name, i.e. non terminal, can be used when defining content models (see example 5).

```
element card {
        name ,
        element phone{ text } ? ,
        element address{ text } ?
} *
name = ( firstName , lastName ) | ( companyName )
firstName = element first{ text }
lastName = element last{ text }
companyName = element company{ text }
}
```

Code Example 5 The card declaration of example 6, but with the content model part declaring valid use of names factored out as a named pattern.

To retain regularity of the content model, it is not possible to recursively refer a name in it's declaration. Recursion is neither allowed using intermediate named patterns.

Interleaving While XML DTD had very poor support for the modelling of unordered data, Relax NG provides a way of modelling an unordered sequence of elements. In example 6, an address book card was modelled as a sequence of a name, phone number and address element. While the schema dictates an order on those elements, there is no conceptual necessity for an order in this case. When modelling the content as shown in example 7 using the interleaving operator, the elements can be given in arbitrary order. Surprisingly, the interleaving construct was available in XML DTD's predecessor—in SGML DTD's. The feature has been removed in XML DTD for the sake of simplicity of implementation. Note, that XML Schema also provide a way of modelling interleaved content.

```
element card {
        ( name | companyName ) &
        element phone{ text }? &
        element address{ text }? &
}
```

Code Example 6 A variation of the address card for the address book in example 6—this card requires all elements, but no order is dictated on them.

While interleaving is used to model unordered sequences of elements, some attention has to be paid to the way it s used in regular expressions, as the interleaving is just applied to sibling expressions in a regular expression. This is best explained along an example: let's assume, we have to model an unordered sequence of `A`, and `B` and `C` elements, or, instead of `B` and `C`, a `D` element. This is nicely expressed as `A & ( (B & C) | D)` . Unfortunately, this expression does not fulfill the requirement, as the sequence `B A C` is not an instance of the given expression. This is due to the fact, that each sub expression has to be fulfilled by a part of the word, i.e. there is no part of the word valid with respect to the sub expression `(B & C)`. In this case, an expression fulfilling the requirement exist: `(A & B & C) | (A & D)`.

Attributes An arguably special feature of Relax NG (compared to DTD and XML Schema), is the equal treatment of attributes and elements in content model declarations Attribute occurrence declarations get hence part of the regular expression, as shown in example 7. The advantage of this approach is, that the occurrence of an attribute can be tightly coupled to the occurrence of an element, which is hardly possible, when elements and attributes are declared separately. Further, the modelling of attribute sets

can benefit from the full possibilities of regular expression modelling, like optionality and disjunctive modelling.

```
element addressBook {
  element card {
    (element name { text }
     | attribute name { text }),
    (element email { text }
     | attribute email { text })
  }*
}
```

Code Example 7 Another variant of the address book example—in this case cards have name and email content, if this is modelled as attribute or element is left to the instance author. The author may even model e.g. the name as attribute while modelling the email address as element of the card-element.

The advantages are unfortunately also tightly coupled with disadvantages: (1) Content modes can be modelled enforcing multiple occurrences of the same attribute name, which is invalid in XML (e.g. using the Kleene star). (2) while a regular expression of elements is modelling an ordered sequence of elements with the known regular expression semantics, the attributes do not occur at the positions as modelled in the content models, as they have their fixed position in the XML syntax—inside of the opening tag of the containing element. Practically, the problems turned out mostly irrelevant, the choice of mixing element and attribute content declarations turned out to be a pragmatically useful decision.

Data Types Relax NG allows patterns to reference externally-defined data types. Relax NG implementations may differ in what data types they support. Only those data types supported by the used Relax NG implementation can be used. The most commonly used data types are those defined by W3C XML Schema Data types [59]. The data type flexibility is arguably god and bad—maximum freedom, in this case of data types, is forward looking in an open system as the web is, on the other hand portability of Relax NG schemata using data types is questionable.

Apart of externally defined data types, enumeration data types are included (see example 8). An enumeration of valid strings can be defined as a data type.

```
element card {
  attribute name { text },
  attribute email { text },
  attribute preferredFormat { "html" | "text" }
}
```

Code Example 8 An extension of an email address card, as defined (among others) in example 7. The email address card now has an attribute for specifying the preferred formatting of emails for the recipient. This is made using an enumeration consisting of two options: "text" and "html".

Relax NG Example Relax NG is considered to be a clean, formally well founded, alternative to XML Schema. The W3C, inventor of XML Schema, itself uses the *"competitor"* for various normative standards as modelling language, i.e. XHTML 2.0 [55] (see appendix B and C of [55]). A short Relax NG grammar is shown in example 9,

```
grammar{
university = element university{ student+ }
student    = element student{
        element name{ nameType } ,
        element matriculation-number{ xsd:integer } ,
        element semester{ xsd:integer{ minInclusive=1 maxInclusive=24 } } ,
        element exams{
                element exam{
                        attribute lecture{ xsd:integer },
                        attribute grade{ text}
                }*
        }
}
nameType = (element first-name{ text } , last-name{ text })
}
```

Code Example 9 For comparability, the XML Schema example presented formerly (see section 2.4.2), is presented here as Relax NG example. The XML Schema Data types Data types are used in this grammar, which is a possible and existing way to integrate data types in Relax NG.

Limitations As mentioned, basic data types are not covered in Relax NG itself. To obtain portable grammars, they can hence not be used.

Another, by now in all schema languages present, limitation, is the lack of typed graph support. Without external data type library, Relax NG does not even support ID/IDREF based references, is hence not able to model graph structured data at all.

Unordered content is partly supported by the interlacing paradigm. As shown above in the paragraph about interlacing, patterns using interlacing are not modelling unordered data in an homogeneous way, sometimes *"chunks"* of content may not be interlaced. This happens, when combining the interlacing operator and the other operators. It is likely, but unproven, that any unordered content model with variable multiplicity, disjunction and arbitrary symbols, can be modelled using regular expressions with interleave operator. However, it is highly demanding to the schema author and error prone, possibly with exponential size complexity in the size of the number of symbols in the worst case.[6]

2.5 Querying The Web

Note, that *Web Querying* is to be distinguished from *Web Searching*: While Web Searching means the use of search engines like `http://www.ask.com` or `http://www.msn.com` and Web directories like `http://dmoz.org/` or `http://www.yahoo.com` by typing in textual search term or by browsing search structures, Web Querying means writing programs in Web Query languages very much in the spirit of writing database queries in SQL.

2.5.1 XPath

With XPath [48] one can localises (sets of) nodes within the tree associated with an XML document, i.e. elements, attributes, and text. XPath has similarity with path expressions as used in Shell or command line operating system environment, as e.g. the Bash on Unix derivations or the `command.com` on Windows, however XPath is also inspired by so-called *"regular path expressions"* first introduced in the

[6] The assumption is unproven. It is based on the recognition, that factorization in the problem formulation has to be unfolded, which means copying the expanded symbol as often as factors occur.

content of the XML query languages XML-QL [24] and Quilt [20], a predecessor of the current XML Query language XQuery [63].

XPath [48] uses so-called “axis specifiers” and “node tests”. The axis specifier, e.g. descendant or child, specify the traversal of the document tree. A node test correspond to a label in a regular path expression. The axis specifiers give rise to a navigation in all directions (to the leaves, to the root, to nodes preceding or following in "document order") the context node.

```
<addressBook>
 <card name="Sacha">
  <phone>21809339</phone>
  <email>sacha.berger@ifi.lmu.de</email>
 </card>
 <card name="Francois">
  <phone>21809310</phone>
  <email>bry@ifi.lmu.de</email>
 </card>
</addressBook>
```

Code Example 10 An XML document representing an address book with phone numbers and email addresses. The example is used in section 2.5.1 to illustrate XPath expression usage.

As an example, consider the xpath expression `descendant::email[position()=1]` to be applied to the example document 10. The result is the element

```
<email>sacha.berger@ifi.lmu.de</email>
```

in line 4—the `descendant` axis indicates to search from the start element (in this case the root element) at arbitrary depth. The elements selected along the descendant axis are restricted to those fulfilling the name test for `email`, but from those elements, only those fulfilling the conditions in square brackets are taken. As the condition restricts the result to the first element in the sequence of intermediate results, the result is as shown above the first `email` element.

XPath has been conceived as an embeddable selection language, not as a whole query language. It lacks features for projection and construction of data, solely selection of data from a given context is covered. Prominent languages incorporating XPath as selection language are XSLT (see section 2.5.2) and XQuery (see section 2.5.3).

2.5.2 XSLT

XSLT [47] is a language for transforming XML documents. It has been conceived as a part of XSL [50], the e**X**tensibme **S**tylesheet **L**anguage. XSLT is a (sort of) functional language with XML syntax. An XSLT program consist of a set of rules expressing how to transform elements in an input document to elements of an output document. XSLT embeds XPath in two contexts: (1) elements from the input documents are transformed by the rule matching it’s XPath expression given as its *match* expression, (2) in rules the transformation of further content is applied recursively on the elements matching the so called *select* expression of the application. Consider example 11 for a simple XSLT program:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

   <xsl:template match="/addressBook">
     <html><body>
       <table>
         <tr><td>NAME</td>  <td>PHONE</td>  <td>EMAIL</td></tr>
         <xsl:apply-templates select="card" />
       </table>
     </body></html>
   </xsl:template>

   <xsl:template match="card">
     <tr>
       <td><xsl:apply-templates select="@name" /></td>
       <td><xsl:apply-templates select="phone" /></td>
       <td><xsl:apply-templates select="email" /></td>
     </tr>
   </xsl:template>

</xsl:stylesheet>
```

Code Example 11 An XSLT program transforming address book documents, as e.g. seen in example 10, into an HTML document with an address book table.

The example 11 consists of two rules, called templates in XSLT and represented by `template` elements. The first rule matches `addressBook` elements and constructs an HTML document with a table with the first row as header row with fixed text columns. Then, the program is applied to all `card` elements that can be found in the given context. The selected `card` elements are transformed using the second template, as this one matches with `card` elements. The second template constructs table rows for the `card` element it is applied to. For the document in example 10, the resulting HTML document is shown in example 12.

```
<html><body>
  <table>
    <tr><td>NAME</td>  <td>PHONE</td>  <td>EMAIL</td></tr>
    <tr>
      <td>Sacha</td>  <td>21809339</td>  <td>sacha.berger@pms.ifi.lmu.de</td>
    </tr>
    <tr>
      <td>Francois</td>  <td>21809310</td>  <td>bry@pms.ifi.lmu.de</td>
    </tr>
  </table>
</body></html>
```

Code Example 12 The result of transforming the document in example 10 using the style sheet in example 11.

Limitations A rather surprising limitation for users of functional programming languages is, that templates may **never** be applied to content that has been constructed, templates are always applied to selected content from the input document. Common paradigms, like stepwise refinement of results, are not applicable to XSLT. The computation is hence solely driven by the input document. Nevertheless, there is evidence, that the computation model of XSLT is Turing-complete [46].

Further features of XSLT Beside the base concepts—template based content modelling and controlled template application—further features help writing reasonably compact and understandable transformation programs:

Conditionals An `if` and a `choose` construct are used to model conditionals similar to *if-then-else* and *case-switch* in traditional imperative programming languages like C or Java. A test, expressed as XPath expression is performed on the current context node, if the test selects *something*, it is considered successful, and the content enclosed by the conditional is processed like regular template content.

Imperative loops Beside `apply-templates`, which iterates over the selected content, the `for- each` construct can be used to loop over sequences of selected elements. The child content of the `for-each` element is then treated like regular template content for each iteration step. The main difference to `apply-templates` is, that no template selection takes place, i.e. the selected node is applied to the child content of the `for-each` element.

Construction Constructs exist, to construct elements and/or attributes, if e.g. their name has to be computed at run time

Projection Using the `value-of` construct, it is possible to project parts of the selected data in the result document. This partly already happens by default, i.e. if no template catches text nodes and `apply-template` is called on them, they are inserted in the result document. `value-of` may also be used to calculate using built in functions for string processing and numerical calculations. Another projection construct is `copy`, which copies, either deep or flat, a selected node (with or without it's attributes).

Variables & Parameters Variables can be used to store and reuse selected content. In the spirit of functional programming languages, the binding of a variable may not be altered within the invocation context of it's containing scope. The scope of a variable may be a template, child content of conditionals or of `for-each` loops. Parameters are like variables of a template, that can be bound at application, i.e. in the `apply-template` construct, very much like function parameters bound at function application. Note, that it is not possible to bind variables or parameters to the result of constructing new content, variables can just be bound to selected input document content. This is related to the fact, that templates may just be applied to content from the input document(s), but never to the constructed content.

Transformation vs. Query Common definitions for query languages claim, that query languages consist of selection, construction and projection of data from a given data model to (possibly) another data model. All those properties are fulfilled by XSLT, rendering it hence by features a query language as well as a transformation language. Indeed, the difference between transformation language and query language is vague. XML has been proposed by the W3C as data (meta) model to be followed by styling, transformation and query language proposals. The transformation language XSLT was proposed before a query language. The versatility of XSLT lead to a not so surprising reaction of it's 'affiliados': when the query language XQuery was proposed by the W3C, it's reference implementation was suggested to be realized in XSLT, while questioning the necessity of a distinct query language at the same time.

2.5.3 XQuery

The fact that with XQuery [63] and XSLT there are two expressive XML *processing* languages that can be used for mainly the same purposes is somewhat surprising. While XSLT was developed by the

‘document community’, XQuery is an XML Query language originating from the Database community, which can already be observed by its syntactical similarity to SQL. XQuery, also called XML Query, is derived from a query language called Quilt [20]. Quilt had been conceived by Jonathan Robie (Software AG), Daniela Florescu (INRIA) and Don Chamberlain (IBM Almaden Research Center) as a query and transformation language for XML and Semistructured data integrating most of the ideas of prototype query and transformation languages designed since the mid 90th – hence the name Quilt.

Like Quilt and XSLT, XQuery uses XPath for locating nodes and/or text in the input data, like XSLT, XQuery is a sort of a functional programming language. Indeed, a plain XPath expression is syntactically already a valid XQuery program—returning as result exactly the content selected by the XPath expression from a context (e.g. document) passed (e.g. as command argument) to the XQuery implementation or run-time environment.

```
//card/@name
```

Code Example 13 An XQuery program in form of an XPath expression to be applied to example 10—the result is all the text of all the `name` attributes given to the `card` elements.

As the result of the example 13 already indicates (plain text is returned), the result of an XQuery program is not always an XML document. The result may be an XML document, plain text, or a sequence of XML nodes (i.e. text nodes, attributes, elements, ...). The sequence of XML nodes is an important concept, used by many language primitives and functions in XQuery.

FLOWR-Expressions The main language construct in XQuery is the so called FLOWR expression (pronounced “flower” expression). FLOWR is an acronym for *For-Let-Orderedby-Where-Return*. Explained on example 14, all parts of the FLOWR expression have been used. However, it is possible to omit any parts in the expression (i.e. the *for*, the *let* etc.)

```
FOR $p in /addressBook/card/phone
        LET $c := $p/..
        WHERE not($c/@confidential = "yes")
                ORDER BY $p/text() ASCENDING
                RETURN <card phone="{$p/text()}">
                               <name>{$c/name/text()}</name>
                       </card>
```

Code Example 14 In this example a FLOWR expression is used to construct an “inverted phone list” for non-confidential phone numbers, where the card element contains the phone number as attribute (and the cards are ordered ascending with respect to the phone numbers) and a name element is a child node of the card. Information about confidentiality is given in the form of an attribute for the `card` elements.

FLOWR expressions may also be nested, giving rise to more complex construction, especially those, where one level of iteration is no enough. In example 15, a 2-dimensional structure, i.e. the result of the XSLT example 11 (namely the document in example 12), is queried—an HTML table is the input document and as a result a rotated HTML table is constructed, i.e. a table where the columns of the input are the rows of the output and vice versa. For each dimension (first row, then column), one level of iteration is needed.

```
LET $table := $htmlDocument//table
RETURN        <table> {
                FOR $x in $table/tr[1]/td/position()
                RETURN         <tr> {
                                FOR $tr in $table/tr
                                RETURN
                                        FOR $td in $tr/td
                                        WHERE $td/position() = $x
                                        RETURN $td
                                } </tr>
                } </table>
```

Code Example 15 An example *rotating* an HTML table. Note, that for simplicity it is assumed, that the table has rectangular shape, what is not enforced in HTML.

Functions XPath queries or parts of it can be organized in functions. Indeed, some functions are even predefined. In contrast to XSLT, functions can construct new XML content that can be queried or processed in the same program, e.g. in another function. Example 16 shows how a function is used to construct data, that is then decomposed by another query. This approach is reasonable e.g. to provide an interface to versatile data formats, i.e. hiding the complexity of information selection and projection.

```
DECLARE FUNCTION harvestData( $url ) {
        FOR $row IN document($url)//table/tr[position() > 1]
        RETURN <data>
                     <name>{$row/td[1]}</name>
                     <phone>{$row/td[2]}</phone>
                     <email>{$row/td[3]}</email>
                   </data>
}
<addressBook>
        FOR $d in harvestData( "http://example.com/addresses.html" )
        RETURN <card name={$d/name/text()}>{
                     IF ( $d/phone/text() )
                             THEN <phone>{$d/phone/text()}</phone>
                             ELSE ()
                     IF ( $d/email/text() )
                             THEN <email>{$d/phone/text()}</email>
                             ELSE ()
                   }</card>
</addressBook>
```

Code Example 16 The example is based on data as shown in example 11, an HTML table of name-, phone-, email- triples. A function, called `harvestData` is used to retrieve the knowledge. The query is then constructing an `addressBook` document in the spirit of example 10. Note how, in contrast to common programming languages like Java, the *return*-statement is iterated—this results in a sequence of `data` elements.

Types XQuery is a language with an *"optional type system"*, which means, that (1) availability of the type system in implementations of the XQuery specifications are optional, and (2) the use of types are optional for the user. Optional types for the user is in this context not to be understood as optional type annotation of generally typed programs due to the availability of type inference as e.g. in Haskell—untyped program fragments exist alongside with typed ones. Typed fragments of XQuery are type annotated with XMLSchema type information, which may be type names

XQuery (when typed) comes along with a static or a dynamic type system, when statically typed, it also needs to be dynamically typed. Types are based on concepts like

- structured- or textual content of any type—any of the known types, sub elements of a structured element may be given again
- structured content of unknown type—no assumption at call can be made on the type, this corresponds to a most general type
- structured- or textual content conforming to some XML Schema type declaration.

Functions, variables and XPath selections may be typed in XQuery. If an XQuery implementation is statically typed, a type 'error-free' program must lead to the same result as a the execution of the program in a dynamically typed implementation, i.e. static type information cannot lead to a different result than dynamic type information.

Further Features of XQuery Beside FLOWR expressions and functional abstraction, further features help writing reasonably compact and understandable queries:

Conditionals Based on boolean operations and XPath expression evaluation (to empty or non-empty selection), *if-then-else* conditionals can be formed as in common programming languages. Surprisingly, while the language description [63] puts lots of emphasize on FLOWR expressions, the evaluation of XQuery [64] is based on rewriting programs to an (arguably) minimal subset of XQuery and XPath mot containing FLOWR expressions anymore, but deeply nested *if-then-else* expressions.

Modules Complex XQuery modules can be split up in modules, giving rise for reuse of functions in various programs. A query is then composed of exactly one main module and arbitrary many library modules. A main module may contain a FLOWR expression at "root" level, i.e. not in the context of a function, and any function and variable declarations, a library module may only contain function and variable declarations.

Limitations The expressiveness of XQuery gives rise to arbitrary computation, i.e. Turing completeness does not leave lots of space for complains about expressiveness.

The downside of the high expressiveness, is the difficulty to provide highly optimized query evaluation. XQuery was initially conceived as an (XML) database query language with potential for optimization—query optimization is an important field of database development. One source of problems is the so called *nested querying*, where e.g. inside an iteration for result construction another sub-query (again possibly with iteration) occur. Nested queries are unfortunately the way of choice to implement complex grouping in XQuery. While nested querying gives rise to writing highly hand-optimized code to the program—he gets full control over the evaluation order of the query—automatic optimizations over the nesting levels is very hard, maybe impossible in general. In many cases, more high level constructs, e.g. grouping-, ordering-, duplicate elimination constructs, etc., would make the use of nested queries unnecessary, giving comfortable tools to the programmer and yielding rise to automatic optimization for XQuery run time engines.

Another limitation of XQuery lies in its specification relating XPath: some parts of XPath (i.e. most reverse axes) are optional in implementation. This does not restrict the expressiveness, as the *missing* axes can all be simulated using other axes and constructs, but on the other hand, the user is encumbered in the use of XPath when trying to write portable XQuery programs. As it has been shown how to rewrite XPath expressions involving reverse axes, the rewriting could have been included in the specification as a must for implementations without native support of the axes in question.

2.5.4 Xcerpt

Xcerpt is a Web and Semantic Web query and transformation language currently in development at the Institute for Informatics of the University of Munich [43] [18]. The main claim is, that the use of arguably declarative programming language paradigms ease writing of programs, as well as efficient evaluation with less need to consider optimization.

Xcerpt is conceived to query and construct XML data in special, and Semistructured Data in general. Xcerpt has an own (syntactical and semantical) data model for Semistructured Data called Xcerpt data terms (see example 17 for an example). Xcerpt Data terms are able to represent tree shaped data with attributes, textual- or structural nodes, ordered or unordered sequences of child nodes, as well as graph shaped, directed, node labelled structures with a dedicated *root node*. Graph shaped documents are syntactically represented as spanning trees with references to nodes of the tree.

```
bibliography{
        a1@author{ name["Eric","van der Vlist"],
                        publications{ ^b1, ^b2 } },
        a2@author{ name["Jean-Jacques"," Thomason"],
                        publications{ ^b2 } },
        b1@book{ "RELAX NG" ,
                      authors[ ^a1 ] } ,
        b2@book{ "XML SchÃĭma (Ãĭdition franÃğaise)" ,
                      authors[ ^a1,  ^a2 ] }
}
```

Code Example 17 An example of an Xcerpt data term, representing a bibliographic database. Nested elements are represented in term syntax, where the use of square brackets indicate, that the sibling order of the nested elements is relevant, while curly braces indicate, that the sequence of child elements can be considered to be irrelevant. Some elements are referable using an identifier, prefixed to them with a trailing *'at'*-sign. Referable objects may be referenced using their identifier, prefixed by an *'hat'*-sign. Textual content is quoted text and can occur along with other elements.

The general structure of an Xcerpt program can be compared to a program in logic programing languages or to a database query in Datalog. The main syntactic concepts along with their informal explanation of Xcerpt are as follows:

- A program consists of one to many rules that may use each others result in the construction of new results.
- A rule consists of a query part clearly separated syntactically from a construct part.
  - The query part is able to query external data on the web or internal data, constructed by other rules.
  - The construct part is able to construct result data to be used as output of a program or as input for other rules to query
  - Query and construct parts of a rule are connected using logical variables, variables are (exclusively) bound in the query parts and multiple variable bindings are possible,like in the spirit of variables in Prolog or Datalog.
- Query parts consist of so called query patterns, which are syntactically an extension of the queried data model (i.e. XML or Xcerpt data terms). The extensions are (1) incompleteness constructs to leave space for incertitude about shape, size and multiplicity of the queried data structures and (2) variables to select data fragments from the input data.

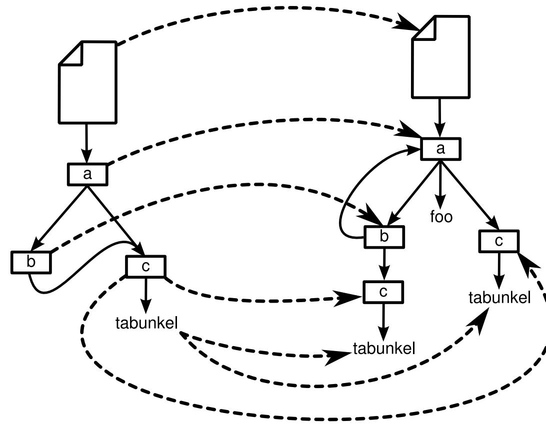


Figure 2.5: The left graph structure is simulated by the right one—for each node of the graph on the left hand side graph, there is a corresponding node on the right hand side graph, such that (1) the labelling is equal and (2) the nodes following the node in question on the left hand side graph are simulated by nodes following node in question in the graph on the right hand side. Textual nodes on the left hand side are simulated by textual nodes on the right hand side. It is possible that one node on the left hand side is simulated by many nodes on the right hand side (i.e. when the left hand side node has multiple incoming edges).

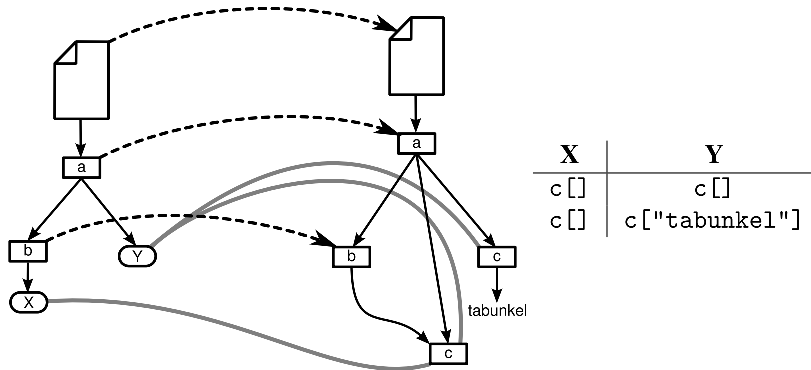
- Construct parts of rules are so called construct patterns, a syntactic extension of XML or Xcerpt data terms. The extensions are (2) variables, that are replaces by their bindings at program evaluation time to construct data, (1) aggregation and grouping constructs to aggregate and group multiple variable bindings.

The evaluation of an Xcerpt rule consists of two phases: (1) the evaluation of the query, resulting in a so called substitution set for the variables, (2) the evaluation of the construction, resulting in data terms due to evaluation of aggregation and grouping constructs and substitution of the variables according to the substitution set gathered in the query part evaluation.

Xcerpt Rule Query Evaluation is done using the so called *simulation unification*, a non standard unification algorithm based on the *simulation preorder* of graphs. First, simulation preorder and it's properties are briefly presented, then the focus will pan back to simulation unification, which yields a set of substitutions, the result of Xcerpt query evaluation.

Speaking informally, a graph simulates another graph, if each state in the second graph is simulated by a state in the first graph. A state in the first graph simulates a state in the second one, if each adjacent state of the state in question in the second graph is simulated by an adjacent state to the state in question of the first graph. For labelled or attributed edges or nodes or plain textual nodes, equal labelling, attributing or text equality can be requested on simulating states or on the traversed edges (see figure 2.5 for an example of the simulation relation between two graphs with node label and textual nodes).

If simulation preorder holds in both directions for two graphs in question, it is called *Bi-simulation*. Bi-simulation is a kind of intuitive equality between two graphs, when the graphs are considered e.g. to be automata—both automata "behave" the same way, if behaviour of automata is seen as a kind of graph traversal. However, Bi-simulation is a weaker equality relation than graph isomorphism for an example of the difference, consider figure 3.1 in section 3.3.2.



X	Y
`c[]`	`c[]`
`c[]`	`c["tabunkel"]`

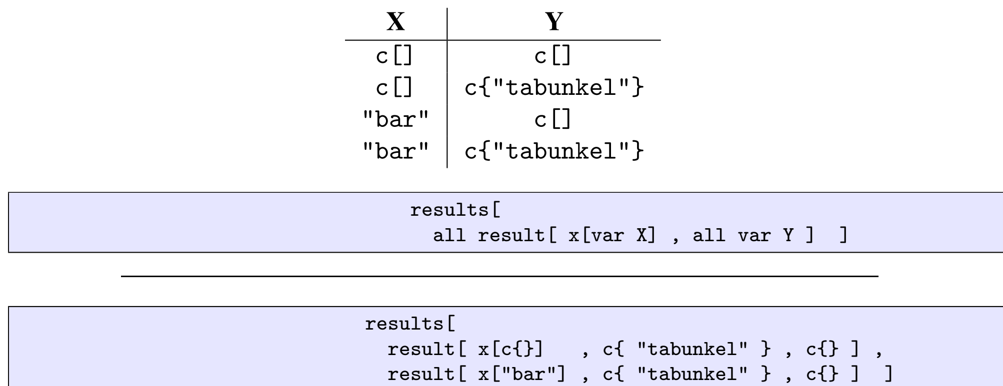
Figure 2.6: A demonstration of simulation unification, where the left graph (serializable as the Xcerpt query term `a{{ b{ var X } , var Y }}`) (in this case tree shaped) has two variables. Possible bindings for the variables are indicated by grey lines, when substituting the variables by the corresponding nodes from the right hand side (a data graph represented by the Xcerpt data term `a{ b{ ∧c1 } , c1@c{} , c{ ‘‘tabunkel’’ } }`), the graph from the left hand side is simulated by the one on the right hand side. The result of simulation unification is a set of unifiers, shown as a table on the right hand side—each substitution is a row in the table, each column represents a variable.

Focusing back on *simulation unification*, the second graph in the explanation of simulation preorder may now contain logical variables as nodes. The second graph is the query pattern, while the first one can be considered to be a data instance. A simulation unifier is now a mapping of variables to sub-graphs of the second graph, such that, after substituting the variables in the second graph with the sub-graphs in question, the first graph simulates the second one. Possibly multiple such unifiers exist, the result of simulation unification is the set of all such unifiers. Figure 2.6 illustrates simulation unification by example.

Incompleteness in breadth (i.e. partial adjacent node sequence specification) is already addressed by the default simulation preorder, as there is just stated, that all states in the second graph have to simulate in states in the first graph, nothing is stated about all states in the first graph. Hence, completeness (in breadth) can be considered as a side condition, about the states in the first graph. Incompleteness in order is addressed by standard simulation preorder, as no conditions about the order of the simulated states is defined in common graph models, implementing ordered simulation is hence a condition about the order of the simulated states in addition to default simulation preorder. This requires some extension like numbered edges in the graph model.

Xcerpt Rule Construct Evaluation is evaluated with a set of substitutions (i.e. the result of an Xcerpt query part evaluation) is input. In the simplest case of a construct term, a construct term without variables, nothing more than the construct term interpreted as data term is returned, there are as many results as there are substitutions in the set, all identical. For a construct term with variables (and without grouping constructs) for each substitution the resulting data term is obtained, by substituting the variables in the construct term with the data terms associated to the corresponding variable in the substitution. Note, that all variables in the construct term need to be substituted, otherwise the whole rule is considered ill formed—this can already be assured at compile time, as it simply means, that each variable in the construct part has also to occur in the query part of the same rule.

For the evaluation of construct terms with grouping (and/or aggregation), the scope of the vari-



X	Y
c[]	c[]
c[]	c{"tabunkel"}
"bar"	c[]
"bar"	c{"tabunkel"}

```
results[
  all result[ x[var X] , all var Y ]  ]
```

```
results[
  result[ x[c{}]    , c{ "tabunkel" } , c{} ] ,
  result[ x["bar"] , c{ "tabunkel" } , c{} ]  ]
```

Figure 2.7: The application of a set of substitutions (in form of a table) to an Xcerpt construct term (shown above the line) is demonstrated here. The result is the Xcerpt data term as shown below the line.

ables with respect to the grouping constructs is relevant: all construct terms in the scope of a grouping construct are sequentially constructed by substituting their variables, called the variables bound by the grouping construct in question. Not all substitutions are used at once in the constructed of the terms in the scope of the grouping construct: the grouping is performed for each subset of the substitution set with equal free variable substitution, the free variable substitutions in this subset then are used in just one construction. For each such subset, the grouping is evaluated, resulting in the end in as many instantiations of the inner most grouped variables, as there are substitution—just that those instantiations get grouped in different contexts. Consider example 2.7 as a demonstration of how a substitution set is applied to a construct term.

Xcerpt Rule Evaluation. After introducing Construct- and Query evaluation, not a lot is left to say about the evaluation of an Xcerpt rule—evaluation the rule consists of evaluating the query part, also called body of the rule, and feeding the resulting substitution set to the construct part, also called the head of the rule, to obtain a set of data terms as result. A query part in a rule may be adorned by an `in`-construct, stating that the query is to be evaluated against an external resource, given as argument of the `in`-construct. Otherwise the rule is to be evaluated against the set of data terms constructable by all the rules in the program. Consider figure 2.8 as an example for rule evaluation.

A special rule is the so called *goal*. A goal does not contribute to the set of data terms constructable by the program, it's result s considered to be the result of evaluating the whole program. As the result has to be a data term or an XML document, just one data term may be instantiated by the goal, it is hence likely, that the head of the goal uses grouping in sch a way, that there are no free variables.[7]

Xcerpt Program Evaluation The evaluation of an Xcerpt program is the chaining of the evaluation of the rules. The goal rule gives the result, the rules attached using the `in`-construct to external resources get input data (which is not forcibly necessary, data can also be defined internally). All the other rules query the internal program store and feed results in this store. The operational semantics of chaining can be define in two ways, there is no commitment about it by now, possibly both ways will be available

[7]The head of a goal may have free variables, it s not defined which resulting element is returned by a program constructing multiple outputs due to more than one substitution being applied during program evaluation to the goal's head.

```
CONSTRUCT
  html[head[title[  "Hyperlinks on www.google.com"  ] ],
    body[
      all (a(href=var HREF)[ all var CONTENT ] , br[] )
    ]
  ]
FROM
  in resource("http://www.google.de")(
    html[[ body[[ a(href=var HREF)[[ var CONTENT]] ]] ]]
  )
END
```

```
<html>
  <head><title> Hyperlinks on www.google.de </title></head>
  <body>
    <a href="/url?sa=p&amp;pref=ig...">In Englisch</a>
    <br />
    <a href="https://www.google.com/accounts/Login..."> Bei Google Anmelden</a>
    <br />
    ...
  </body>
</html>
```

Figure 2.8: Above the line an Xcerpt rule is given. It is conceived to retrieve all the hyperlinks from the start page of Google and to present them in a plain HTML document. The result of evaluating the rule is shown below the line—this time not in Xcerpt data term syntax, but in XML (and hence XHTML) syntax.

as choice for the programmer. In example 18 an Xcerpt program is presented.

```
GOAL
  TheResultOfTheCalculationThreePlusTwoIs[ var Result ]
FROM
  plus[ three[] , two[] , var Result ]
END

CONSTRUCT
 var NAT
FROM
 in resource "http://example.com/naturalNumbers.xml" (
  naturalNumbers{{ var NAT }}   )
END

CONSTRUCT
 plus[ var X , zero[] , var X ]
FROM
 succ[[ var X ]]
END

CONSTRUCT
 plus[ X , Y , RESULT ]
FROM
 and(
  plus[ X , PRE_Y , PRE_RESULT ] ,
  succ[ PRE_Y , Y ] ,
  succ[ PRE_RESULT , RESULT ]
 )
END
```

Code Example 18 This example illustrates an Xcerpt program for the addition of natural numbers very much in the spirit of it's mathematical definition. First, natural numbers are defined as a set of elements and a binary successor relationship. The natural numbers are considered to be in an external document (theoretically of infinite size, but for the sake of concreteness, an upper bound can be assumed). The document hence looks somehow like this: "`naturalNumbers{ succ[zero[] ,one[] ], succ[ one[] , two[] ] , ... }`". The Program consists of a goal returning the sum of three and two. The addition is defined as a ternary relation between the first and second operand and the result. The definition is recursive and based on the successor relationship and the base case $x+0=x$.

The first kind of chaining, called forward chaining, is driven by the input data. The rules attached to the input data are evaluated first and initially feed the internal store. The other rules are then applied until saturation is reached, i.e. no further rule application to the internal store produces content, that is not already in the store (the store can be seen as a set of data terms). Then, the goal is evaluated against the store and the result is returned. This type of chaining s very much in the spirit of Datalog. The advantage is, that, even without focusing on a goal, all knowledge is pre-processed and rapidly accessible for various possible goals.[8]

The second chaining approach is called backward, or goal driven, chaining. When Xcerpt is implemented using goal driven chaining, the query term f the goal is evaluated against the heads of all rules, using a slightly extended version of simulation unification, able to unify a query term with a construct term. The rules, of which the simulation unification with the head provided non empty substitution sets, are then recursively evaluated the same way as the goal (i.e. by simulation unification with other rule

[8]Note, that an Xcerpt program always contains only one goal, so the pre-processing can be compared to a system setting, where programs are seen as queries to a database system with materialized views.

heads). In substitution sets produced by this kind of simulation unification, variables may not only be mapped to data terms, but may also be mapped to construct terms containing variables. When a branch in the recursive process reaches an end, the substitution sets are then applied along the branch to the goal in a similar way to applying them to a single construct term.

In comparison to forward chaining, backward chaining may calculate less 'useless' facts, as just facts that could contribute to the result are evaluated. The approach is especially useful, when the set of facts to reason about or the data to query is very big, possibly infinite, but the query is very selective. On the other hand, the backward driven approach has no mean to store intermediate results, called *materialized vies* previously in the context of forward chaining. Further on, some recursive programs on finite data can trap the evaluation in an endless loop, as there is no inherent recognition of construction of duplicates, as in the set concept of the store in forward chaining.

Further Constructs of Xcerpt are presented now. They are considered *"further"*, because they are not essential for understanding Xcerpt's paradigm. Some of them can be replaced using the base constructs, they are useful to ease programming or to ease efficient evaluation due to optimization of higher level constructs.

Xcerpt Queries may contain

Descendant Sometimes it is desirable to state, that a certain sub-term *may* occur as the child term of it's parent, but that it may also be a grand-child, grand-grand-child or any descendant. As a practical example, consider the case of searching for hyperlinks (like previously when introducing Xcerpt rules, consider example 2.8) in a Web-page: actually hyperlinks may occur at any nesting level (e.g. nested in style conditions, tables, etc.). To capture really all hyperlinks, the *descendant*-construct is useful (see example 19. A term can be adorned by the leading `desc` keyword, to indicate, that the desired sub-term may occur at any depth in the document starting from subterms position.

```
html[[ desc a(href=var HREF)[[]] ]]
```

Code Example 19 This query term can be considered as a replacement of the query term in the rule example 2.8. Using this query pattern, not only hyperlinks that are direct children of the `body`-element will be found, but also those nested at deeper level in the document structure.

As The *as*-construct is useful, to specify a query-(sub)-term and a variable at the same location. This is useful, to get a variable binding restricted to terms simulated by the query term associated to the variable. As an example (see example 20) consider the case, that in an extension of the Xcerpt rule example (see example 2.8), the list of hyperlinks found in Google should not only retrieve the URLs, but the whole hyperlink (hence the `<a>`-element with all attributes and the anchor text).

```
html[[ desc var HYPERLINK -> a[[]] ]]
```

Code Example 20 This query term is an extension of the term in example 19. It is used to harvest hyperlinks in HTML documents. They are easily found using the `a[[]]` sub-query. Using the *as*-construct, those elements are bound to a variable called `HYPERLINK`.

Optional A sub-term of a query can be marked as optional pattern. An optional pattern simulates *if possible*—if the queried data contains appropriate sub-terms, they will be simulated by the

optional term, providing corresponding variable bindings for variables occurring in the optional part, if no simulation is possible for the optional (sub)term, but for the non-optional parts, the query term simulates as well. Assuming, that variables are in the scope of an optional term not simulating data, the variables will be bound to a default value, that can be given in the program. Example 21 presents a query term with optional parts.

```
university{{
  student(name=var NAME){{
    optional grade{ var GRADE with-default "No Grades by now" }
  }}
}}
```

Code Example 21 To query a university database for all the students with the grades they received, the following query is useful. Assuming, that not all students achieved grades, the variable `GRADE` is only to be bound to data, if grades are available. In the other case, a default binding is given for the variable, the text `''No Grades by now''`.

as-construct. It is hence also called *variable restriction*.

Negation A (sub)-query term may be adorned by *not* or *without*, indicating that the query term containing the negated sub-term only matches, if there is no match for the negated sub-term.

Position Sometimes it is necessary to either specify the position at which a term matched by a query term has to be in it's sequence of sibling nodes. The corresponding query term is attributed with the *position*-construct which gets a number as argument. It is also possible to use a variable as argument, in this case *position* is not restricting the position of the matched sub-term, it binds the variable to the position of the sub-term.

Regular Expression Regular expressions are useful to restrict simulation of textual content, i.e. text nodes. It is also possible, to use regular expressions for element labels, giving rise to patterns matching various, differently named, elements.

Logical Connectors In a query is is possible to specify many query terms, connected with the logical connectors *"and"* and *"or"*. Multiple occurrences of the same variable in a conjunction of query terms may only unify, if they have the same bindings in the simulations of both query terms.

Where Clause The *where*-construct is used to specify restrictions to variables, that are not of structural nature or not related to the query term. This can be e.g. restricting the match of a variable to a certain value or fulfilling conditions involving function, like arithmetical functions, for certain variables, or simply expressing relations between many variables.

Part I

R_2G_2

Chapter 3

R_2G_2—Regular Rooted Graph Grammars

For the typing of web query languages, a type declaration formalism is needed. From the point of view of a person willing to use a query language with type support and willing to use the given schema languages as type declarations, the shortcomings presented in section 2.4.1 and 2.4.2 are annoying.

The new type and schema language R_2G_2 is a schema language for tree and graph shaped Semistructured data like XML documents (or Xcerpt data terms). R_2G_2 is formally based on extensions of regular rooted tree grammars and is hence similar to existing schema languages as DTD, XML Schema, Relax NG or the type system of XDuce. New contributions are:

- Handling of graph shaped data based on the simulation preorder of graphs and rational trees.
- Unordered interpretation of regular expressions for unordered content specifications.
- Typed references for XML.
- Modelling of tree and reference based serializations of graph shaped data.

3.1 Regular Tree Grammars

Regular tree grammars describe sets of trees by their shape. They are precisely introduced in [22]. They are syntactically a subset of context free grammars (as shown e.g. in [26]).

As a prerequisite, let there be the definition of Trees over an alphabet Σ.

Definition 3.1
Let Σ be a set of tree node labels and $\Sigma_p \subseteq \Sigma$ all labels of nodes that have p direct child nodes. The set $T(\Sigma)$ denotes the set of all trees that can be constructed using the symbols in Σ applying the following rules:

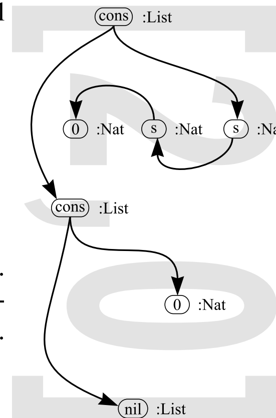
- $\Sigma_0 \subseteq T(\Sigma)$, those nodes will be called leaf nodes.
- for $p \geq 1$, if $l \in \Sigma_p$ and $t_1, \ldots, t_p \in T(\Sigma)$, then $l(t_1, \ldots, t_p) \in T(\Sigma)$

Definition 3.2
A regular tree grammar $G = (S, N, \Sigma, R)$ contains a so called axiom S, a set of non terminal symbols N, a set of terminal symbols or tree labels Σ and a set of production rules of shape $\alpha \rightarrow \beta$ with $\alpha \in N$ and $\beta \in T(\Sigma \cup N)$. Note, that $T(\Sigma \cup N)$ denotes trees where the symbols of N all have arity 0 and the symbols of Σ arbitrary, but fixed arity.

A (tree) grammar is sometimes called a generator for a language (of trees). A grammar $G = ((S, N, \Sigma, R)$ generates the language $\mathcal{L}(G)$ consisting of all tree instances that can be derived starting from the start axiom S of G using the rules R of G. A grammar generates trees, that can be derived, by replacing non terminal symbols—symbols of N—in trees of $T(\Sigma \cup N)$ by a right hand side of a grammar rule of R, if the left hand side of the rule corresponds to the given non terminal symbol. The starting non terminal symbol is always the axiom S. A non terminal replacement step is called a *derivation step*. A valid data tree is generated by a sequence of derivation steps that replace all non terminals in the intermediate tree of $T(\Sigma \cup N)$, i.e. a successful sequence of derivations always yields an instance of $T(\Sigma)$.

Example 3.1

A language of (head/tail encoded) lists of (unary encoded) natural numbers can be modelled as regular tree grammar as follows:
$G = (List, \{List, Nat\}, \{0, nil, s_{/1}, cons_{/2}\}, R)$
where $R =$
$\{\ List \rightarrow nil$
$,\ List \rightarrow cons(Nat, List)$
$,\ Nat \rightarrow 0$
$,\ Nat \rightarrow s(Nat)\ \}$
An instance of the language $\mathcal{L}(G)$ is e.g. $cons(s(s(0)), cons(0, nil))$—this corresponds to the (unary encoded) sequence of (decimal encoded) natural numbers $[2, 0]$.



Regular Tree Grammars for Unranked Trees The presented tree grammar formalism is able to generate ranked tree languages, i.e. tree languages with symbols of fixed arities—in the example 3.1 s is of arity 1, *nil* of arity 0 and *cons* of arity 2. Semistructured data, XML and Xcerpt data terms are data models with trees of variable arity, also called *unranked trees*—a tree of a given type or label may have arbitrary many child trees. All properties of languages of regular ranked trees can be extended to unranked trees, if a special transcription of unranked trees to ranked trees is applied—unranked trees are decomposed in such a way, that the child trees of a tree get sequenced in a list as presented in example 3.1.

To directly model unranked regular trees using a tree grammar formalism two extensions can be sought of:

1. Allow additional grammar rules in the shape $A \rightarrow B, t$ with $A, B\ \in N$ (where N is the set of non terminals) and $t \in T(N \cup \Sigma)$. Further, the symbols in Σ have to be unranked, i.e. allowing sequences of arbitrary many child terms. The additional grammar rules correspond to production rules of *Chomsky type III* grammars and can model sequences of trees.

2. Allow regular expressions of non terminals at the position of the child non terminals in the right hand side of production rules, e.g. $A \rightarrow l[\ re\ ]$ where $A \in N$ and re is a regular expression over N.[1]

The approach with Chomsky type III production rules is arguably syntactically simpler, yet more demanding for the author of a grammar, as it is easy to write non type III rules violating the regularity constraint on the language. Further the sequence sought of is less obvious, as split up in rules and mixed with tree structure rules—the order of element sequences is not automatically reflected by the order of the grammar rules. The regular expression approach will be followed in this deliverable due to the shortcoming of the first approach, yet it is a bigger syntactical extension of regular tree grammars than the Chomsky Hierarchy Type III production rule extension. The regular expression approach is also more established than the pure tree grammar approach.

Definition 3.3
A regular unranked tree grammar $G = (S, N, \Sigma, R)$ contains a so called axiom S, a set of non terminal symbols N, a set of terminal symbols or tree labels Σ and a set of production rules R of shape $\alpha \rightarrow \beta$ with $\alpha \in N$ and β of shape $l[\ re\ ]$ where $l \in \Sigma$ and re is defined as follows:
$re ::= \varepsilon$
$re ::= a \quad a \in N$
$re ::= (\ re\ )$
$re ::= re, re$
$re ::= re|re$
$re ::= re^{+}$
$re ::= re^{*}$
$re ::= re^{?}$

By ε an empty regular expression is denoted, it accepts the empty word. In concrete syntax of regular expression, the ε is omitted.

Example 3.2
A regular tree grammar for apple trees with branches containing arbitrary leafs and apples or other branches (at least two) is presented as an example.
$G = (Tree, \{Tree, Branch, Leaf, Apple\}, \{tree, branch, leaf, apple\}, R)$
where $R =$
$\{\ Tree \rightarrow tree[Branch]$
$,\ Branch \rightarrow branch[(Branch, Branch+)|(Leaf|Apple)*]$
$,\ Leaf \rightarrow leaf[]$
$,\ Apple \rightarrow apple[]\ \}$

3.2 Regular Tree Grammars for Unordered Unranked Trees

In many applications of data modelling, the order of instances in some context is of minor importance. Typical examples are object oriented databases or programming languages, where the order in which the attributes of a class are defined is irrelevant e.g. for their access—the name is the only property for accessing attributes. With XML in contrast, the document centric paradigm always imposes an order of elements in a sequence of child nodes. However, attributes are considered to be unordered information of

[1]The square braces "[]" have been chosen to better distinguish unranked tree grammar rules from ranked ones for which parenthesis "()" are used.

an element, so XML provides a limited[2] concept of mixed ordered and unordered content. An example of XML documents where the order of the elements is of high importance is e.g. an HTML document—headings and paragraphs occur in an order absolutely relevant for the document and ignoring the order of those elements could arguably render the document meaningless.

Many applications of XML are *data centric* and not *document centric*, e.g. more in the spirit of Semistructured databases. For many of such applications, the order imposed by the document is irrelevant. As an example, consider a bibliographic database: the order of the books in the database may be completely arbitrary, e.g. if maybe the order in which the librarian grabbed the books out of the shelves to index them. The relevant information for access of the books is e.g. title, author or any information about the book suitable for indexing. For such an application, arguably the following two databases can be considered to be equivalent yet the XML semantics considers them to be different.

```
<bib>
 <book>
  <title>Data on the Web</title>
  <authors>
   S.Abiteboul,
   and P.Buneman,
   and D.Suciu
  </authors>
 </book>
 <book>
  <title>
   Automata, Languages,
   and Programming
  </title>
  <authors>
   S.Abiteboul,
   and E.Shamir
  </authors>
 </book>
</bib>
```

Code Example 22 First variation of a bibliography with arguably irrelevant order of the `book` elements

```
<bib>
 <book>
  <title>
   Automata, Languages,
   and Programming
  </title>
  <authors>
   S.Abiteboul,
   and E.Shamir
  </authors>
 </book>
 <book>
  <authors>
   S.Abiteboul,
   and P.Buneman,
   and D.Suciu
  </authors>
  <title>Data on the Web</title>
 </book>
</bib>
```

Code Example 23 Other variation of bibliography as seen in code example 22

Xcerpt distinguishes ordered child sequences from unordered child multisets, to be able to realize data for both application paradigms—data centric and document centric—in an appropriate manner. Using unordered Xcerpt content specifications, the former two databases are modelled to be semantically equivalent.

To be able to model unordered content appropriately in a grammar, a dedicated modelling formalism is necessary. With the type checking goal in mind, a formalism needs to be decidable concerning membership test, emptiness check, subset test and it needs to be closed under intersection, as it is motivated in 8.

3.2.0.1 Current Modelling Formalisms for languages of Multisets

Various formalisms for modelling languages of multisets have been proposed of which some will be briefly presented here:

[2]limited, because attributes are arguably less expressive than elements, as they may not contain structured information but only atomic values.

Multiplicity Lists The multiplicity lists [17] are a metalanguage for specifying decidable sets of data terms, which are be used in later work of the authors [66] [14] as types in processing of tree-structured data. The idea is motivated by DTDs and by XML Schema. A multiplicity list is a regular type expression of the form $s_1(n_1 : m_1) \cdots s_k(n_k : m_k)$ where $k \geq 0$ and $s_1, \ldots, s_k$ are distinct type names. Informally, the meaning is, that a multiset of symbols is valid with respect to the given multiplicity list, if the symbol s_i occurs at least n_i times but not more than m_i times. Other symbols than the mentioned s_1 through s_k may not occur.

Multiplicity lists are closed under intersection but not under union and complement. Multiplicity lists have been applied successfully in static type checking of Xcerpt programs as presented in [Descriptive Typing Rules for Xcerpt and their Soundness] and [A Prototype of a Descriptive Type System for Xcerpt.].

$\mathcal{L}$-Formulae Frank Neven and Thomas Schwentick presented the $\mathcal{L}$-Formulae [39] as a decidable formalism for unordered content models on the Web. $\mathcal{L}$-Formulae are defined as $\varphi ::= true \mid false \mid a = i \mid a \geq i \mid \neg\varphi \mid \varphi \vee \varphi$ with $a \in \Sigma$ and $i \in N$. Roughly speaking $a = i$ means, that the symbol a occurs exactly i times in a valid multiset, $a \geq i$ that it occurs at least i times and so on.

$\mathcal{L}$-Formulae are closed under intersection, complement and union. When typing or type checking semi structured data queries in general and Xcerpt programs in special, an interesting property emerges when querying ordered data without caring of the order, i.e. querying the ordered data using order agnostic query constructs. As an order agnostic query construct has unordered type semantics, the compatibility of an unordered type under ordered and typed data has to be checked. Ordered regular expressions give rise to modelling arity dependencies of some symbols, e.g. the regular expression "`(aab)+`" states, that there are twice as many `a` symbols than `b` symbols in valid data. Using Schwentwick formulae, it is not possible to express such dependant occurrence constraints on data, as the multiplicities are always formulated absolutely. Using linear equation systems restricted to natural number solutions gives rise to expression such constraints. Interestingly, it is *almost* enough to use Presburger arithmetic [42], of which formulae are much easier to solve under the given natural number solution constraint. This approach is used as starting point of an unordered type specification language later on.

Presburger Arithmetics Expressions A formalism not only conceived for unordered content models, yet formally elegant, are Presburger arithmetic formulae [42]. The approach has been applied to unordered content models in XML by Denis Lugiez and Silvano Dal Zilio to formalize XML Schema [67] and by Seidl, Schwentick, Muscholl, and Habermehl [45] as formal model for unordered content models. Presburger arithmetic is the first order theory of natural numbers with addition but without multiplication. For any given statement in Presburger arithmetic it is decidable, if this statement is true or not and, if partially variable, with which variable bindings the expression is true.

The relationship between languages of multisets and Presburger arithmetic is the commutativity and associativity of the addition (in Presburger arithmetic) and of the sequence operator (in the serialisation of multisets). The commutativity and associativity of sequence operators in serialisations of bags, multisets or sets has been presented e.g. in [19]. A multiset of symbols can fully be characterized by the multiplicity of the occurring symbols. A Presburger arithmetic expressions with variables could have many solutions, hence many possible natural number assignments for the variables. Each assignment can be interpreted as the multiplicity of a given symbol in a multiset which is valid with respect to the given Presburger arithmetic expression.

3.2.0.2 Applications, Shortcomings and Extension of current Approaches

Yet multiplicity lists are very restricted and not closed under intersection, arguably they are an elegant way to model multisets in the context of mixed ordered and unordered contents, as multiplicity lists are restricted regular expressions wrapped with a clear and simple interpretation of *unorder*. While Presburger Arithmetic are more expressive and offer the decidability and closedness properties for type checking, end users without scientific background arguably can be confused by the very mathematical formalism and the need to learn two content model specification mechanisms—regular expressions for ordered content models and Presburger arithmetic formulae for unordered content models. For modelling of unordered content with the grammar formalism proposed later, an extension of multiplicity lists will be introduced—unrestricted regular expressions with unordered interpretation.

As a not formally introduced consider the regular expression for a data object representing a course in a dancing school as seen in code example 24:

```
DancingMaster, ((Boy, Girl)+ | BalletDancer+ )
```

Code Example 24 A regular content model representing well balanced dancing classes for either ballet dancers or standard couple dancers. While the regular content model implies an order, the order of the objects are arguably not of importance for the application.

For every course there must be a dancing master. Courses may be standard dance courses, in which the same amount of boys and girls have to be registered so everyone always can have a partner. Courses may be ballet dancing courses, in which case at least one ballet dancer has to be registered. The order of the objects in a content set corresponding to this regular expression is irrelevant, e.g. the dancing master may be assigned at the end when the kind of course is clear. Therefore all words with a permutation of symbols that is matched by the regular expression is member of the unordered language represented by the regular expression.

An example of a Presburger arithmetic expression representing the same constraints on a valid dancing course is arguably more difficult to understand and to write for authors:

Let d be the multiplicity of `DancingMaster` objects, m the number of `Boys` (m for male), f of `Girls` (f for female) and b for `BalletDancers`. A multiset of objects is in the given language, if the multiplicities of the symbols fulfills the following formula:

$$d = 1 \wedge ((m = f \wedge m \geq 1 \wedge f \geq 1 \wedge b = 0) \vee (m = 0 \wedge f = 0 \wedge b \geq 1))$$

As shown in Chapter 7 the unordered interpretation of regular expressions can always be mapped to Presburger arithmetic constraints and is therefore decidable and closed under the desired properties, yet no system for translation of Presburger formulas and constraints back to regular expressions will be presented (as not strictly needed for type checking). An advantage of this approach for type checking is, that it can also be used for type checking of unordered queries on ordered content models (see e.g. section 2.5.4), which is valid and useful in Xcerpt as presented in [43]

3.3 Regular Rooted Graph Grammars

In the context of the Web and the Semantic Web often graph shaped data is considered in addition to tree shaped data. It is usually the case, that a dominant spanning tree in the graph is given and the 'missing' links are added by means of some reference mechanism. XML provides an integral reference mechanism known as ID-/ID-Ref [61] and various linking and reference based standards have been built

around XML, like *XML Fragment Interchange* [52] , *XML Linking Language (XLink)* [53] and many others. A practical example of such graph structured data is widespread in HTML documents—the use of internal links. Consider the HTML document in code example 25 with two references modelling a kind of circular link structure between two paragraphs:

```
<html>
 <body>
  <p id="p1">
   This paragraphs refers to the <a href="#id">next paragraph</a>
   (and is referred by it).
  </p>
  <p id="p2">
   This paragraphs refers to the <a href="#id">previous paragraph</a>
   (and is referred by it).
  </p>
 </body>
</html>
```

Code Example 25 An HTML document with two references modelling a kind of circular link structure between two paragraphs.

Another example of typically graph shaped structures modelled in XML are RDF documents.

3.3.1 Reference Types and Typed References

Modelling of graph shaped structures is supported in current XML schema formalisms like DTD, Relax-NG and XML Schema by modelling elements containing ID and ID-Ref attributes. The ID/ID-Ref mechanism is global throughout the whole document, meaning that any reference may refer to any identifier. Unfortunately this does not permit to model typed references, e.g. references to elements of certain type. Consider the following example for illustration of the ID/ID-Ref mechanism:

Example 3.3
Consider a grammar using similar syntax as in example 3.2 that models books and authors in a kind of bibliographical database. To prevent redundancy and misspelling of the author names, the authors are kept in an index of authors and are referred to from the definition of the books in another section of the document. The grammar uses the special type name " $\frown$ " for references and the type name " @ " is used to denote IDs. An element should only get one identifier, which arguably simplifies understanding document instances. To emphasise the special role in multiplicity of an identifier (each element may not contain more than one identifier), it is prefixed to the element name in the type rules. An author element contains an " @ ", denoting that instances of that type have an identifier and the authors list of a book has an arbitrary amount of references to author elements. Note, that for the sake of conciseness with the syntax of R_2G_2 presented later on, the type names " $\frown$ " and " @ " have been chosen so that they harmonize with R_2G_2. In turn, the syntax of R_2G_2 has been streamlined with the syntax of Xcerpt. The mapping to XML attributes named `id` and `ref` is given in an opaque way in this example, yet a concrete schema or type formalism would need a way to specify such mappings. Further on the type names *Name* and *Title* are synonyms for plain text, or CDATA in the sense of XML—again, a concrete schema or type formalism needs support for atomic data types to be practically useful.

$$G = (Bibliography,\\ \{\frown, @, Bibliography, AuthorIndex, Author, BookIndex, Book, Authors\},\\ \{bib, authors, author, books, book\},\\ R)$$

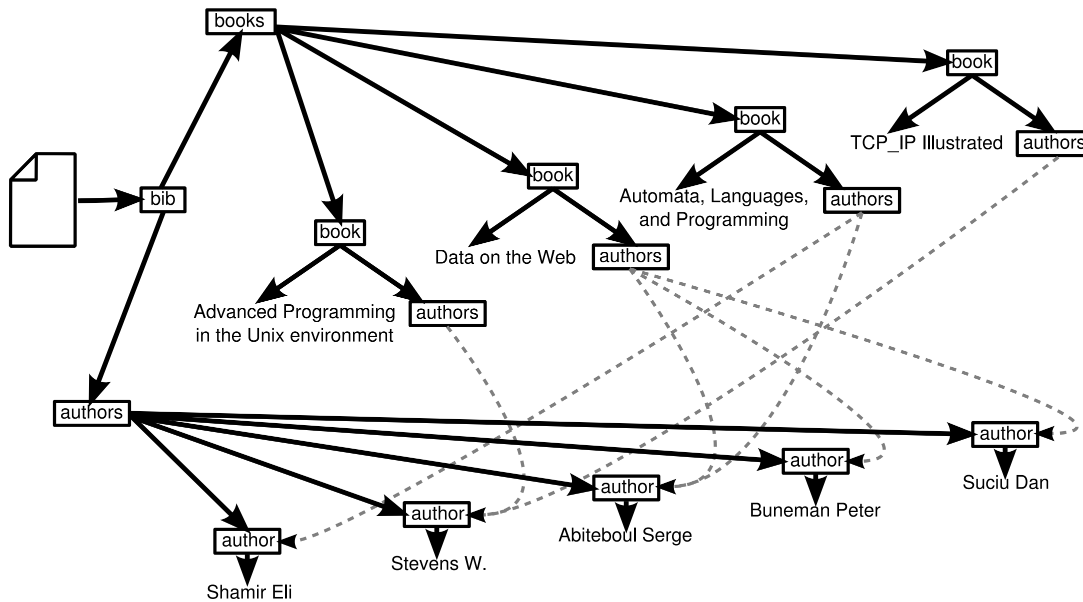
where $R =$
$\{\ Bibliography \rightarrow bib[AuthorIndex, BookIndex]$
$,\ AuthorIndex \rightarrow authors[Author+]$
$,\ Author \rightarrow @author[Name]$
$,\ BookIndex \rightarrow books[Book+]$
$,\ Book \rightarrow book[Title, Authors]$
$,\ Authors \rightarrow authors[\ \hat{}+]\ \}$

See code example 26 for an XML instance valid to the schema shown above.

```
<bib>
  <authors>
   <author id="se">Shamir Eli</author>
   <author id="sw">Stevens W.</author>
   <author id="as">Abiteboul Serge</author>
   <author id="bp">Buneman Peter</author>
   <author id="sd">Suciu Dan</author>
  </authors>
  <books>
   <book>
     Automata, Languages, and Programming
     <authors refs="as se"/>
   </book>
   <book>
     Data on the Web
     <authors refs="as bp sd"/>
   </book>
   <book>
     Advanced Programming in the Unix environment
     <authors refs="sw"/>
   </book>
   <book>
     TCP_IP Illustrated
     <authors refs="sw"/>
   </book>
  </books>
</bib>
```

Code Example 26 An XML document instance valid with respect to the schema shown above.

The former document could be graphically interpreted as follows:



Now, consider the following example for illustration of problems with untyped references:

Example 3.4
In difference to the former example, books are also referable elements here, and author elements contain references to their books. This is modelled by providing the reference type name "ˆ" and the identifier type name " @ " both in the author and book elements.

$$G = (Bibliography,$$
$$\{\hat{}, @, Bibliography, AuthorIndex, Author, BookIndex, Book, Authors\},$$
$$\{bib, authors, author, books, book\}, R)$$

where $R =$

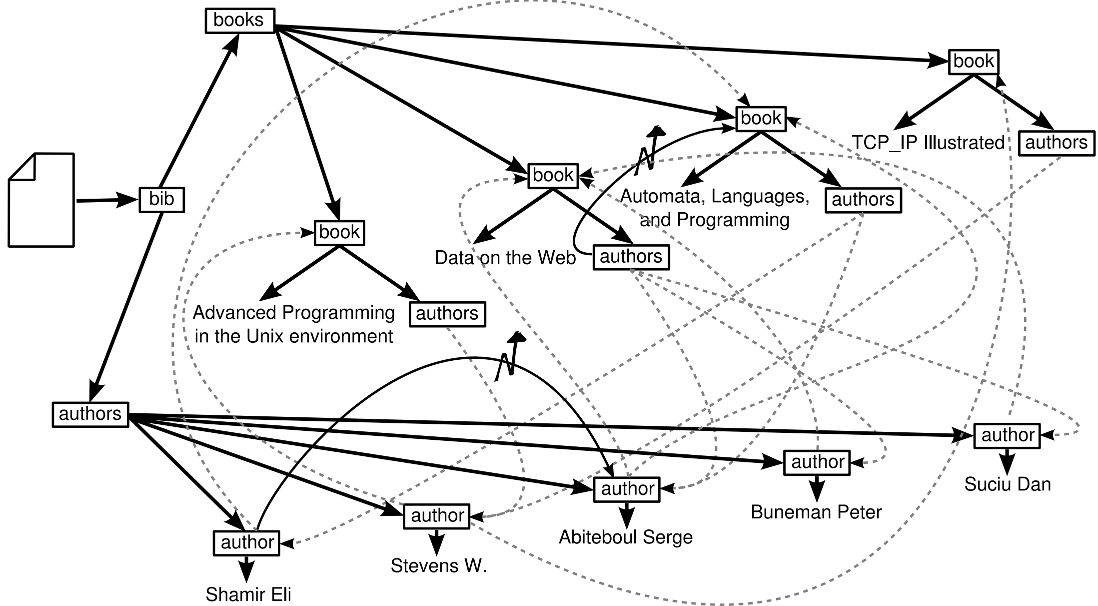
$$\{\ Bibliography \rightarrow bib[AuthorIndex, BookIndex]$$
$$,\ AuthorIndex \rightarrow authors[Author+]$$
$$,\ Author \rightarrow @author[Name,\ \hat{}+]$$
$$,\ BookIndex \rightarrow books[Book+]$$
$$,\ Book \rightarrow @book[Title, Authors]$$
$$,\ Authors \rightarrow authors[\ \hat{}+]\ \}$$

The following example document illustrates a valid document with respect to the grammar. Unfortunately, it is not possible to distinguish references to authors from references to books, resulting in a *conceptually* invalid document, where a book is referred to as an author of another book and an author contains another author in his list of published books.

```
<bib>
  <authors>
   <author id="se" ref="alap as"> <!--CONCEPTUALLY WRONG-->
     Shamir Eli
   </author>
   <author id="sw" ref="apitue ti">Stevens W.</author>
   <author id="as" ref="alap dotw">Abiteboul Serge</author>
   <author id="bp" ref="dotw">Buneman Peter</author>
   <author id="sd" ref="dotw">Suciu Dan</author>
  </authors>
  <books>
   <book id="alap">
     Automata, Languages, and Programming
     <authors refs="as se"/>
   </book>
   <book id="dotw">
     Data on the Web
     <authors refs="as pb sd alap"/> <!-- CONCEPTUALLY WRONG -->
   </book>
   <book id="apitue">
     Advanced Programming in the Unix environment
     <authors refs="sw"/>
   </book>
   <book id="ti">
     TCP_IP Illustrated
     <authors refs="sw"/>
   </book>
  </books>
</bib>
```

Code Example 27 This document is *conceptually wrong* with respect to the schema 3.4.

The document in code example 27 could be graphically interpreted as follows (the conceptually wrong references are denoted by flashes crossing the edges):



The proposed schema language R_2G_2 will introduce typed references to regular tree grammars to model graph shaped data in a more precise way. Syntactically, typed names are type name extensions of references—a type name intended to be referred to is appended.

Example 3.5
This grammar is an extension of the grammar in example 3.4, such that the references to authors and books are clearly separated. The conceptually erroneous example document of example 3.4 is invalid under this grammar:

$G = (Bibliography,$
$\{\hat{}, @, Bibliography, AuthorIndex, Author, BookIndex, Book, Authors\},$
$\{bib, authors, author, books, book\}, R)$
where $R =$
$\{\ Bibliography \rightarrow bib[AuthorIndex, BookIndex]$
$,\ AuthorIndex \rightarrow authors[Author+]$
$,\ Author \rightarrow @author[Name,\ \hat{}Book+]$
$,\ BookIndex \rightarrow books[Book+]$
$,\ Book \rightarrow @book[Title, Authors]$
$,\ Authors \rightarrow authors[\ \hat{}Author+]\ \ \}$

3.3.2 About (Non Tree Structured) Graphs and Tree Grammars

On some graph serialisation formalisms like RDF, the structure of the serialisation is semantically irrelevant in the sense, that the underlying graph semantics of different serialisations is considered to be isomorphic. The underlying graph structure may nevertheless need some sort of schematizing. As long as the graphs have a special node, called the root node from now on, which is chosen as the starting point for graph traversals, a tree grammar can also be used to model some structural properties of the graph. The root is used as starting point for graph traversal. For a given root, the set of all possible graph traversals is unambiguously determined. A tractable way to realize rooted graph modelling using tree grammars is based on simulation preorder: a tree grammar is a generator for the language of all trees that can be obtained by means of rule application and also of all rooted graphs that can be obtained by sharing of nodes that result from the same (possibly infinite) chain of rule applications. An implication is that, concerning schema validity, there is no distinction of two graphs where one of them shares one instance of a node in many positions, and the other one has multiple instances with identical shape or value used instead of sharing. One graph is then indistinguishable from another one, if it is *simulated* by the other graph. Graph isomorphism is arguably the most precise notion of *equality of indistinguishable* for graphs, simulation is weaker in the sense, that graphs that are not isomorphic may simulate or even bi-simulate. Consider example 3.1 for two rooted directed graphs that bi-simulate, but that are not isomorphic—the central difference between (bi)simulation and isomorphism is, that there is no bijection between the nodes of two bi-simulating graphs such that the related nodes have similar in and outbound *behaviour*. In contrast, such a bijection is needed for isomorphic graphs. The disadvantage of identification, distinction or recognition of objects using graph isomorphism is, that decidability comes with exponential cost, while the simulation preorder of two graphs can be checked in polynomial time. Arguably identification, distinction or recognition of objects based on simulation is useful on many practical contexts on the web, as many practical use cases with Xcerpt prove [34]. For a brief introduction of simulation and simulation unification along with some examples, see section 2.5.4.

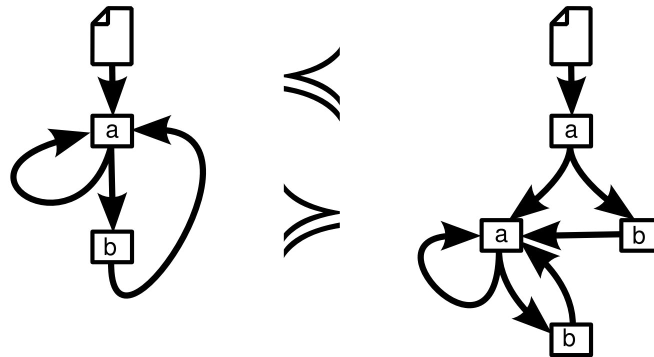


Figure 3.1: Two graphs that bi-simulate but that are not isomorphic.

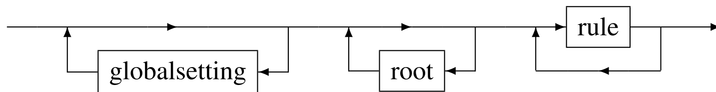
3.4 The Syntax of R_2G_2

A grammar for R_2G_2 is given as a set of syntax graphs. The graphs are interleaved with corresponding explanations.

3.4.1 Core R_2G_2 Syntax

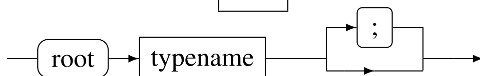
The core R_2G_2 syntax consists of all constructs necessary to model rooted graph grammars. Features as namespace handling, modules, predefined atomic data types and the various global settings of a grammar are not core syntax features. The separation of core and non core features is roughly motivated by application of R_2G_2 definitions for type checking or validation—non core R_2G_2 definitions can mostly be translated into core definitions and operations like automata construction are done on the core level. To get a concise language definition, non terminals for non core features are in the language definition rules of the core syntax, yet their definitions follow in a later sub section.

Definition 3.4 (grammar)



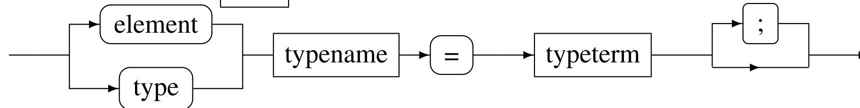
A *grammar* contains one to many rules and relates them to root declarations. In the context of Xcerpt a grammar as type definition will be located either in a separate file (being included using the modularization mechanism presented later) or it may be in a special context inside the Xcerpt program. A grammar may also contain global switches that affect the behaviour or semantics of the whole grammar.

Definition 3.5 (root)



A root declaration declares a type name as a possible type for a root element, therefore as a possible type for a data tree or graph. The type name used has to be declared later on, otherwise the grammar is invalid. Note that ␣ denotes a white space, `\n`, `\r`, and `\t` new line, carriage return and the tabulator character.

Definition 3.6 (**rule**)



A rule is either introduce by the keyword `element` or `type`. The keyword `type` is optional and especially useful for compatibility with type definitions as found in Xduce [29], as many Xduce type definitions are also R_2G_2 definitions with similar syntax and semantics. A rule relates a type name (also called non terminal sometimes) and a type term. A type definition rule may or may not be terminated using a semicolon and arbitrary spaces and new lines. The same type name may be used in many rules involving different type terms.

Definition 3.7 (**typename**)

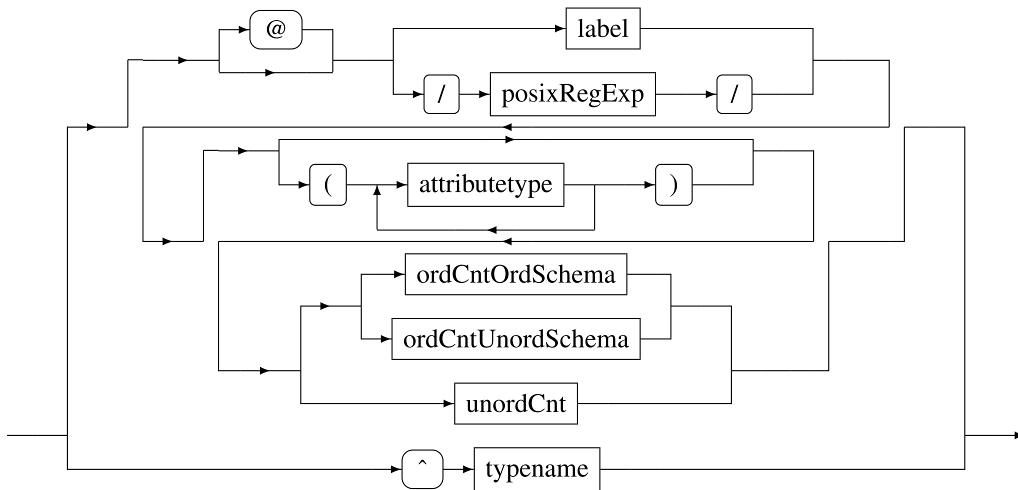
Definition 3.8 (**label**)

Type names and labels are alphanumerical strings with letters as first character.

For the purpose of compatibility with XML, this can be extended to NMTOKEN as defined in the XML recommendation [61]. The label of a type term corresponds to element labels of data trees matching a type term. Note, that NMTOKEN allows the use of the colon sign ":", which allows namespace aware schema modelling. Namespaces and namespace prefixes can be declared, as shown in section 3.4.5.

Type names and type term labels may compete, as the context determines the meaning of a corresponding token.

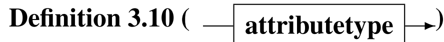
Definition 3.9 (**typeterm**)



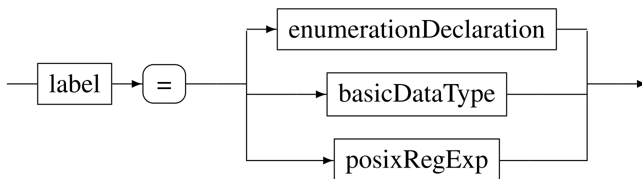
A type term may be a type declaration term, optionally prefixed with an "@"—it is a type declaration term for referable instances, or it may be a reference type term—a type name with a hat as prefix. A type declaration term always has a label declaration and a content model, this content model may either be an unordered content model or an ordered content model. Label declarations may be specified as regular (string) expression at the position of the label, regular (string) expressions at label position are enclosed by slashes. Ordered content models can be specified in an unordered way or in an ordered way. Optionally, a type declaration term may have attribute type declarations in the sense of XML attributes. They are declared in parenthesis between the label and the content specification.

While usually any data instance may be a reference to some defining instance, even if the corresponding type is not declared as reference type, the global grammar setting `strictreferences` can be used to restrict valid data instances such, that references may only be instances of reference types.

A non reference type term can be prefixed by an "@", denoting that instances of that type must be referable, e.g. in a XML ID/ID-Ref setting they must have an identifier. While usually any data instance may be defined using an identifier, even if te corresponding type is not declared as referable, the global grammar setting `strictreferences` can be used to restrict valid data instances such, that only referable types can and must be indexed.

Definition 3.10 (

)



Attribute types are tuples of a label and a base type (base types are explained later at 3.4.2) that may be braced and question mark annotated for optionality in the spirit of regular expression optionality.

Definition 3.11 (ordCnt)

The ordered content of a type term is a regular expression (of type terms and type names) surrounded by square braces.

An ordered type term specifies a type such that all data tree instances of that type have ordered content and the child trees types match the regular expression of the content model of the type term. This corresponds roughly to the content definitions expressible using XML Schema, DTD and Relax NG. To specify types with empty content model, the brackets can be left empty.

Definition 3.12 (ordCntUnordSchema)

Some ordered content specification may be specified more comfortably in an unordered way. As an example consider an element representing a member list of a well balanced dancing class with the same amount of boys and girls:

```
<dancing-class>
  <girl>Anna</girl>
  <boy>Fitzgerald</boy>
  <boy>Guillermo</boy>
  <girl>Quibee</girl>
</dancing-class>
```

Code Example 28 An example document representing a well balanced dancing class—there are as many boys as girls in the class. The order of the members is conceptually irrelevant.

The given constraint on the language is not regular—this can easily be seen, as the language contains the sub language *"n times* `<boy>` *followed by n times* `<girl>`*"* which is well known to be non regular. However, the language *"one* `<boy>` *followed by one* `<girl>` *and this repeated n times"* is regular (e.g. $(boy, girl)^*$). Ordered content with unordered specification uses a regular expression to model a language, but also the permutation of all words contained in the language of that regular expression are member of the unordered specified ordered language. The order of the data instances is considered to be relevant—e.g. in the former example we have an alphabetically sorted lists of dancing class members—but the specification does not impose an order.

Definition 3.13 (unordCnt)

A type term may have unordered content specification, expressing that conforming data trees have an (unordered) multiset of sub trees instead of an (ordered) sequence of sub trees. By modelling data with unordered content, a storage or indexing system is given the freedom to ignore the order given while data instantiation and reorder it e.g. for more efficient querying.

Definition 3.14 (regexp)

typeterm
typename
contentMacroName
(regexp)
+
*
?
(number - number)
number -
- number
regexp
| regexp

Regular expressions as defined here have sequences, disjunctions, optionality and various forms of repetitions. The symbols or atoms of the regular expressions may be type names or type terms. A type name specifies a sub tree of the tree containing the current content model of that type. A type term as atom specifies a tree of anonymous type name. Obviously type names may only be used, if declared elsewhere, either in the grammar or imported by some matter.

In XML Schema and DTD a restriction called *deterministic content model* is required for the regular expression content models. In the W3C recommendation of XML it is said:

> [...]
> More formally: a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman]. In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error.
> [...]

As there are algorithms constructing non deterministic finite state automata out of arbitrary regular expression, the mentioned restriction arguably is not necessary. The language class of regular expressions with this restriction is not closed under union, which is not essential for type checking of Xcerpt, but arguably a nuisance, hence the type of a query disjunction is inconvenient—if not impossible—to model.

3.4.2 Base Data Types

An important part of an XML document are textual leaf nodes, that a structuring element may contain. Often this data is **the** relevant information of a document, in this case the element structure wrapping it is considered to be mere *markup* or auxiliary information. Textual leaf data may occur wherever any structured element is allowed (except the document root—it must be an element) but two textual leaf nodes may not occur as direct neighbours in a sequence of nodes. The only valid information, that an attribute may contain is also a textual node.

In XML the textual information is often called CDATA for *character data*. As for data oriented applications also numerical or other data is used, such nodes will be referred to as *data* or *data of base type*. Arguably, data can nevertheless be called character data, as in the XML world they have a textual serialization in the corresponding XML serialisation—the proposition here is, that the schema formalism models the data on a conceptual level, not on a syntactical level.

The versatile nature of XML and it's query languages makes the choice of an arbitrary set of fixed base data types questionable. For R_2G_2 hence a generic base data type system is proposed, that can be instantiated with various concrete base data types. A small set of base data types is then given as a pragmatic yet simple to use and to implement set of basic data types.

The generic base type system has been conceived with the needs of the Web and Semantic Web query language Xcerpt in mind and hence has features like functions, which are not relevant at the given point for R_2G_2 as a schema language. Functions are an important concept for base data types in Web query languages and are of relevance for the application of R_2G_2 as a type language for Xcerpt, hence for Web query languages in general, in the course of this deliverable for Xcerpt and Xcerpt type checking.

3.4.2.1 Definition of the Generic (abstract) Base Data Type System

By T the set of all base types is denoted.

The set $F = F_* \cup \bigcup_i F_i$ is the set of all functions, where $F_i \subseteq T \times T^i \times \Sigma \times \mathtt{F}_i$ is the set of all functions of arity i with result type out of T and argument types out of T^i. By Σ the set of all function names is denoted. By $\mathtt{F}_i$ the set of functions with arity i at implementation level is denoted. The functions at implementation level are untyped in the sense of the discussed type system—they may be typed at the level of the implementation language, if typed dynamic function application is available in the hosting programming language. By $F_* \subseteq T \times T \times \mathtt{F}_*$ aggregation functions are denoted. They operate on a sequence or set of values of an homogeneous type, returning a value of a (maybe) different type.

A value in an instance document is always a tuple (v,t) where $t \in T$ and v is an opaque object at the level of the type system—an implementation level representation of a value of type t.

The directed relation $S \subseteq T \times T$ between two types t_1 and t_2 contains the information, that the given transformation function $(t_1, t_2, c_{t_2}) \in F$ between $t_1 \in T$ and $t_2 \in T$ never fails. The reflexive transitive closure of S over all types of T is $\vec{S}$. The relation S is defined by the user of the general purpose type system, which is usually the creator of a new base data type. When defining a new base data type it is hence possible to define conversion functions and to inform the type system about convertibility, but it would be inconvenient to define convertibility for any base data type on the web—especially for upcoming base types it is not reasonable. The subtype concept is related to $\vec{S}$, yet different—generally, the subtype concept implies, that instances of a sub type have all the properties of instances of their super types and possibly further. For the proposed base type system for Web query and schema languages, the relation $\vec{S}$ implies only one property—instances of a type are (for sure) convertible to another type. The rational behind this is, that data objects are not representations of structures and functions as e.g. in programming languages, but serialisations of XML documents. As an example for the difference of $\vec{S}$ and the subtype relation as commonly known, consider the type of the natural numbers and of strings—each natural number can be transformed into a String, but natural numbers are not a subtype of strings in many programming languages, as they do not share common semantics—strings have string operations like concatenation and tokenization and natural numbers have arithmetic operations.

As counter pole to the $\vec{S}$ relation, the undirected relation $\vec{D} \subseteq T \times T$ contains the information, which type is for sure not convertible to which other type. D is the relation of types marked by the user of the generic base type system. The relation $\vec{D}$ is the reflexive transitive closure over D. Note, that the set $T \times T \neq D \cup S$, as some types have instances, that can be converted to another type while not all instances are convertible to that type. These tuples are neither in S nor in D.

3.4.2.2 A pragmatic set of base types and functions

To show that the type and function framework can actually be used to extend Xcerpt reasonably, a pragmatic set of base types and functions is presented.

Figure 3.2 shows the base type hierarchy. The types are a direct copy of a branch of the hierarchy presented in the XML Schema data types recommendation [59]. Five of the arguably most important types of the XML Schema recommendation are rebuilt using the generic type system. Others are possible, but for pragmatic reasons not treated in this context.

3.4.3 Conversion functions

The following set of conversion functions between instances of the base types are proposed. They cover safe and unsafe functions. Note, that all functions convert retaining syntactical equivalence. It is i.e. possible to convert the integer 12 to the rational number 12, but there is no Boolean interpretation of e.g. the integer 12. Type conversion of syntactical equivalence is important for Web query languages, as data of various types always gets serialized to XML character data in the end.

string	:	*rational*	$\rightarrow$ *string*	safe conversion from *rational* to *string*
string	:	*boolean*	$\rightarrow$ *string*	safe conversion from *boolean* to *string*
string	:	*IRI*	$\rightarrow$ *string*	safe conversion from *IRI* to *string*
rational	:	*string*	$\rightarrow$ *rational*	conversion from *string* to *rational*
rational	:	*integer*	$\rightarrow$ *rational*	safe conversion from *integer* to *rational*
integer	:	*rational*	$\rightarrow$ *integer*	conversion from *rational* to *integer*
boolean	:	*string*	$\rightarrow$ *boolean*	conversion from *string* to *boolean*
IRI	:	*string*	$\rightarrow$ *IRI*	conversion from *string* to *IRI*

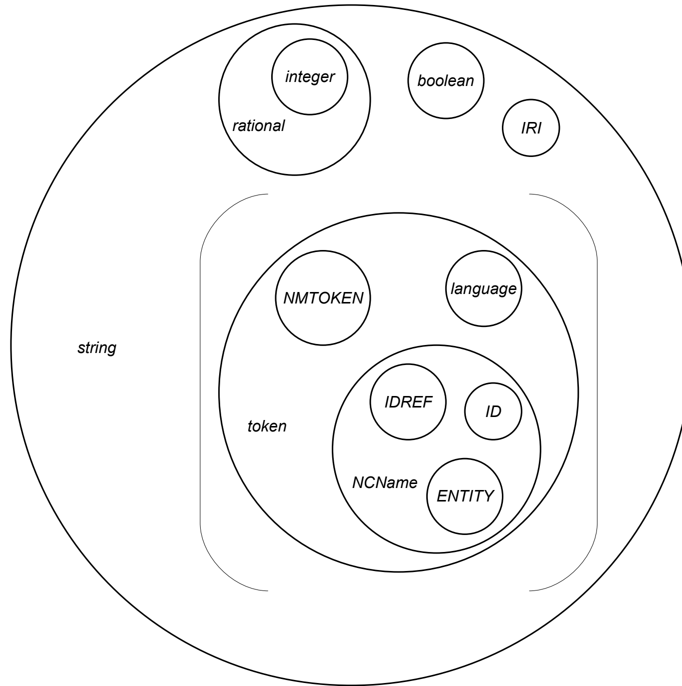


Figure 3.2: Base type system hierarchy

The following set of base functions by no means claims to be exhaustive of what one may want to do using the proposed base data types. It is however a small collection of arguably most useful functions and operations for the given types—nothing more than a pragmatic set of functions.

3.4.4 Base functions

3.4.4.1 *string* functions

concat	:	$string \times string$	$\to string$	concatenation of two strings
substr	:	$string \times integer$	$\to string$	extracts sub string from string
substr	:	$string \times integer \times integer$	$\to string$	...with given length
lowerCase	:	$string$	$\to string$	converts string to lower case
upperCase	:	$string$	$\to string$	converts string to upper case
invCase	:	$string$	$\to string$	inverts case of a string
trim	:	$string$	$\to string$	trims string
find	:	$string \times string$	$\to integer$	finds search string inside of a string
find	:	$string \times string \times integer$	$\to integer$	...with given start position
replace	:	$string \times string \times string$	$\to string$	replaces search string inside of a string
replace	:	$string \times string \times string \times integer$	$\to string$	...with given start position for search
replaceAll	:	$string \times string \times string$	$\to string$	...with all occurrences being replaced
tokenize	:	$string \times string \times integer$	$\to string$	string tokenization

3.4.4.2 *rational* functions

abs	:	*rational*	$\rightarrow$ *rational*	calculates the absolute value of a given number
sign	:	*rational*	$\rightarrow$ *rational*	calculates the sign of a given number
inv	:	*rational*	$\rightarrow$ *rational*	calculates the (additive) inverse of a given number
add	:	*rational* $\times$ *rational*	$\rightarrow$ *rational*	addition of two numbers
sub	:	*rational* $\times$ *rational*	$\rightarrow$ *rational*	subtraction of two numbers
mul	:	*rational* $\times$ *rational*	$\rightarrow$ *rational*	multiplication of two numbers
div	:	*rational* $\times$ *rational*	$\rightarrow$ *rational*	(rational) division of two numbers
pow	:	*rational* $\times$ *rational*	$\rightarrow$ *rational*	exponentiation of number
sqrt	:	*rational*	$\rightarrow$ *rational*	calculates the square root of a number
round	:	*rational*	$\rightarrow$ *integer*	rounds a number (explicit, lossy conversion to *integer*)

3.4.4.3 *integer* functions

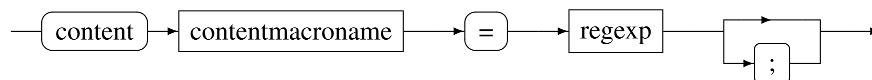
abs	:	*integer*	$\rightarrow$ *integer*	calculates the absolute value of a given number
sign	:	*integer*	$\rightarrow$ *integer*	calculates the sign of a given number
inv	:	*integer*	$\rightarrow$ *integer*	calculates the (additive) inverse of a given number
add	:	*integer* $\times$ *integer*	$\rightarrow$ *integer*	addition of two numbers
sub	:	*integer* $\times$ *integer*	$\rightarrow$ *integer*	subtraction of two numbers
mul	:	*integer* $\times$ *integer*	$\rightarrow$ *integer*	multiplication of two numbers
intDiv	:	*integer* $\times$ *integer*	$\rightarrow$ *integer*	integer division of two numbers
mod	:	*integer* $\times$ *integer*	$\rightarrow$ *integer*	calculates the remainder of an integer division
pow	:	*integer* $\times$ *integer*	$\rightarrow$ *integer*	exponentiation of number
intSqrt	:	*integer*	$\rightarrow$ *integer*	calculates the square root of a number

3.4.4.4 *boolean* functions

and	:	*boolean* $\times$ *boolean*	$\rightarrow$ *boolean*	logical conjunction of two Boolean values
or	:	*boolean* $\times$ *boolean*	$\rightarrow$ *boolean*	logical disjunction of two Boolean values
not	:	*boolean*	$\rightarrow$ *boolean*	logical negation of a Boolean value

3.4.5 The Non-Core R_2G_2 Constructs

Definition 3.15 (contentMacro)



Content macros are substitutions for regular expressions as used in content model specifications. A content macro is a regular expression of type names, content macro names or type terms that is associated to the content macro name on the left hand side of the macro declaration rule. Content macro names, data type names and element type names all share the same set of names, e.g. they may not conflict. As content macros are expanded statically, content macro declarations may not be circular. Note, that the non circularity is a sufficient criterion for retaining the regularity property of content models, as it prevents intuitively the construction of context free like grammar rules with a pumping circle surrounded of pumped terminal symbols.

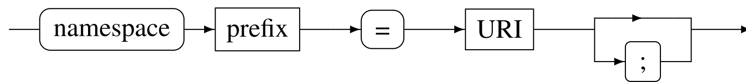
3.4.5.1 Namespaces

From a practical point of view a very important property of XML is namespace handling. The purpose of name space handling is to distinguish equal names by partitioning the set of labels in sets that are likely to be named in a globally unique way. Name space handling is realized as an extension of the naming scheme—label and attribute names are tuples of a so called local name part and a namespace part. URIs are used as name spaces, as they arguably fulfill the necessity of uniqueness by definition. As URIs tend to be long strings, cumbersome to repeatedly type, a substitution mechanism called *namespace prefixing* is used. The user hence defines a short, yet unique (in it's scope, e.g. the grammar) namespace prefix for a namespace and uses this whenever he wants to reference the namespace, the system resolves the prefix to the expected namespace URI.

Name spaces have to be reflected when modelling schemata—labels of elements or attributes have to be composed of a local and a namespace part, as instance documents may need name spaces.

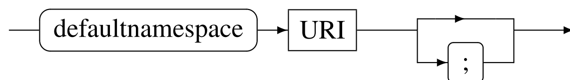
Additionally to label and attribute names, R_2G_2 contains type names, that may also need partitioning in different "name spaces"[3], as name conflicts can occur between type names of different grammars that may interact when modelling modular grammars or grammar fragments. Name conflicts for type names are not handled using the XML name space mechanism, but a module system specialized for Web rule languages and presented later (see chapter 4).

Definition 3.16 (**nsPrefixDeclaration**)



Definition 3.17 (**prefix**)

Definition 3.18 (**defaultNs**)



A namespace prefix declaration is done by associating an URI to a prefix. Prefixes are (like type names) tokens similar to labels. The set of prefix definitions and the set of type definitions are not shared,

[3] "Name space" is quoted here, as it refers to the concept of name spaces, not to XML name spaces specially.

e.g. prefixes and type names may be identical.[4] A default namespace can be declared, concerning all definitions—element label or type names—without namespace prefix. The *default* default namespace is empty.

3.5 XML Serialisation of R_2G_2 Valid Data Terms

When modelling graph structured data using R_2G_2 the typed reference mechanism is able to conceptually reflect a graph structure, yet the XML serialisation is not able to directly reflect this conceptual graph without further help. Modelling of graph references and identifiers leaves unspecified how to reflect the information e.g. in ID and ID-Ref based node identities in data instances, e.g. how ID or ID-Ref attributes have to be named and especially how references have to be de-referenced—as ID-Ref instances are attribute values, they cannot be replaced by the referred elements, as attributes may not contain structured information. The approach chosen to solve this problem is to specify the structural relationship between the references, respectively identifiers, and their de-referencing, respectively the identified constructs. R_2G_2 provides some special built in constructs that represent serialisations of identifiers or references of a so called context.

The context of an identifier construct is either the document order or the nesting relation of an identifiable element type or a reference type, more concrete:

- An identifier serialisation type can occur as descendant of an identifiable element type. No other identifiable element type may occur on the path in between identifier serialisation location and identifiable element.
- An identifier serialisation type can occur in document order before or after an identifiable element type. No other identifiable element type may occur in between identifier serialisation location and identifiable element in document order. Solely for easier comprehension, the identifier serialisation location and the identifiable element must occur in the same content model definition, e.g. they may not be spread through different element definition rules[5].
- A reference serialisation type can occur before or after a typed reference location in document order of an instance of a type. Multiple instances of the same reference type can occur aggregated using the Kleene Star or Plus operator in a regular expression, if the corresponding reference type serialisation is also multiplied using the same regular expression operator. This makes it possible to e.g. model an ID-refs attribute with a list of references all reflecting objects to be linked in the content of the corresponding element.

The presented approach does not claim to completely solve the problem for all structural possibilities of a relationship between concept and serialisation of identifiers and references. It is a proposal of a pragmatic approach to the problem, as for practical usability of R_2G_2 it is needed, some useful cases can be modelled using this technique. The serialisation of the conceptual graph structure is considered to be of minor importance in the context of this deliverable, yet worth while more investigation.

3.5.1 Examples and Explanations of (De-)Reference Serialisation

Consider the following example of how to specify reference and de-reference serialisation types followed by explanations.

[4] ideally the sentence would be *"Prefix definitions and type names do not share the same namespace"*—the *namespace* meant here is not an XML namespace, but the definition space of names in R_2G_2.

[5] This should prevent the use of identifier serialisation types out of the scope of an identifiable element by mistake.

Serialisation of Referable Objects

```
namespace r2g2 = http://pms.ifi.lmu.de/ns/r2g2/v1.0/ ;
element Book = @book[ Authors , Title , key[ r2g2:ancestor-id ] ] ;
element Authors = .... ;
element Title = .... ;
```

Code Example 29 Using the `ancestor-id` type to locate an identifier in an XML serialisation of a referable book type declaration.

The example 29 models an object that represents a book as it may be found in a bibliographic database. The book element is intended to be referable, hence indexed. The index identifying the book is contained as unique identifier text node in an element called `key`, that is a direct child of the book element. This is done by using the special type name `ancestor-id` in the namespace `http://pms.ifi.lmu.de/ns/r2g2/v1.0/` that serves to generate a textual value that is assured to be a valid identifier to the nearest ancestor node that is tagged as referable (using the `@` prefix). Hence the `ancestor-id` has to be a descendant of the indexed book element type term.

The following instance document is valid with respect to the schema in example 29, it can hence be parsed and transformed with an appropriate R_2G_2 aware graph builder to a conforming conceptual graph, as well a conceptual graph conforming to former schema can be serialized to that instance document.

```
...                                ...
<book>                             id1@book[
 <authors>...</authors>              authors[...],
 <title>...</title>                  title[...],
 <key>id1</key>                      key["id1"],
</book>                            ]
...                                ...
```

Code Example 30 An XML instance, given on the left, that is a serialisation of a conceptual graph, given on the right in Xcerpt data term syntax, both valid with respect to the schema 29

The example 31 models a kind of technical report, where sections are referable. In the serialisations, the identifiers of a referable section are given as a preceding node, more precisely as named anchor in the spirit of HTML.

```
namespace r2g2 = http://pms.ifi.lmu.de/ns/r2g2/v1.0/ ;
element TR         = technical-report[ (Title, Section)* ];
element Title      = title[ A, r2g2:any-text ];
element Section    = @section[ (r2g2:any-text|(Title,SubSection))+ ];
element SubSection = @sub-section[ r2g2:any-text ];
element A = a(name=r2g2:previous-id)[ ];
```

Code Example 31 Using the `previous-id` type to locate an identifier in an XML serialisation of a referable section declaration.

The predefined type `previous-id` is an identifier serialisation that occurs previous of the referable object, in the examples case, the section or subsection. If the identifiers is to occur after the referable object, then the predefined type `following-id` can be used.

The following instance document, example 32, is valid with respect to the schema in example 31.

```
<technical-report>                      technical-report[
 <title>                                  title[
  <a name="s1"/>First Section               a(name="s1")[],
 </title>                                   "First Section" ],
 <section>                                s1@section[
  Some content ...                          "Some content ...",
  <title><a name="ss1"/>                    title[a(name="ss1")[],
    A Subsection                                  "A Subsection"
  </title>                                       ],
  <subsection>                              ss1@subsection[
  Some sub content...                         "Some sub content..."
  </subsection>                             ]
 </section>                               ],
 <title>                                  title[a(name="s2")[],
  <a name="s2"/>Second Section                  "Second Section"
 </title>                                 ],
 <section>                                s2@section[
  Content in 2nd section.                   "Content in 2nd section."
 </section>                               ]
</technical-report>                     ]
```

Code Example 32 An XML instance, given on the left, that is a serialisation of a conceptual graph, given on the right in Xcerpt data term syntax, both valid with respect to the schema 32

Serialisation of References The book example (example 29) is extended in example 33 such that the referable books are referenced in elements representing author information and vice versa.

```
namespace r2g2 = http://pms.ifi.lmu.de/ns/r2g2/v1.0/ ;
element Book = @book[ Authors , Title , key[ r2g2:ancestor-id ] ] ;
element Author = @author[ Name, Books ] ;
element Authors = authors[ (key[r2g2:following-ref], ^Author)* ];
element Books = books[ (^Book, key[r2g2:previous-ref])* ];
element Title = title[r2g2:any-text] ;
element Name = name[r2g2:any-text] ;
```

Code Example 33 A schema for books and authors in e.g. a bibliography database that may refer each other and with the reference serialisation types `r2g2:previous-ref` and `r2g2:following-ref`.

While identifier serialisations may be descendant nodes of the identified elements, references are atomic, i.e. have no structure, and hence may not contain content representing their serialisation. Therefore references have their serialisations always before or after their occurrence in document order. The corresponding example schema 33 is providing the reference serialisations of elements of type Author just before the reference in a `key`-element containing the identifier, the references of type Book are followed by similar `key` elements, describing the serialisation of those Book typed elements.

```
...
b1@book[
  authors[ key["a1"] , ^a1 , key["a2"] , ^a2 ],
  title["..."], key["b1"]
]
...
```

Code Example 34 An example instance of a bibliographic database conforming to 33.

Some regular expressions involving references and reference serialisations may however introduce problems: if between the reference and its serialisation a type with non empty intersection to the reference serialisation occurs, the interpretation of the serialisation may get ambiguous, especially if the intermediate type is optional or repeated. To prevent such problems, the sequence of types in document order between the reference and its serialisation must have empty intersection with the reference serialisation type.

In DTD's, XML Schema and Relax NG there exist kinds of serialisations that all correspond the DTD concept called IDREFS attribute value—an attribute with a sequence of references. As attribute values may not represent structured values, the need of anchoring a sequence of references in the context of a sequence of it's serialisations arises. This obviously breaks with the just stated condition for intermediate types with empty intersection between a reference and it's serialisation—a sequence of serialisations brings in ambiguity problems for any except the last element in the sequence of serialisations. An extra condition for this purpose exists: References may occur in uninterrupted sequence, i.e. in abstract syntax as child of a Kleene star or a plus construct, if reference serialisation types occur as repetition sequence in the scope of exactly the same regular repetition construct without ambiguity constraint violation in between. As an example, consider the following modification of a bibliography database similar to example 33, but with the authors of a book now located in a sequence of reference values in an appropriate `authorrefs`-attribute.

```
namespace r2g2 = http://pms.ifi.lmu.de/ns/r2g2/v1.0/ ;
element Book = @book(
                 authorrefs=(r2g2:following-ref)*
               )
               [
                 ^Author* , Title , key[ r2g2:ancestor-id ]
               ] ;
element Author = @author[ Name, Books ] ;
element Books = books[ (^Book, key[r2g2:previous-ref])* ];
element Title = title[r2g2:any-text] ;
element Name = name[r2g2:any-text] ;
```

Code Example 35 A schema for books and authors in e.g. a bibliography database that may refer each other and with the reference serialisation types, while references may occur in sequence as long as their serialisation also occur in sequence e.g. using the `r2g2:following-ref` type in a sequence.

```
...
b1@book(authorrefs="a1 a2")[
  ^a1, ^a2,
  title["..."], key["b1"]
]
...
```

Code Example 36 An example instance of a bibliographic database conforming to 35.

3.6 Semantics of R_2G_2

A declarative semantics for R_2G_2 is presented now mostly in the spirit of the acceptance relation of sheaves automata as presented in [36]—the acceptance of a data term under a schema is expressed. For implementation purpose and for later type checking, the declarative semantics is arguably too vague, so

an operational semantics is also given later on as the semantics on automata, that are considered to be the implementation base of R_2G_2. Complexity estimations are also more convenient to be done on the operational semantics level. The declarative semantics is however arguably easier to comprehend, as clearer related to data and schema.

The declarative semantics of R_2G_2 is given as a relation $\Gamma, R \vdash d : \tau$ meaning that an instance or a list of instances of data d satisfy a regular expression τ[6] under the schema declarations in Γ where Γ is the set of all `element` rules. The `content` macros are already expanded and not part of the schema anymore at that stage. The root declarations, more precisely the type names of the root declarations, form the set R. Γ will also be referred to as *environment* as common in literature about typing, e.g. in [41] and [36]. As the environment is not altered during the process of checking acceptance of a term, the usual extension operation on environments as in [36] is omitted here. The empty environment is denoted as $\emptyset$, the empty environment is well formed. The type name or variable is uniquely associated to one type declaration, i.e. $\texttt{element}X_1 = d_1$ and $\texttt{element}X_2 = d_2$ and if $X_1 = X_2$ then also $d_1 = d_2$ in a well formed environment or schema.

In constrast to [36] the environment is considered to be fixed, the schema is the environment. In [36] the name-schema associations are read out of the schema with an extra rule and introduced into the environment.

The document (root) must be accepted by a root type The possible types for a document, or better, for a root of a document are declared as root types in the R_2G_2 grammar and are therefore in the set R. By $\Box$, the document type, not really a type as modelled in R_2G_2 is denoted, like *document*(.) is not a construct of the data tree neither. Document and document type are external prerequisites to match a tree with a root type, to bootstrap the relationship. The notion of element types is recursively define, the root type gives an entry point for the recursion—this is in accordance with the recursive definition of XML data as defined by the W3C e.g. in the Infoset [56] or in the *Document Object Model* (short DOM) [60].

$$\frac{X \in R \qquad \Gamma, R \vdash t : X}{\Gamma, R \vdash \mathit{document}(\ t\ ) : \Box} \quad \text{(RootType)}$$

Types with ordered content model match nodes with ordered content The type of a node has the same name as its type declaration term, represented by the atomic regular expression consisting of only that type, and the content can be typed using the declaration terms content regular expression.

$$\frac{\Gamma, R \vdash t_1, \ldots, t_n : \tau}{\Gamma, R \vdash l[t_1, \ldots, t_n] : l[\tau]} \quad \text{(OrderedNodeTyping)}$$

$$\frac{\texttt{element}X = \tau \in \Gamma \qquad \Gamma, R \vdash t_1, \ldots, t_n : \tau}{\Gamma, R \vdash l[t_1, \ldots, t_n] : X} \quad \text{(NamedTypes)}$$

A data tree with named type has the type declared by the type term that is associated with the element declaration rule for the corresponding type name.

[6] τ is a regular expression of the domain of types

Optional Content An optional regular expression may be matched by no node or no node set, that means, if no node (or node set) matches that regular expression, it can be ignored. The rule is complementary to the second optional rule, that states that an optional regular expression may match.

$$\frac{\Gamma, R \vdash \diamond}{\Gamma, R \vdash \varepsilon : l[re]^?} \qquad \text{(OPTIONAL1)}$$

$$\frac{\Gamma, R \vdash t : l[re]}{\Gamma, R \vdash t : l[re]^?} \qquad \text{(OPTIONAL2)}$$

Sequence types and node sequences A sequence of nodes or sub sequences can be matched by a sequence of regular expressions, if each node or sub sequence is matched by a (positionally) corresponding regular expression in the regular expression type sequence.

$$\frac{\Gamma, R \vdash t_1, \ldots, t_{n-1} : \tau_1 \qquad \Gamma, R \vdash t_n, \ldots, t_{n+m} : \tau_2}{\Gamma, R \vdash t_1, \ldots, t_{n-1}, t_n, \ldots, t_{n+m} : \tau_1, \tau_2} \qquad \text{(SEQUENCE)}$$

Type disjunctions A node or node sequence can be matched by a disjunctive regular expression, if any of the regular expression matches the node or sequence.

$$\frac{\Gamma, R \vdash t : \tau_1}{\Gamma, R \vdash t : \tau_1 | \tau_2} \qquad \text{(DISJUNCTION1)}$$

$$\frac{\Gamma, R \vdash t : \tau_2}{\Gamma, R \vdash t : \tau_1 | \tau_2} \qquad \text{(DISJUNCTION2)}$$

Kleene star An arbitrary repeatable regular expression—an expression in the scope of a Kleene star—may match as many nodes or sub sequences in a sequence of nodes, as possible without interruption of the sequence by any other data.

$$\frac{\Gamma, R \vdash t_1, \ldots, t_n : \tau \qquad \Gamma, R \vdash t_{n+1}, \ldots, t_{n+n} : \tau \qquad \cdots \qquad \Gamma, R \vdash t_{n\times m+1}, \ldots, t_{n\times m+n} : \tau}{\Gamma, R \vdash t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+n}, \; \ldots \; , t_{n\times m+1}, \ldots, t_{n\times m+n} : \tau^*}$$

(KLEENESTAR)

Nodes with unordered content are matched by types with unordered content models Unordered content models are modelled as regular expressions that have to match any permutation of a given content sequence. Mainly this corresponds to an associative commutative interpretation of the sequencing operator in regular expressions as well as for the list operator in data instances. The semantics rule uses the auxiliary function $permutations(d)$ which returns the set of all permutations of a sequence of nodes d.

$$\frac{\begin{array}{c} t'_1, \ldots, t'_n \in permutations(t_1, \ldots, t_n) \\ \Gamma, R \vdash t'_1, \ldots, t'_n : \tau \end{array}}{\Gamma, R \vdash l\{t_1, \ldots, t_n\} : l\{\tau\}} \qquad \text{(UNORDEREDNODETYPING)}$$

Matching nodes with ordered content by types with ordered content specified in an unordered way Ordered node content can be matched by unordered specified ordered content the same way as unordered node content.

$$\frac{\Gamma, R \vdash l\{t_1, \ldots, t_n\} : l\{\tau\}}{\Gamma, R \vdash l[t_1, \ldots, t_n] : l[\{\tau\}]} \text{ (OrderedUnorderedSpecNodeTyping)}$$

Matching leaf nodes with basic data types Leaf data is matched by the basic data types, if the super type matches the data and if the constraints are also fulfilled. The matching of the basic data types, is defined in applications of the generic basic data type mechanism as introduced in section 3.4.2.2, hence not further treated here. Note, that some typing may also involve type coercion, e.g. each textual leaf node is per se a string, but due to numerical typing it may be *rescanned* and coerced to basic data of another type. However, coercion is an application dependant topic and not a matter at the level of the schema declaration language. The following rule just states the subtype condition as used for R_2G_2 and the delegation of basic type checking to the level where the basic data types and a notion of their language (denoted as $\mathcal{L}(\ .\ )$ are defined.

Chapter 4

From a Generic Module System to a Module System for R_2G_2

When many developers work on large scale grammars or if they want to reuse given languages as e.g. HTML, all the work can conceptually be merged together in one large grammar. Such large grammars are difficult to maintain, as the scope of type names is globally scoped over the whole grammar. Name conflicts are likely to occur and there is no separation of concern. To overcome the mess arising from large scale monolithic applications, modules for R_2G_2 are introduced now.

While conceiving the module system for R_2G_2 it turned out, that the module system in mind was more general and applicable to many rule languages only with marginal parametrization. As Xcerpt, the rule based query language to be extended with static typing using R_2G_2, neither had a module system, a general purpose module system has been developed and applied in two use cases to (1) Datalog [8] and also to (2) Xcerpt [7]. In this chapter, the general module system, as presented in [8] and [7], is briefly introduced and applied to R_2G_2.

4.1 The Purpose of Modules

From a practical point of view modules are very important for various applications especially when the data to be managed for those applications gets complex. Typical candidates of applications that need module support are query- , programming or schema languages as well as large scale CAD programs or even ontologies. In the following, modules will be only be considered for artificial languages like query or schema languages, yet the generalization to arbitrary applications with need for modules mostly not more than a replacement of language specific terms in the following paragraphs.

A module is a unit of language terms like e.g. a part of a query or a part of a schema. A module has not forcibly to be an usable instance of an application of that language, it is often a building block, library or skeleton for other applications. A module is itself a named entity, yet prone to (rare, maybe even intended) name conflicts. The purpose of modules is to

1. provide distinct name spaces for user defined names or symbols in terms of the language,
2. hide certain information or complexity in form of some names or symbols in the application,
3. provide an easy way to integrate different units or building blocks of an application with the goal of providing potential for reuse of code.

In the context of the Web *mhtml*[33] has been proposed as a module system for HTML and XHTML. The idea of modularization in HTML helps factoring HTML pages into reusable in the spirit of page templates and snippets. Using mhtml means (1) declaring modules—usually HTML documents or parts of them— and (2) combining them using the `import` clause referring to a module in the including module. To realize non trivial real world scenarios, parametrization is also needed: modules can be parametrized with named parameters that can be instantiated when importing with other modules, that are imported into the parametrized module. The approach can be compared with parametrized macro expansion as e.g. used by the C programming language pre processor. An implementation of *mhtml* exists using the *TOM*[1] pattern matching compile developed at INRIA[2].

4.2 Modules and XML Name Spaces

Pragmatically, the XML Name Space mechanism, has a lot in common with module systems. The purpose of name space handling is to distinguish equal names by partitioning the set of labels in sets that are likely to be named in a globally unique way. Name spaces in XML are realized as an extension of the naming scheme—label and attribute names are combinations of a so called local name part and a namespace part. Concluding, name spaces are identified again using names, yet delegating the name conflict problem to a higher level, but as one name space usually groups many (local) names, the *"name space"* name space is arguably less populated and name space identifiers may be chosen in a way less prone to ambiguity conflict—URIs are chosen as name space identifiers. As name spaces are URIs, and may therefore be very long strings, a substitution mechanism called *namespace prefixes* is used. Conceptually name space and the label name—also called local name—can be merged to one name by concatenating namespace (after namespace prefix expansion) and local name.

Name spaces have to be reflected when querying, constructing or schematizing data—labels of elements or attributes have to be composed of a local and a namespace part.

To compare module systems and XML Name Spaces, the three principles of module systems are recalled here: A module system (1) provides distinct name spaces for user defined names or symbols in terms of the language, (2) hides complexity, and (3) provides means for integration of components.

The common property of XML Namespaces and modules is the distinction of name spaces, point (1) in the former enumeration, in the sense of preventing name conflicts. Arguably to some extend point (3), integration of building blocks, is partly common to modularization and XML Namespaces—XML name spaces facilitate the combination of various XML languages in one document instance, modules facilitate the integration of different instance applications of a language. The module principle (2) though has no correspondence in XML Namespaces—hiding information in data is arguably not a reasonable property on the level of the data structure formalization, which XML is. However, visibility of data portions may be of interest for applications involving XML technology, like e.g. an XML database management system, but this is already at an application level from the "point of view" of XML. (XML related) languages or applications with need of name spaces hence may have two sorts of user definable names that have to be distinguished—names at the level of (XML) data (like element and attribute names) and names at the level of the language or application like variable-, method-, function-, predicate-, type-names and so on.

To a certain extend this distinction of modules and name spaces is weakened by the HTML module system mhtml mentioned earlier, where there is no distinction between data and program symbols and

[1]see `http://tom.loria.fr/` for the official project page of TOM.
[2]see `http://www.inria.fr/` for the homepage of INRIA.

no visibility rule—it is not relevant to distinguish symbols in HTML, as there is no potential naming conflict in HTML data—a module is a component of hypertext which is to be used at a given place.

4.3 Modular R_2G_2

A module extension to R_2G_2 is proposed. The purpose of the module system for R_2G_2 is the proper partitioning and integration of type names.

4.3.1 Syntax of Modular R_2G_2

The syntax is given as a set of EBNF grammar rules extending the current grammar rule set as presented in section 3.4 on page 56 ff.

Note, that the former top level non terminal *grammar* is now not the top level anymore, *module* takes the role of the top level non terminal now.

$$module \leftarrow grammar$$

A module may be used as a grammar used to be, it is then not identified as a module and not importable in any other module.

$$module \leftarrow visibility^{?} \text{ ”}\texttt{module}\text{”} \; identifier \; parameter^{*} \; use^{*} \; grammar$$

$$visibility \leftarrow \text{“}\texttt{public}'' | \text{“}\texttt{private}''$$

$$identifier \leftarrow URI$$

$$parameter \leftarrow \text{“}\texttt{parameter}\text{”} \; identifier$$

$$use \leftarrow \text{“}\texttt{use}\text{”} \; identifier$$

$$use \leftarrow \text{“}\texttt{use}\text{”} \; identifier \; \text{“}\texttt{as}\text{”} \; prefix$$

$$prefix \leftarrow label$$

A module is usually associated to a module identifier. A URI is used as module identifier. The visibility given in the module declaration is the default visibility of the types defined in the module, the visibility may be `public` or `private`, private types are not visible outside of the module. An *invisible* type can not be used in content models outside of the module defining the invisible type, it can also be called a local type. An *invisible* type can later on not be used for annotation in a typed Xcerpt program by the programmer, yet the system is able to infer it internally for the purpose of type checking.

Modules can be imported with the `use`-clause, they can also be imported in a qualified manner, in which case the visible types are imported in a qualified manner. Qualified types can be used in the scope of the importing module by prefixing them with the qualifier, it is up to the module author to choose a conflict free prefix (i.e. a prefix not occurring as prefix in other types or non terminals declared in the current module). Imported types, no matter if qualified or unqualified, are not reexported and are hence `private` in the importing module.

Parameters are used to declare prefixes, that are not bound to an imported module at the time of writing the parametrized module, but when the parametrized module itself is imported. A parametric module hence exposes an interface of types that the user of a module can *inject* into the used module by parametrizing it.

As an example, consider a container format representing a website consisting of a hierarchy of document containers containing just documents of a user selectable type.

```
module "WEBSITE-CONTAINER"
parameter "DOCUMENTDEF-";

element Container = container(name=String)[ ContainerContent ];
element ContainerContent = ((Container|Document)*);
element Document  = document(name="String)[ DOCUMENTDEF-DOCUMENT ];
```

Code Example 37 This is a parametric module modelling a website of homogeneous documents. The container may contain either (recursively) other containers or documents of the type `DOCUMENT`, which has to be declared in a module passed in for the parameter `DOCUMENTDEF-`.

The website container (see example 37) format is parametric about the type of the documents is may contain. The type of the payload document will be bound to the module prefix `DOCUMENTDEF-`. The *interface* of the module parameter is given by the types used from `DOCUMENTDEF-`, in the case of example 37 this is just `DOCUMENT`. Note, that the author of a module is advised to properly document the interface, there is no formal documentation mechanism. A module using the website container (see example 38) will bind the parameter do a schema declaring a type called `DOCUMENT`.

```
module "MY-WEBSITE"
use "WEBSITE-CONTAINER" where "DOCUMENTDEF-"="HTMLDEF";
root MySite;
element MySite = website(baseurl="http://example.com/mySite/")[ Container ];
```

Code Example 38 This module models a website using the parametric container model of example 37. The parameter is set to the module `HTMLDEF` defined in example 39, hence modelling a website consisting of a container with solely XHTML documents.

```
module "HTMLDEF"
use "XHTML";
content DOCUMENT = (HTML);
```

Code Example 39 This module is created for parametrization of the module in example 37. It is used in example 38

$$rule \leftarrow visibility^{?} \text{ ”}\texttt{element}\text{” } typename \text{ ”}^{*}{=}^{*}\text{” } typeterm \text{ ”}^{*};^{?}[\backslash\texttt{n}\backslash\texttt{t}]^{*}\text{”}$$

$$rule \leftarrow visibility^{?} \text{ ”}\texttt{type}\text{” } typename \text{ ”}^{*}{=}^{*}\text{” } typeterm \text{ ”}^{*};^{?}[\backslash\texttt{n}\backslash\texttt{t}]^{*}\text{”}$$

The *rule* as defined earlier is extended by visibility, that is optionally to be prefixed to a type declaration rule. If no visibility is prefixed, the default visibility as declared in the *module* rule is applied.

4.4 Realizing the Module System using the *"Divide and Rule"* approach

The generic module approach as presented in [8] is based on reduction semantics for module operators, i.e. modular rule language programs are translated into non modular programs. A necessary requirement

of the rule language to be modularized is the notion of some kind of chaining or rule dependency. In the case of R_2G_2 this is given by the dependency between right- and left hand side occurrences of non terminal symbols or type names. The term *rule dependency* as used in [8] is defined as follows:

Definition 4.1 (Rule dependency)
With a program P, a (necessarily finite) relation $\Delta \subsetneq \mathbb{N}^2 \times \mathbb{N}^2$ can be associated such that $(r_1, b, r_2, h) \in \Delta$ iff the condition expressed by the b-th body part of rule r_1 *is dependent* on results of the h-th head part of the r_2-th rule in P (such as derived data, actions taken, state changes, or triggered events), i.e., it may be affected by that head part's evaluation.

When applied to R_2G_2, there is just one rule part in the read—the type defined by the current rule, more precisely the type name. On the right hand side, the parts are the occurrences of type names. Using the rule dependency quadruple, the rule parts are mapped to natural numbers. Applied to R_2G_2 this could be according to the sequential order of the occurrence of the type name.

In example 40 an R_2G_2 'program' consisting of three rules is shown. When considering, that the rules are counted in sequential order, starting with index 1 as well as the body parts, the corresponding rule dependency quadruple set could be as follows:

$$\Delta_1 = \{(r_1,1,r_2,1),(r_2,1,r_3,1),(r_2,2,r_3,1),(r_3,1,r_3,1)\}$$

The first rule, denoted as r_1 has `B` as the first (and only) type on it's right hand side, and `B` is declared in the second rule (i.e. r_2). As all rules declare exactly one type, hence have one 'part' (i.e. non terminal or type) on the left hand side, the 4th component of the quadruples is always 1 for R_2G_2 rules. The rule r_2 has two parts on the right hand side (in this case both depending on r_3). this is reflected by two quadruples modelling the dependency of r_2. The third rule is a recursive rule, this is reflected by the fact, that the dependency modelled is a self-dependency.

The dependency presented in the last paragraph was a precise dependency, reflecting exactly which rule depends on which other rules. This however, requires a precise knowledge of the semantics of the rule language, which is not forcibly required by the generic module system. Assuming, that 40 is a module, another possible dependency reflecting the fact, that all rules in a module may depend on each other would be

$$\Delta_2 = \{(r_i, j, r_k, 1) | (i, j) \in \{(1,1),(2,1),(2,2),(3,1) \wedge 1 \leq k \leq 3\}\}$$

Δ_2 is the kind of dependency used further on for the module system.

```
element A = a[ B ];
element B = b[ C,C+ ];
element C = c[ cc[ C ]* ]
```
Code Example 40 A simple R_2G_2 grammar.

A module in [8] is defined as follows:

Definition 4.2 (Module)
A module M is a triple $(R_{priv}, R_{pub}, \Delta) \subsetneq \mathcal{R} \times \mathcal{R} \times \mathbb{N}^4$ where $\mathcal{R}$ is the set of all finite sequences over the set of permissible rules for a given rule language. We call R_{priv} the private, R_{pub} the public rules of M, and Δ the dependency relation for M. For the purpose of numbering rules, we consider $R = R_{priv} \diamond R_{pub}$[3] the sequence of all rules in M.

[3] $\diamond$ denotes *sequence concatenation*, i.e., $s_1 \diamond s_2$ returns the sequence starting with the elements of s_1 followed by the elements of s_2, preserving the order of each sequence.

The application to R_2G_2 is straight forward—public rules of an R_2G_2 module belong to R_{pub}, private ones to R_{priv}. If a module is declared as a public module, all rules without visibility annotation or with public visibility annotation are public, only rules with private visibility annotation are private. If a module is declared as a private module, all rules without visibility annotation or with private visibility annotation are private, only rules with public visibility annotation are public. The dependency relation without consideration of *use*d (or imported) modules is complete with respect to the rules of the module, i.e. each part on the right hand side of a module depends on all rules (more precise, it's head) in the module.

The *use* statements are treated by the so called scoped import, the one and only module composition operator in [8].

Definition 4.3 (Scope)
Let $M = (R_{priv}, R_{prub}, \Delta)$ be a module (or program if R_{pub} is the empty sequence). Then a set of body parts from M is called a scope in M. More precisely, a scope S in M is a set of pairs from $\mathbb{N}^2$ such that each pair identifies one rule and one body part within that rule.

Definition 4.4 (Scoped import $\rtimes$)
Let $M' = (R'_{priv}, R'_{pub}, \Delta')$ and $M'' = (R''_{priv}, R''_{pub}, \Delta'')$ be two modules and S a scope in M'. Then

$$M' \rtimes_S M'' := (R_{priv} = R'_{priv} \diamond R''_{priv} \diamond R''_{pub}, R_{pub} = R'_{pub}, \Delta'_{slided} \cup \Delta''_{slided} \cup \Delta_{inter}), \text{ where}$$

- $\Delta'_{slided} = \mathsf{slide}(\Delta', |R'_{priv}|, |R'_{pub}|, |R_{priv}|)$ is the dependency relation of the *importing* module M' with the *public* rules slided to the very end of the rule sequence of the new module (i.e., after the rules from M''),
- $\Delta''_{slided} = \mathsf{slide}(\Delta'', 1, |R''_{priv}| + |R''_{pub}|, |R'_{priv}|)$ is the dependency relation of the *imported* module M' with all its rules slided between the private and the public rules of the importing module (they have to be "between" because they are part of the private rules of the new module),
- $\Delta_{inter} = \{(r_1, b, r_2, h) : (r_1, b) \in S \wedge \exists$ a rule in R_{priv} with index r_2 and head part $h : r_2 > |R'_{priv}| + |R''_{priv}|\}$ the inter-dependencies between rules from the importing and rules from the imported module. We simply allow each body part in the scope to depend on all the public rules of the imported module. This suffices for our purpose, but we could also choose a more precise dependency relation (e.g., by testing whether a body part can at all match with a head part).

where slide is an operator used to renumber rules. The 'sliding' of the rule numbering is a pure technical tool used to get again unique numbering and fixing the dependencies, which gets necessary when concatenating sets of rules. The dependency slide is defined as follows:

Given a dependency relation Δ, slide computes a new dependency relation by sliding all rules in the slide window $W = [s+1, s+\mathit{length}+1]$ in such a way that the slide window afterwards starts at $s_{new}+1$:

$$\mathsf{slide}(\Delta, s, \mathit{length}, s_{new}) = \{(r'_1, b, r'_2, h) : (r_1, b, r_2, h) \in \Delta$$
$$\wedge\, r'_1 = \begin{cases} s_{new} + 1 + (r_1 - s) & \text{if } r_1 \in W \\ r_1 & \text{otherwise} \end{cases} \wedge\, r'_2 = \begin{cases} s_{new} + 1 + (r_2 - s) & \text{if } r_2 \in W \\ r_1 & \text{otherwise} \end{cases}\Bigg\}$$

After unwinding all the formal definition, importing a module is a mere concatenation of the rule sequences and a union of the rule dependencies, after proper re-indexing of the rules to prevent conflicts. Further on, for a scoped import $\rtimes_S$, the resulting dependency is extended by new dependencies derived by the information found in "S", such that the addressed rule parts of the importing rule set get dependencies to all the public rule heads declared in the imported rule set.

```
public module "MAIN-MODULE"
use "SUB-MODULE"
element A = a[ B , C ];
element C = c[ ]
```

Code Example 41 A simple R_2G_2 module that uses another module called 'SUB-MODULE'.

```
public module "SUB-MODULE"
element B = b[ C ];
private element C = ccc[ ];
```

Code Example 42 A simple R_2G_2 module with one private rule and one public (by default) rule.

As an example consider the two modules presented in the examples 41 and 42. Formally, these modules, without considering the module use statements, can be denoted as

$$R_{\texttt{MAIN}} = (\{\}, \{1,2\}, \{(1,1,1,1), (1,1,2,1), (1,2,1,1), (1,2,2,1)\})$$

$$R_{\texttt{SUB}} = (\{2\}, \{1\}, \{(1,1,1,1), (1,1,2,1)\})$$

The *use*-statement of example 41 is not a scoped import. However, the scoped import is more general as the default *use*-statement and it can be translated to a scoped import—each rule and each part of the importing module gets scoped. For 41 the *use*-statement hence results in expressing a scoped import $R_{\texttt{MAIN}} \rtimes_{(1,1),(1,2)} R_{\texttt{SUB}}$. Application of the scoped import operator in the end results in one *use*-statement-free module as follows:

$$R_{\texttt{MAIN}} \rtimes_{(1,1),(1,2)} R_{\texttt{SUB}} = \left(\begin{array}{c} \{4,3\}, \{1,2\}, \\ \left\{ \begin{array}{l} (1,1,1,1), (1,1,2,1), (1,2,1,1), (1,2,2,1), \\ (3,1,3,1), (3,1,4,1), \\ (1,1,3,1), (1,2,3,1) \end{array} \right\} \end{array} \right)$$

So, the resulting module is a concatenation of all modules with a restricted rule dependency (compared to the plain module concatenation). A promise of the generic module system is, that base don the reduction semantics, no change in the core module language is forcibly necessary. How can this goal be achieved, i.e. how can the controlled rule dependency be realized? A simple way in the case of R_2G_2 is the realization by rewriting: the type names on the left hand side are rewritten such that they reflect the separation of modules, e.g. by prefixing a module identifier, those on the right hand side are replaced by disjunctions for all modules involved in dependencies to the body part in question, e.g. a disjunction consisting of all involved module identifiers postfixed by the former type name. As an example, the resulting module of applying the module system reduction to the examples 41 and 42 is presented in example 43

```
element MAIN_A = a[ (MAIN_B|SUB_B) , MAIN_C ];
element MAIN_C = c[ ]
element SUB_B = b[ SUB_C ];
element SUB_C = ccc[ ];
```

Code Example 43 An R_2G_2 grammar resulting from the application of the generic module system to the examples 41 and 42. The dependency modification is achieved by type renaming, i.e. by prefixing the (unique) module identifiers to the type names. To preserve the original type names for the outside, type aliases can easily be used. Note, that the first rule uses an undefined type name, MAIN-B, for technical reasons. Orphaned right hand side type names are however easily detected and can always be removed from the grammar.

Further features of the module system not presented in details, i.e. not demonstrated by example are:

qualified modules Arguably, qualified modules are exactly what the scoped import is necessary for, as it gives rise to express that the imported rules get visible only for the body parts using the qualifier. This can be expressed using rewriting by selectively applying the rewriting to the qualified body parts the same way as presented in example 43 for all body parts.

parametrized module import parametrized module use is similar to qualified module use, except that the assignment of a module to a qualifier does not occur in the declaring module, but in the using module. As the module reduction takes place on all modules, globally and at compile time, this is not a big deal for the generic module system—the assignment of a module to a parameter is just postponed until bound for a module use. Note however, that for multiple imports (in possibly different modules) of the same parametrized module, multiple instances of the rules using the parameter have to be created during reduction, as they possibly have dependencies to different types. Right hand side occurrences of the type declared by the multiple instances are accordingly selecting their corresponding rule instance.

parametric modules A parameter of a parametric module is treated exactly the same way as a qualified module use, after binding of the parameter.

Chapter 5

Use Cases—Modelling Data and Documents with R_2G_2

5.1 Beyond Regular Tree Grammars—The Use of Macros

Arguably in practice, especially for larger real word applications, any possibility of factorization of code is welcome and used by programmers. When modelling content models, a way of factoring out common content of different element types is to use content macros.

The normative Schema of XHTML[1] is written in Relax-NG employing a large number of grammar rules, that correspond to the R_2G_2 macro rules. Macros are used to factor out content elements as well as common attribute declarations. The XHTML draft defines different modules, represented by different schema modules that are imported, partly overridden in sub modules and aggregated to the complete XHTML schema. The separation of modules is useful to provide small units of XHTML that can be used in other XML schema definitions, e.g. the hyperlink mechanism is defined as a separate module and can thus be reused e.g. for another document markup language. The extract in example 44 of R_2G_2 declarations corresponds to the parts of the XHTML 2.0 schema draft [49] used to declare hyperlinks:

```
module "hypertext-module"
use "text-model-module"
use "common-attributes"

element A = a(A.AttList)[ Text.Model ] ;
alias A.AttList = Common-Attrib ;
alias Text.Class = Text.Class | A ;
```

Code Example 44 The example is mostly a transcription of the XHTML module for hyperlinks written in Relax NG. Comprehension is rather easy, as most of the irrelevant (in this context for comprehension) declarations are located in other modules.

The simplicity of the definition is obviously due to modularization.

[1] Either 1.0 or 2.0, even the HTML DTD's share this properties.

5.2 Design Patterns—R_2G_2 Best Practices

In the spirit of the well known "XML Schema best practices" [23], some design patterns for R_2G_2 will be presented. Many of them are inspired by the XML Schema counterpart.

5.2.0.1 Design Objectives

In the style of the design objectives document of "XML Schema best practices", three design principles can be distilled:

- The instance author's power vs. rigidness of the schema, or to which extend the most general type is used, such that an instance author may choose which type of data to use in certain contexts.
- Reusability, or to which extend may a user reuse the schema declarations for another schema in e.g. another namespace.
- Extendability, or to which extend should the schema be extensible by third parties, e.g. using parametrization.

The **first principle** is **freedom of data and structure choice for the document author**. This can can be divided in two aspects:the freedom of choice concerning atomic data, i.e. arbitrary CDATA in terms of XML, and the freedom of choice concerning structured data, i.e. arbitrary, maybe untyped or schema-less XML sections of the document. The first aspect is mostly known since DTD, by providing PCDATA as content type. In R_2G_2 this is achieved using either a most general regular expression at content level or by using the predefined *string* data type.

```
element AnyText1 = /.*/ ;
element AnyText2 = r2g2:String ;
```

To model arbitrary structured content, the R_2G_2 schema author can use the predefined *Any* type, which represents exactly one arbitrary element or text node. Defining the *Any* type using R_2G_2 is also possible and actually not complicated, as shown in the following example:

```
element Any = /.*/( (/.*/ = /.*/)* )[ Any* ]
element Any = /.*/( AnyAttr* )[ Any* ]
            | /.*/( AnyAttr* ){ Any* }
            | r2g2:String ;
attribute AnyAttr = /.*/ = r2g2:String ;
```

The **second design principle** is **reusability**. A schema author has to decide to which extend the schema is exposed to other schema authors, such that they can easily include the schema in their own schema declarations. In R_2G_2 the schema author has control over reuse of his schema declarations by controlling the visibility of type names for users of a module. Only visible type names can be used in other grammars or applications. A further notion of reusability is to let the namespace of elements unspecified, thus giving users of the module the ability to fully integrate the declarations in their own schemata, using e.g. their own namespace. Having homogeneous name spaces in a schema is desirable from the point of view of the document authors, as it arguably eases reading and authoring of a document. A disadvantage of a schema not bound to a specific namespace, yet being bound to different name spaces of hosting schemata is that a query author or a processing application needs to be aware of all possible hosting name spaces to support such a variable name space document component.

The **third design principle** is **extendability**. Third party schema authors have to provide the schemata for the components of a given schema that models a container like structure. To achieve this in R_2G_2 parametric modules can be used. The feature is roughly comparable to the term *dangling type* as used in the 'XML Schema best practices' documents—dangling types are types used but not declared in a schema (they have to be declared by the user of the schema). A using schema instance hence has to *implement* the module parameter by providing a module with the type declarations for the type names in question. As an example, consider the website container format examples 37, 38 and 39 presented in section 4.3.

5.2.1 Global versus Local

In the 'XML Schema best practices' documents, this issue is about when and how to declare elements locally and when to declare them globally. In this context the terms *local* and *global* are used to distinguish named types, i.e. types declared with a rule (called global types) and those declared in-line as child type terms used in the content model of it's parent type term (called local types). Using R_2G_2 the declaration of local element definitions can be achieved in two ways: indirectly, by declaring element types using the `private` visibility modifier for a rule or directly by declaring element types locally as child elements in an element declaration of another element. For global declarations, the declared element has to be in a publicly visible rule. The direct modelling of local definitions is similar to the way of locally defining elements in XML Schema using the Russian doll principle, and the principle is directly applicable to R_2G_2. When modelling schemata using the **Russian doll** paradigm, elements are declared in a nested way in a type term.

The grammar in example 45 declares an address book in the spirit of the Russian doll paradigm.

```
root AddressBook;
element AddressBook = addressbook{
 card{
    name[ /.*/ ],
  ( phone[ /.*/ ]
  | email[ /.*/ ]
  | im-contact[ type[ /(icq)|(msm)|(aim)|(irc)|(yabber)/ ] ,
                user-id[ /.*/ ] ]
  | address[ street[ /.*/ ] ,
             detail[ /.*/ ] ,
             city[ /.*/ ] ,
             zip-code[ /.*/ ] ,
             state[ /.*/ ]? ,
             country[ /.*/ ]? ,
    ]
  )+
 }*
};
```

Code Example 45 An address book modelled in the spirit of the Russian doll design principle—the child node types of a node are declared using type terms as child terms of their corresponding parent type.

The advantage of the Russian doll paradigm is, that the schema reflects, almost like an example, the structure of instance documents—element declarations occur in similar nesting context as their instances. A disadvantage of the Russian doll approach is, that multiple occurrences of the same type results in multiple declarations of that type. Note, that the Russian doll paradigm, does not allow to model data with recursive declarations, as a circular structure referring a declaration inside the declara-

tion (possibly at deeper nesting level) is necessary, but the reference of a type name makes it necessary to use a rule for the declaration of that type name.

The opposite of the Russian doll paradigm is the so called **salami slices** paradigm, where each element type declaration is represented by an own rule, very much the traditional way of formal tree grammars as found in section 3.1, 3.2 or [22].

Example 46 presents the same document type as for the Russian doll example (see example 45) modelled using the salami slices paradigm. Each element declaration is associated with an own type name, hence exporting of all symbols can easily be achieved.

```
root AddressBook;
element AddressBook = addressbook{ Card* } ;
element Card        = card{ Name ,
                            (Phone|Email|IMContact|Address)* } ;
element Name        = name[ /.*/ ] ;
element Phone       = phone[ /.*/ ] ;
element Email       = email[ /.*/ ] ;
element IMContact   = im-contact[ IMType , UMUID ] ;
element IMType      = type[ IMTypeToken ] ;
element IMTypeToken = /(icq)|(msm)|(aim)|(irc)|(yabber)/ ;
element AnyText     = /.*/ ;
element IMUID       = user-id[ AnyText ] ;
element Address     = address[ Street , Detail ,
                                 City , ZipCode ,
                                 State? , Country? ] ;
element Street      = street[ AnyText ] ;
element Detail      = detail[ AnyText ] ;
element City        = city[ AnyText ] ;
element ZipCode     = zip-code[ AnyText ] ;
element State       = state[ AnyText ] ;
element Country     = country[ AnyText ] ;
```

Code Example 46 An address book modelled in the spirit of the salami slices design principle—all types are declared employing rules at top level of the grammar, the child node types of a node are referenced in its content model using the type name declared in the corresponding rule.

In the 'XML Schema best practices' document, a third paradigm between Russian dolls and salami slices is presented, the so called **venetian blinds**. In XML Schema, a type is not bound to an element declaration but to a content model. An element declaration hence involves a label declaration and a type declaration. The venetian blinds pattern consists in referencing type declarations in element declarations, that occur themselves nested in type declarations. This contrasts to the salami slices paradigm, where element declarations are referenced in type declarations. The concept is applicable to R_2G_2 using `content` non terminals.

Example 47 illustrates the same address book as used in the examples 45 and 46 in the spirit of the venetian blinds paradigm using content macros—the only necessary explicit element declaration is the root element declaration:

```
root AddressBook;
element AddressBook = addressbook{ card{ Card }* } ;
content Card        = name[ TXT ] ,
                      (phone[ Phone ]
                      |email[ AnyText ]
                      |im-contact[ IMContact ]
                      |address[ Address ]
                      )* ;
content IMContact   = type[ IMType ] , iser-id[ AnyText ] ;
content IMType      = /(icq)|(msm)|(aim)|(irc)|(jabber)/ ;
content AnyText     = /.*/ ;
content Address     = street[ AnyText ] ,
                      detail[ AnyText ] ,
                      city[ AnyText ] ,
                      zip-code[ AnyText ] ,
                      state[ AnyText ]? ,
                      country[ AnyText ]? ;
```

Code Example 47

Note, that `content` is a mere macro pre processing mechanism, therefore the content non terminals are not type names, that can be accessed e.g. in validated and type annotated data instances. Similar to the Russian doll example, only one element type is declared in this example, hence all other implicit element declarations are anonymous and may not be exported for import or explicit type annotation. In XML Schema the different paradigms are of essential importance for the exposure or hiding of declaration. In R_2G_2 dedicated constructs for import, export and namespace handling have been introduced, rendering the exposure and hiding aspect of the modelling style less relevant, yet leaving the freedom of taste to the schema author. On the other hand, such structural properties of XML Schemata may be reflected in R_2G_2 type declarations making transitions of legacy schemata easier.

Some rules of thumb for the applications of the different paradigms are given here, yet they only reflect the authors preferences:

- Introduce a new type name when the type is reused in different contexts
- When modelling schemata with querying in mind, use more type names, as when modelling for document validation, as this exposes the possibility of type annotation to the query programmer
- Try to expose specialized textual content types as they are most likely to be queried or transferred to constructions. Further on, regular text expressions of characters are arguably harder to read for document or data authors than well chosen type names.
- Hide type names of types that are still in development
- When elements with different labels always share the same content and the elements have similar semantics, it is maybe advisable to use a label regular expression in one and the same type declaration, instead of using one content declarations and many element type declarations.
- Try to factor out common parts of content models in content declarations.

5.2.2 Composition vs. Sub Classing

In XML Schema a powerful object oriented modelling feature that is arguably orthogonal to the grammar philosophy has been introduces—modelling by sub classing. When modelling by sub classing, a new type can be derived of an existing type, by *restricting* the set of elements belonging to the super class

of the subclass. This feature is powerful in the sense of object oriented modelling or programming, as it allows the programmer or modeller to specialize the behaviour of a new class of objects based on the behaviour of a base- or super class. In XML Schema, sub classing or restriction is merely more than a contract of a type to fulfill the subset property of its base- or super class, i.e. the author of a sub type has to model a type in such a way, that it fulfills a stated subset, base type or restriction property, an XML Schema validator has to check the given type for conformance to that property. The advantage for a program operating on data structures fulfilling the given subclass property are the same as for object oriented programming—the program can rely on compatibility of operation on subtypes for operations designed for the super type of the subtype. For the Schema author, no benefit exist over modelling data without the subtype support. However, for a query author for example it is valuable to know about the subtype property of given types, as this ensures the applicability of queries written for a super type to elements of the subtype. Arguably, type checking does not really need to rely on sub typing information for useful operation, as we will see in the chapter about type checking of Xcerpt—type inference is able to deduce the super type of it's subtypes without the information about it. When modelling elements of a given type, the benefit of automatically being able to use the subtypes of the given type exists. However, the base type has to be modelled from the beginning in a way to easily be extensible, otherwise it has to be altered or widened along new sub classes.

Interestingly, even though the object oriented principle of sub classing has been introduced to ease a separation of concern, a tight integration of complexity and so called tight coupling is the result. Let us assume as an example a database of a camera vendor modelled following a sub classing paradigm: starting as a vendor of cameras, there exist a type `Camera` with all relevant properties. When digital cameras came to the market, some (analogue) camera properties got irrelevant while other digital camera properties gained relevance. This logically leads to a sub classing of camera, where some of the former camera features have to be shifted into a subclass of camera, `AnalogCamera`, while digital camera features are aggregated in the `DigitalCamera` sub class. As the store may also sell tripods as non optical devices, sound recording devices or projectors as optical device along with cameras, a consequence for the schema designer is to provide a super class to the `Camera` class—the `OpticalDevice` class. When altering the optical device class, potentially all camera classes may have to be adapted along with valid instances, which is a consequence of the so called tight coupling.

An alternative approach to sub classing is called *compoisition* or *loose coupling*. When modelling using composition and loose coupling, the main principle is to encapsulate the differences instead of abstracting away the similarities of objects. In XML and related Schema languages, this is achieved by extending the content model by adding new elements or element containers to a given type, possibly optional to retain backward compatibility. In the camera vendor example, the difference between digital and analogue camera could be modelled either as an alternative between analogue and digital camera child members, or as a common abstract container for the `RecordingMedia`. This way altering the data model will not affect e.g. 'camera container structures', as the modification is internal to the camera.

> Unquestionably, design by sub classing has an important role to play in XML Schema design. However, it is being greatly overused. Long, extended type hierarchies lead to brittle, non-modifiable designs that are virtually impossible to understand.
>
> Design by composition is the preferred approach. It yields simpler, robust, modifiable, plug-and-play designs.
>
> "Favoring element composition over type inheritance helps you keep each element encapsulated and focused on one task. Your type hierarchies will remain small and will be less likely to grow into unmanageable monsters." [23][2]

[2] In the "XML Schemas: best practices" collection of documents the cited part resides in the document "Composition versus

As a consequence, the concept of sub typing by restriction is not included in R_2G_2, yet due to the regularity of the language it is easy for external applications to ensure the subtype property of R_2G_2 type declarations. Sub typing is a mayor concept for type checking of query programs using R_2G_2 types.

5.2.3 'eXtreme eXtensibility'

A design pattern presented in the *"XML Schema best practices"* document is the *'eXtreme eXtensibility'* pattern. The concept behind is to couple a schematized, typed or structured world with the aspect of freedom of semi structured data, such that no restriction on the structure of data is given, except the ones already available. As a concrete example think of a schema for bibliographical information: as the first party may provide a schema for data containing just author and title along with the data, another party may provide extended information containing e.g. the publisher and the number of pages of books. A third party again, may provide based on the first schema an extension for e.g. a review of books.

Technically, the concept is based on the use of a type disjunction containing the most general type in a repetition sequence to allow arbitrary elements (represented by the most general type) along with well defined elements (the other disjunctive parts). It is however advisable to restrict the most general type to a namespace specially devoted to the purpose of being *the namespace of arbitrary extension elements* for the concrete application in mind. Third party document and schema authors hence declare the elements they need in that name space and thus extend the schema to their need.

The following example models the previously mentioned librarian database with first the base schema and a document instance, second and third two derived schemata and document instances. Last, an aggregation of all the schemata and documents is presented, which is a common usage scenario of documents and schemata applying the 'eXtreme eXtensibility' pattern.

```
default namespace http://book.ext.ext.org/ ;
namespace ex = http://bookext.ext.ext.org/ ;

root BookDB;

element BookDB = literature{ @Book* };
element Book = book{ BookContent* };
content BookContent = (ex:AnyBookContent
                      |Title
                      |Author) ;
element Title = title[ r2g2:String ] ;
element Author = author[ r2g2:String ] ;
element ex:AnyBookContent = ex:/.*/{
                              r2g2:Any
                             } ;
```

Code Example 48 The grammar models a bibliographic database following the "eXtreme eXtensibility" paradigm. It is a container for book-elements where no more is given, than that it contains a title and an author element and any element, as long as it is in the namespace given by the prefix ex in this example.

Subclassing" at http://www.xfront.com/composition-versus-subclassing.html

```
use http://book.ext.ext.org/schema.r2g2 ;
namespace bdb = http://book.ext.ext.org/ ;
defaultnamespace http://bookext.ext.ext.org/ ;

root bdb:BookDB;

element NrOfPages = pages{ r2g2:Integer } ;
element Publisher = publisher{ r2g2:String } ;
```

Code Example 49 A grammar defining types for the number of pages of a book and for information about a publisher. The types have been designed to be used in a bibliographic database as shown in example 48. The contract between the schemata committed to the "eXtreme eXtensibility" paradigm is to use a given namespace.

```
use http://book.ext.ext.org/schema.r2g2 ;
namespace bdb = http://book.ext.ext.org/ ;
defaultnamespace http://bookext.ext.ext.org/ ;

root bdb:BookDB;

element Review = review[ r2g2:String ] ;
```

Code Example 50 Another grammar defining a type for the extension of the grammar in example 48.

```
<literature>
  <book id="isbn:0007110472">
    <title>Three to See the King.</title>
    <author>Magnus Mills</author>
  </book>
  <book id="isbn:0810117312">
    <title>His Master's Voice</title>
    <author>Stanislaw Lem</author>
  </book>
</literature>
```

Code Example 51 An instance document valid with respect to the grammar of example 48.

```
<literature>
  <book id="isbn:0007110472">
    <pages>167</pages>
    <publisher>HarperPerennial</publisher>
  </book>
  <book id="isbn:0810117312">
    <pages>199</pages>
    <publisher>Northwestern University Press</publisher>
  </book>
  <book id="isbn:0863697313">
    <pages>290</pages>
    <publisher>Virgin Books</publisher>
  </book>
</literature>
```

Code Example 52 An instance document valid with respect to the grammars of example 48 and 49.

```
<literature>
  <book id="isbn:0007110472">
    <title>Three to See the King.</title>
    <review>
       Novella-like in form, Magnus Mills' Three
       to See the King is an uneasy read that ...
    </review>
  </book>
  <book id="isbn:0684865114">
    <title>The Restraint of Beasts</title>
    <review>
       Building high-tension fencing with a couple
       of rural Scots louts--what could be a more ...
    </review>
  </book>
</literature>
```

Code Example 53 An instance valid with respect to the grammars of example 48 and 50 used together.

```
<literature>
  <book id="isbn:0810117312">
    <pages>199</pages>
    <publisher>Northwestern University Press</publisher>
  </book>
  <book id="isbn:0863697313">
    <title>His Master's Voice</title>
    <author>Stanislaw Lem</author>
    <pages>290</pages>
    <publisher>Virgin Books</publisher>
  </book>
  <book id="isbn:0007110472">
    <title>Three to See the King.</title>
    <pages>167</pages>
    <publisher>HarperPerennial</publisher>
    <review>
       Novella-like in form, Magnus Mills' Three
       to See the King is an uneasy read that ...
    </review>
  </book>
  <book id="isbn:0684865114">
    <title>The Restraint of Beasts</title>
    <review>
       Building high-tension fencing with a couple
       of rural Scots louts--what could be a more ...
    </review>
  </book>
</literature>
```

Code Example 54 This instance document could have been generated by an aggregation service (e.g. written as a web query). It is the join of all three document examples 51, 52, and53. As a consequence of the "eXtreme eXtensibility" approach, the document is valid with respect to the union of all the involved grammars, i.e. the examples 48, 49, and 50.

A drawback of this approach is, that name conflict may arise in the extension name space. An advantage of this approach is the high flexibility and the ease of mering different document instances due to common outer structure.

Relationship between *'eXtreme eXtensibility'* and RDF The application scenario presented in the example and those of for 'eXtreme eXtensibility' is interestingly related to RDF: using 'eXtreme eXtensibility', it is possible to describe different aspects of the same concept in a distributed manner, providing means of aggregating the knowledge. RDF to some extend has the same goal—describing properties of subjects (identified using URI's) using a triple like relation between subject and object along a predicate. A proposed W3C recommendation—the *RDF/XML Syntax Specification*—provides a very loose schema for XML documents and an interpretation in RDF of the data of that document. The main idea is to have a kind of striping in the depth of subject/predicate/object, such that

- the subject is an element (directly under the root element[3] identified, e.g. using an id attribute.
- the child nodes of a subject are predicates
- the child node (just one is allowed) is the object of the subject/predicate/object relationship, it may be CDATA.

Arguably, RDF/XML is not of high relevance in providing an RDF interpretation of legacy XML applications, as they are very unlikely to fulfill the RDF/XML syntactic requirement and even less likely to provide RDF with meaningful semantic. For new applications however, RDF/XML provides a way to defile an XML application that at the same time yields an RDF application, when well designed.

The presented example application fulfills the RDF/XML requirement. The data may be interpreted as RDF data. The schema however gives no hint about that. A Schema for data fulfilling the RDF/XML recommendation must be the schema of a language, that is a subset of the RDF/XML specification. Unfortunately, due to the high freedom and flexibility of the label naming conventions, it is neither possible to give a general schema for RDF/XML using XML Schema, nor DTD, nor Relax NG. Using R_2G_2 it is possible. The distinguishing feature is the regular expression based element label specification. As it is possible to check the subtype (or language subset) relationship between two R_2G_2 specifications, it is possible to check an R_2G_2 schema for RDF/XML conformance. Note, that this does not forcibly lead to schemata or documents (conforming those schemata) with useful RDF interpretation, it is mere a useful tool along the step of defining XML languages fulfilling also the RDF/XML specification.

[3]Subjects may also occur in other, deeper locations in the document, please refer to `http://www.w3.org/TR/2003/PR-rdf-syntax-grammar-20031215/`.

Part II

Automata Models

Chapter 6

An Automaton Model for Regular Rooted Graph Languages

An automaton model used for validation and type checking with languages defined using R_2G_2 is presented. First, tree-shaped data is considered to be handled by the automaton model, then the approach is extended to graph shaped data. The presented approach is based on specialized non-deterministic finite state automata. The specialisation copes with unranked tree shaped data. Graph shaped data will be treated as, possibly infinite in depth, trees.

The choice of using non-deterministic automata is motivated by complexity issues: as the tree automata are based on regular expressions, non-deterministic automata are a necessary intermediate step. Arguably deterministic tree automata are more efficient on validating data, but the derivation of such automata from non-deterministic ones comes with potentially exponential costs. As all the needed algorithms can be achieved on non-deterministic automata in sub-exponential time and space complexity, no need to transform to deterministic automata arises.

6.1 Introduction to Regular Tree Automata

Traditionally, regular tree automata are defined as follows (cf. [22]).

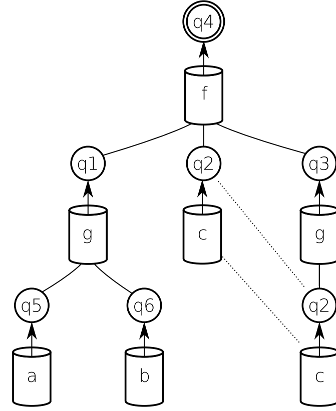
Definition 6.1 (Non-deterministic Finite Tree Automata)
A non-deterministic finite tree automaton (NFTA) over Σ is a tuple $A = (Q, \Sigma, Q_F, \Delta)$, where Q is a set of (unary) states, $Q_F \subseteq Q$ is a set of final states, and Δ is a set of transition rules of type $f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n))$, where $n \geq 0$, $f \in \Sigma_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in X$.

The set Σ contains the symbols or the alphabet of the tree. Note, that traditionally regular tree automata operate on ranked trees, therefore the symbols have fixed arity—the number of child nodes in a corresponding tree is fixed. The set $\Sigma_p \subseteq \Sigma$ is the set of all symbols in Σ with arity p. The set $T(\Sigma)$ denotes the set of all tree that can be constructed using the symbols in Σ. Therefore

- $\Sigma_0 \subseteq T(\Sigma)$
- for $p \geq 1$, if $f \in \Sigma_p$ and $t_1, \ldots, t_p \in T(\Sigma)$, then $f(t_1, \ldots, t_p) \in T(\Sigma)$

Example 6.1
A non-deterministic[1] finite tree automaton able to recognize a language containing (under many others) the tree $f(g(a,b),c,g(c))$ is generated. The figure on the right of the automaton informally illustrates the relationship between the states,[2] transitions and the data tree: a transition is denoted as a kind of *tube*. If some sub-trees of the data tree have been recognized the automaton is in corresponding states. A transition is used, if the automaton is in all the corresponding input states (in the example below the tube) and the father node of the sub-trees recognized with those states is labeled like the tube. The automaton is then not any longer in the states below the transition, but in the target state (precisely in all the target states of all the transitions traversable in that step). The root of the tree has to be accepted in such a way, that the resulting state is a final state. Note, that the two instances of the c transition and the state q_2 denote the same objects in the automaton, they have been duplicated to illustrate acceptance of the input data that contains two sub-tree accepted by the same transition.

$$
\begin{array}{lll}
A = & \{ & \{q_1,q_2,q_3,q_4,q_5,q_6,q_7,q_8\} \\
& , & \{f_{/3},a_{/0},b_{/0},c_{/0},g_{/2},g_{/1}\} \\
& , & \{q_4\} \\
& , & \{f(q_1(X),q_2(Y),q_3(Z)) \rightarrow q_4(f(X,Y,Z)) \\
& & ,g(q_5(X),q_6(Y)) \rightarrow q_1(g(X,Y)) \\
& & ,c \rightarrow q_2(c) \\
& & ,a \rightarrow q_5(a) \\
& & ,b \rightarrow q_6(b) \\
& & ,g(q_2(X)) \rightarrow q_3(g(X)) \ \} \\
& \} &
\end{array}
$$



Acceptance Procedure The acceptance procedure recognizes, if a given tree is member of the tree language represented by a given automaton. A tree t is in the language $\mathcal{L}(A)$, if it is accepted by A. The acceptance procedure can be defined as non-deterministic algorithm expressed by a set of rules. The rules relate so called configurations of an automaton to each other. A configuration is a tree on which some nodes are annotated with a state, more formally $c \in T(\Sigma \cup Q)$—note, that Q is defined as unary states, e.g. a state can be seen as a (special) node in a tree with exactly one child node.

The rules have the following general shape:

$$
\frac{\begin{array}{c} C_1 \\ \vdots \\ C_n \\ t \end{array}}{t'} \qquad \text{(EXAMPLERULE)}
$$

where t and t' denote configurations. C_i denotes constraints on the configurations, part of them or their sub trees. The style of rules presented here is inspired by Gentzen or tableaux calculus rules and is often used in the context of type system formalization [41]. Whenever t is matched in the current configuration, t can be replaced by t'. The rules are applied until no rule is applicable anymore resulting in a

[1] Indeed it is deterministic, but the difference is not relevant at the moment.
[2] It corresponds loosely to what will later on be introduced as *"aggregated acceptance path in the derivation tree"*.

sequence of configurations. If it is possible that more than one rules is applicable on one configuration (which is usually the case), a tree of possible configurations exists with sequences of configurations as paths through the tree.

The use of rules to express the acceptance procedure with finite automata is not common, yet useful to introduce the rule formalism, that will be used throughout this deliverable in different places.

Rules for Acceptance Procedure based on the Finite Tree Automata A given tree t is member of a language $\mathcal{L}(A)$ for an automaton $A = (Q, \Sigma, Q_F, \Delta)$, if there is a derivation of configurations based on the following rules with at least one closed branch of the derivation tree.

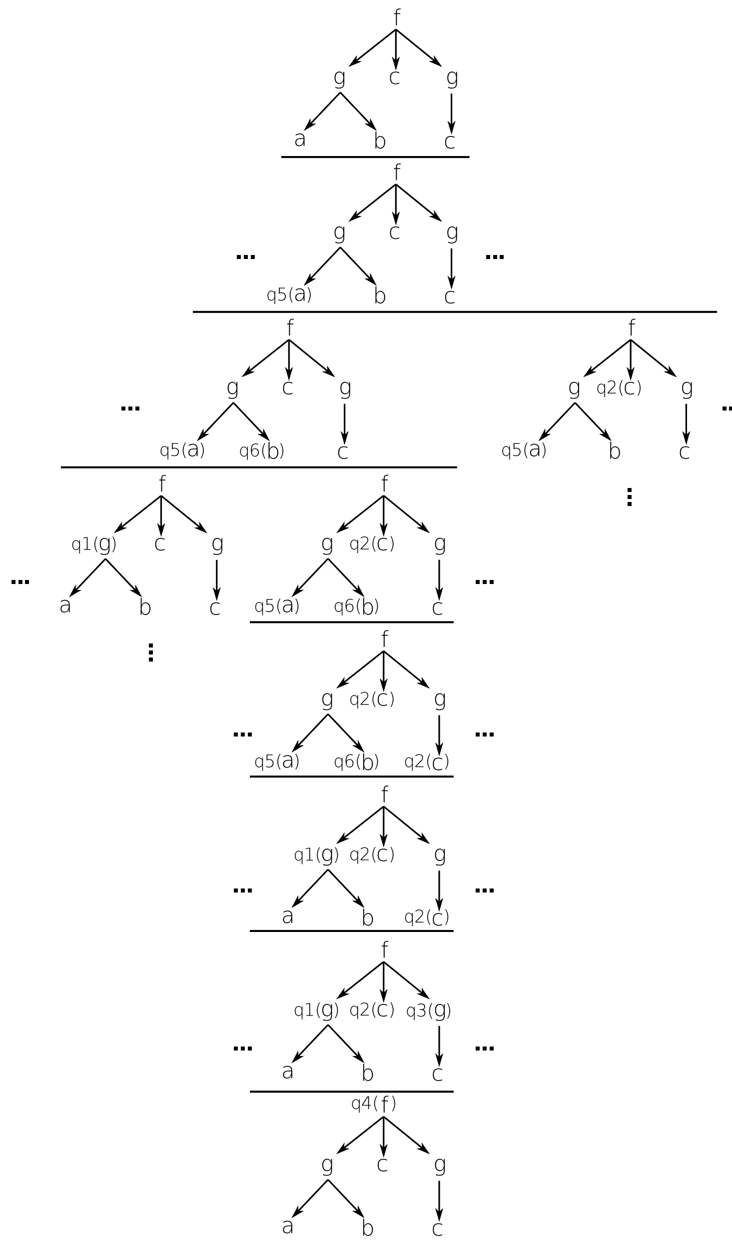
$$\frac{\begin{array}{c} q \in Q_F \\ t \in T(\Sigma) \\ q(t) \end{array}}{} \quad \text{(ROOT)}$$

The first rule states, that the given tree t is accepted, when a configuration is derivable such that t is accepted with a final state $q \in Q_F$. A branch of configuration derivations is successfully closed. At least one successfully closed branch is necessary to prove membership of a given tree in the language represented by the given automaton.

$$\frac{\begin{array}{c} u_i \in T(\Sigma) \\ f \in \Sigma_n \\ f(q_1(X_1), \ldots, q_n(X_n)) \rightarrow q(f(X_1, \ldots, X_n)) \quad \in \quad \Delta \\ q, q_1, \ldots, q_n \in Q \\ f(q_1(u_1), \ldots, q_n(u_n)) \end{array}}{q(f(u_1, \ldots, u_n))} \quad \text{(REC)}$$

The Rec rule relates two configurations, if the tree contains a (sub)tree matching the left hand side of a transition in Δ. The (sub)tree is then replaced by the sub-tree on the right side of the transition rule with all variables (e.g. X_i) substituted with the bindings of the left hand side. The only difference in the two configurations is the annotation of nodes with states. This is due to the nature of the transitions—left and right hand side are identical except of the change of intermediate state labeled branch parts.

Example 6.2
Given the tree $f(g(a,b), c, g(c))$ and the example automaton A presented above, the following derivation justifies the recognition of the tree as an instance of the language represented by A:



Aggregated Acceptance Path of the Derivation Tree Given a path of rule applications that proves the membership of a tree in the language of the corresponding automaton, the aggregated acceptance path is the tree resulting, when aggregating all the configurations of the path to one configuration such that all state annotations interlaced with the path are part of this configuration. This *artificial* configuration gives the information, which node was accepted with which transition. The transition can then be seen as some sort of type annotation for the nodes of the tree. Later on (see chapter 8), for type checking

Xcerpt, this is used to deduce the types of nodes, as the transitions are shown to be related to grammar rules and therefore to grammar non terminal symbols which in turn represent types or type names.

ε-rules It is possible to extend the non-deterministic regular tree automata with ε-rules. Those are rules of the form $q \rightarrow q'$. Yet ε-rules are convenient in some cases (e.g. for construction of an automaton based on some regular expression like formalism as shown in 6.3), they do not enhance or restrict the expressiveness of non-deterministic regular tree automata.[3]

deterministic finite tree automata Another common variant of non-deterministic finite tree automata are *deterministic finite tree automata*. A tree automaton $A = (Q, \Sigma, Q_f, \Delta)$ is deterministic (DFTA) if there are no two rules with the same left-hand side (and no ε-rules). Many text book approaches of standard operations on automata like intersection and union require deterministic automata. It is always possible to get a deterministic automaton of a non deterministic one, yet the resulting automaton may be of exponential size with respect to the input.

6.1.1 Handling Ranked Trees

For XML and any ordered Semistructured data model, using regular tree automata for ranked trees is not possible without modification, as the data models are indeed unranked. A common way to handle unranked trees with tree automata is to map the unranked trees to ranked counterparts. A way to achieve this, is to lift tree nodes to a view, where nodes are represented e.g. by a $node_{/3}$ item with the label as one child[4] of $node_{/3}$, the first child of the unranked tree as second child of $node_{/3}$ and the following sibling—if present—of the current node in the context of its parent node as third child of $node_{/3}$. As $node_{/3}$ is always of arity 3, it is necessary to provide an additional node type denoting the end of a branch, e.g. the end of a list of siblings or an empty child list—this node will be called $eob_{/0}$ for *end of branch*.

Example 6.3
The unranked tree $f[a, b[d], c]$ is mapped to the ranked counterpart

$$node(f, node(a, eob, node(b, node(d, eob, eob), node(c, eob, eob))), eob)$$

A corresponding automaton has to recognize the ranked transcription of unranked trees the regular way.

[3] For an equivalence proof see [22], page 20.
[4] A label l is mapped to a node $l_{/0}$ in the ranked mapping.

Example 6.4
An automaton accepting the former example could for example be:

$$
\begin{array}{rl}
A = & \{ \quad \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}\} \\
& , \quad \{node_{/3}, a_{/0}, b_{/0}, c_{/0}, eob_{/0}\} \\
& , \quad \{q_{11}\} \\
& , \quad \{ \ node(q_3(X), q_6(Y), q_6(Z)) \rightarrow q_7(node(X,Y,Z)) \\
& \quad , \ a \rightarrow q_1(a) \\
& \quad , \ b \rightarrow q_2(b) \\
& \quad , \ c \rightarrow q_3(c) \\
& \quad , \ d \rightarrow q_4(d) \\
& \quad , \ f \rightarrow q_5(f) \\
& \quad , \ eob \rightarrow q_6(eob) \\
& \quad , \ node(q_4(X), q_6(Y), q_6(Z)) \rightarrow q_8(node(X,Y,Z)) \\
& \quad , \ node(q_2(X), q_8(Y), q_7(Z)) \rightarrow q_9(node(X,Y,Z)) \\
& \quad , \ node(q_1(X), q_6(Y), q_9(Z)) \rightarrow q_{10}(node(X,Y,Z)) \\
& \quad , \ node(q_5(X), q_{10}(Y), q_6(Z)) \rightarrow q_{11}(node(X,Y,Z)) \ \} \\
& \}
\end{array}
$$

6.2 An Automaton Model for Unranked Regular Rooted Graph Languages

In this section an automaton model for R_2G_2 is the introduced. As R_2G_2 models languages of unranked trees, handling of unranked ordered trees is essential for the automaton model sought of.

As the class of tree grammars in use can be captured solely using automata operating on unranked tree transcriptions of ranked ones, it is useful to introduce an automaton model solely coping with such kind of languages. A new hyper graph based formalism is introduced. This formalism has proved useful for didactic purpose along this deliverable as well as easy to implement. All methods involving data handling (e.g. validation or typing) with automata are formulated directly on the unranked data formalism. For this reason, the automata are considered to be automata for unranked tree, opposed to automata for ranked trees as presented in [22].

6.2.1 Labelled Directed Hyper Graphs as Non-Deterministic Regular Tree Automata

In the spirit of non deterministic tree automata (for ranked trees) as briefly introduced in section 6.1, a hyper graph based automaton approach is presented now. The main difference is the treatment of unranked trees or graphs in the style of XML abstract syntax trees over ranked trees in the style of classical logical terms with symbols of fixed arity. The main difference in formalisation is the use of (hyper) graph edges as transitions instead of term transitions as presented in section 6.1 and widely used in [22].

Definition 6.2 (Labelled Directed Hyper Graphs as Non-Det. Regular Tree Automata)
A non-deterministic regular tree automaton M is a 5-tuple (Q,Δ,F,R,Σ) with label alphabet Σ, states Q, final states F where $F \subseteq Q$, transitions Δ where $\Delta \subseteq (Q\times\Sigma\times Q\times Q) \cup (Q\times Q)$ (regular transitions are of the domain $Q\times\Sigma\times Q\times Q$ and ε-transitions are of the domain $Q\times Q$) and a set of root transitions R with $R \subseteq \Delta$.[5] A transition $(s,e) \in \Delta$ will be called an "ε-transition" from now on, where s is called "start state" of the transition and e is called its "end state". A transition $(s,l,c,e) \in \Delta$ is a non-ε-transition, where s is called "start state", e is called "end state", l is called the "label" of the transition and c is called the "content start state". The transition may be traversed partly, or in one *dimension*, where a traversal along the component from s to e is called "horizontal transition step" and from s to c "depth transition step".

The hyper graph automata introduce ε-edges as commonly used for (string, not tree) finite automata, as e.g. presented in [44] and [26]. They do not raise the expressiveness of the automaton model, they are just more convenient for automata construction based on regular expressions. Algorithms for ε-edge removal exist, as shown in [44] and [26].

Definition 6.3 (Projection of hyper graph components)
For an automaton $A = (Q,\Delta,F,R,\Sigma)$ projection of the components is defined as $Q_A = Q$, $\Delta_A = \Delta$, $F_A = F$, $R_A = R$ and $\Sigma_A = \Sigma$. The union of two automata A_1 and A_2 is defined as the pairwise union of its components, e.g. $A_1 \cup A_2 = (Q_{A_1} \cup Q_{A_2}, \Delta_{A_1} \cup \Delta_{A_2}, F_{A_1} \cup F_{A_2}, R_{A_1} \cup R_{A_2}, \Sigma_{A_1} \cup \Sigma_{A_2})$. The difference of two automata is defined in a similar way, yet consistency of all transitions must be retained, e.g. all states and symbols involved in transitions are defined in Q, respectively Σ. Useful functions for the construction of automata are the addition and subtraction of transitions to (or from) an automaton defined as follows: $A_1 + \tau = A_2$ such that $\tau = (s,l,c,e)$ and $A_1 \cup (\{s,c,e\},\{\tau\},\{\},\{\},\{l\}) = A_2$, $A_1 - \tau = A_2$ such that $\tau = (s,l,c,e)$ and $A_1 \setminus (\{s,c,e\},\{\tau\},\{\},\{\},\{l\}) = A_2$.

For the sake of concreteness an example automaton for the following grammar G is presented:
The language generated by the grammar

$$\begin{array}{lcl} \texttt{element } A & \rightarrow & a[\ (B,B)^{+}\ ]; \\ \texttt{element } B & \rightarrow & b[\ A^{*}\ ]; \end{array}$$

is accepted by the following automaton

Comparing Hyper Graph Automata and Ranked Tree Automata Applied to Unranked Data Transcription The hyper graph automaton model is a special notation for fixed arity tree automata where fixed arity nodes of arity 3 are matched—one position represents the label of the matched unranked node, one for the content list of this node and one for the following sibling node in the content list in which this node is contained. This is reflected in the hyper edges of arity 4 relating (1) the start state of the transition, (2) the label, (3) the start state of the child list and (4) the start state of the list of following siblings. The advantage of this approach is the arguably less bulky notation, as the node abstraction is not explicit. The disadvantage is the need to redefine many operations already available for ranked tree automata to hyper graph automata. From a practical point of view the hyper graphs are arguably well suited as automata models for type checking on Xcerpt.

[5] Usually we need just one root transition, but for technical reasons it is convenient to have a set of root transitions.

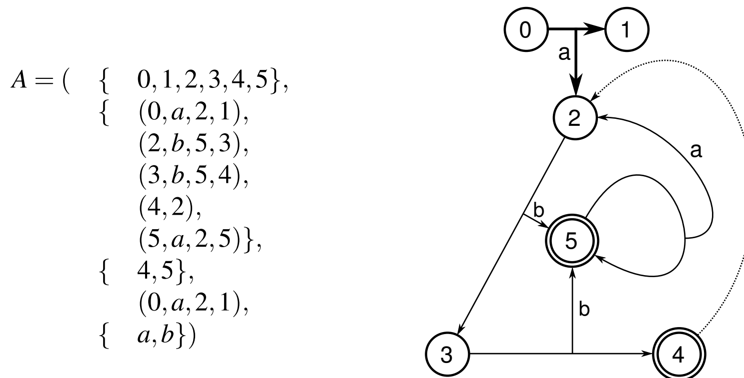


Figure 6.1: An example hypergraph automaton in a typical textual representation on the left and with graphical representation on the right.

Deterministic vs. Non-Deterministic Automata It is possible to restrict tree automata to *deterministic* tree automata—deterministic tree automata have always exactly one *matching* transition from a given left hand side to a new state along a given node label, while non-deterministic automata may have more than one matching transition. The decision procedure for membership test is simpler using deterministic automata, as all possible derivations (e.g. paths of a decision tree) lead to a successfully closed branch. If no derivation rule is applicable any more, the data tree is not member of the language represented by the deterministic automaton. With non-deterministic automata, it is still possible, that earlier in the derivation tree another decision (e.g. another choice of a transition) leads to a successfully closed branch. A deterministic algorithm checking membership using non-deterministic automata has therefore to retract choices in dead ends of the derivation tree, if further derivations are possible and membership has not been proved at that moment. The property of not having to retract derivation choices is called *confluence*. The decision procedure for membership test on deterministic automata is a confluent system.

While deterministic automata are favorable for membership testing, their creation from regular expressions may be of exponential complexity. Generation of non-deterministic finite automata based on regular expression as language specification can be done in polynomial time.[6] As R_2G_2 uses regular expressions to specify content models, the translation of R_2G_2 to deterministic finite automata may be of exponential complexity.

Fortunately, all necessary operations for type checking, e.g. intersection, emptiness test, subset test, can be implemented in polynomial complexity directly on non-deterministic automata. This will be shown along this chapter when introduced.

6.2.2 Membership Test for a Tree using Hyper Graph Automata

An algorithm able to test membership of unranked trees in a language represented by a hyper graph based automaton is presented. In contrast to the standard approach for ranked tree as shown in [22] and introduced earlier, this algorithm is able to operate directly on the unranked tree model without prior transcription of data instances to a ranked tree representation. Calculus rules are used to explain the algorithm in a non-deterministic way. Rules are of the following shape:

[6] With respect to the size of the regular expression.

$$\frac{\begin{array}{c} C_1 \\ \vdots \\ C_n \\ e_1 : a_1 \qquad e_n : a_n \end{array}}{e : a} \quad \text{(EXAMPLE)}$$

C_i denote constraints on e, a, e_i, a_i and e, e_i are trees or content lists of trees, i.e. sequences of trees that all share the same parent node. By a, a_i either states or transitions of an automaton are denoted. An expression $e : a$ will also be called a configuration of the automaton. The rules relate configurations of automata. Two different kinds of configuration exist: (1) configurations of shape $t : \tau$ where t is a tree and τ is a transition, (2) or $[t_1, \ldots, t_n] : S$ where $[t_1, \ldots, t_n]$ is a list of trees and S is a state of the automaton.

$$\frac{\begin{array}{c} \tau \in R_A \\ (s,l,c,e) = \tau \\ se \in F_A \\ t : \tau \end{array}}{} \quad \text{(ROOT)}$$

The ROOT rule matches the root of the data tree, if there is a transition in the set of root transitions from which on a whole derivation tree can be found.

$$\frac{c \in F_A}{[\,] : c} \quad \text{(END)}$$

The END rule accepts an empty list, if the configuration involves an empty list and a state of the set of final states.

$$\frac{\begin{array}{c} \tau \in \Delta_A \\ \tau = (s,l,c,e) \\ [t_1, \ldots, t_n] : c \end{array}}{l[t_1, \ldots, t_n] : \tau} \quad \text{(NODE)}$$

$$\frac{\begin{array}{c} n \geq 1 \\ \tau \in \Delta_A \\ \tau = (s,l,c,e) \\ t_1 : \tau \qquad [t_2, \cdots, t_n] : e \end{array}}{[t_1, t_2, \ldots, t_n] : s} \quad \text{(LIST)}$$

In a successful derivation, applications of the NODE and the LIST rule are interwoven and all branches end with an application of the END rule while the root of the derivation tree is an application of the ROOT rule. A tree without possible derivation is not valid with respect to the given automaton, multiple derivations may exist.

Example of a Tree Recognition using a Hyper Graph Automaton Given the tree $a[b[],b[a[b[],b[]]]]$ and the automaton A as presented in the former example (example 6.1, the following derivation is a possible recognition:

Operational Semantics of The Recognition Rules The rules presented above give an abstract description of a recognition algorithm. Neither the control flow nor decision in case of ambiguity are captured by the rules. It is possible, that an automaton can recognize a data tree using different derivations, a concrete algorithm should be designed to either choose one of those derivations, maybe driven by other parameters.

Exemplary, a simple algorithm choosing one rule is sketched now:

Algorithm 6.2.1: MEMBERSHIPTEST($A = (Q, \Sigma, \Delta, S, F), e : a, \rho$)

comment: ρ denotes the set of rules

comment: e denotes either a tree or a list of trees

comment: either $a \in Q$ or $a \in \Delta$

comment: To check a tree t, call MEMBERSHIPTEST(A,t:s,R) with $s \in S$

for $\dfrac{e_1 : a_1 \quad e_n : a_n}{e_\rho : a_\rho} \begin{matrix} C_1 \\ \vdots \\ C_m \end{matrix} \in R$

do {
  if $C_1 \wedge \cdots \wedge C_m = true$
  then {
    if MEMBERSHIPTEST$(A, e_1 : a_1, \rho) = true$
    $\wedge \cdots \wedge$
    MEMBERSHIPTEST$(A, e_n : a_n, \rho) = true$
    then return $(true)$

return $(false)$

An Upper Bound Complexity for Membership Test of Tree Shaped Data Various ways of document validation with automata have been proposed in *"Tree Automata Techniques and Applications"* [22]. Easily adaptable to the presented approach is the non-deterministic bottom-up approach. The upper complexity is polynomial in the number of nodes and the number of states. Applied to the former algorithm this can be explained as follows:

1. On each node, there may be at most all rules to apply, each with all states or edges (depending on the rule—some apply to states other to edges) to be checked. The number of rules is constant (there are 4 rules), the number of edges is linear in the size of the grammar, as shown in section 6.3.

2. A naive, top down, approach could easily result in a combinatorial search, yielding exponential complexity. A bottom-up approach is an easy way to overcome that:

   (a) Each child node is *reached up* in a recursive application in all possible typing. Note, that *reaching up* in the recursive application corresponds to reaching

      - the typed sequence of nodes to it's parent node in applications of the NODE typing rule,
      - the typed sequence of following siblings to it's direct preceding sibling in the applications of the LIST typing rule.

(b) Only the typed contributions, that can contribute to successful typing in a given recursion are kept, at the same time, if they occur in different successful typing in the given recursion, they can be reused without recalculating them—their validity is independent on their context.

3. When checking a node and a type (state or transition), this type may have to be checked against all types returned by the recursive calls (at most two of them exist—one for the following sibling and one for the content model). As a typed node may at most have as many types as edges exist, this step is quadratic in the number of edges.

4. Hence, as consequence of (1), each node at most has 'number of edges' types, and as consequence of (2), it is not necessary to calculate the possible types of a node more than once. As consequence of (3) each node takes at most quadratic time in the computation—this gives us a **polynomial complexity in the order of $O(N \times M^2)$ where N is the size of the tree (or the number of nodes) and M is the size of the automaton (or the number of edges as an upper bound)**.

6.2.3 Recognition of a Rooted Graph using Hyper Graph Automata

The recognition of rooted graphs is defined in analogy to the recognition of trees—a rooted graph is recognized by an automaton, if it is in the language accepted by this automaton. There is a certain correlation of the acceptance of a word by an automaton with the recognition of equality of two words: for each word, it is easy to obtain an automaton such that exactly this word is accepted. Therefore, this automaton provides a way to decide about equality of two words. The most precise way to judge about equality of two graphs is graph isomorphism. The decision procedure for graph isomorphism has exponential complexity. Assuming, that we base a recognition procedure for graphs on graph isomorphism—a graph is accepted, if it is isomorphic to a graph in the language accepted by the automaton—then the while process has to have exponential complexity, as otherwise the graph isomorphism itself would have sub exponential complexity (e.g. it could be reformulated by means of graph recognition)

A weaker kind of membership relation between graphs will be chosen: the simulation relation—a graph is accepted by an automaton, if there is an instance in the language accepted by the automaton that simulates the graph.

> A simulation preorder is a relation between [graphs] associating systems which behave in the same way in the sense that one system simulates the other. Intuitively, a system simulates another system if it can match all of its moves.[7]

A first advantage of using the simulation preorder as base of the membership test in the recognition procedure is, that the decision procedure can be achieved in polynomial time. Second advantage, Xcerpt is based on a non standard unification called *simulation unification*, which itself is based on simulation preorder. In [43] (especially see section 4.4 *"Query Evaluation: Ground Query Term Simulation"* in [43]) the simulation preorder on so called ground query terms—of which data trees and graphs are a subset—is presented including also complexity results. Ground Xcerpt query term simulation has been shown to be a useful relation between trees or graphs. Arguably simulation preorder reflects well a notion of *"expected result"* for many applications of querying Web and Semantic Web data.

(Possibly Infinite) Tree Representations of Graphs As the recognition procedure is defined for trees so far, it is arguably useful to base the handling of graph shaped data on tree recognition. Note, that trees are a special kind of graphs, so trivially those graphs are already handled. For directed, acyclic graphs it

[7] From `http://en.wikipedia.org/wiki/Simulation_preorder`.

is always possible to find a spanning, finite tree, where nodes accessible from one node chosen as a root using different paths are duplicated in the tree representation. In general, a graph can be spanned by different trees, capturing different possible graph traversals. As the root in a rooted graph is fixed, there is only one possible such spanning tree. Cyclic graphs can conceptually be represented using infinite trees where infinite always means finite in breadth (a node in a finite graph can only have finite many successors, so can the corresponding node in the tree) and branches of infinite depth for cycles.

By applying the tree approach for recognition on acyclic graphs, an algorithm is achieved that possibly checks the same nodes multiple times, but that always terminates. It is possible, that the same node is checked multiple times using different or the same automata transitions. Arguably it is reasonable to *remember* acceptance results of nodes with corresponding transitions, as not only the testing of validity of a certain node in a context can be omitted, but also the testing of all child nodes can be skipped. The process of *remembering* earlier calculations in this state is called memoization.

By applying the tree approach to cyclic graphs, the recognition process gets stuck in non termination. However, the explained extension of memoization guarantees termination. This is due to the fact, that any data tree node can in worst case only be tested against finite many transitions of the automaton, as the automata are finite.

An Algorithm for the Recognition of Rooted Graphs The former algorithm for tree recognition is now extended by memoization to recognize rooted directed graphs. A graph is recognized by an automaton A, if it is simulated by a graph in the language $\mathcal{L}(A)$. The set of rules will not be affected by this change, but the algorithm for the application of the rules on a given data tree. This emphasizes the declarative nature of the rules and the fact, that conceptually trees and graphs are handled in a similar way:

Algorithm 6.2.2: MEMBERSHIPTEST($A = (Q, \Sigma, \Delta, S, F), e : a, \rho$)

global *memo*
comment: ρ denotes the set of rules

comment: e denotes either a graph node or an adjacency list of graph nodes

comment: either $a \in Q$ or $a \in \Delta$

comment: To check a graph t call MEMBERSHIPTEST(A,t:s,R); $s \in S$ and $memo = \{\}$

if $e : a \mapsto b \in memo$
then return (b)
else
- $memo \leftarrow memo \cup \{e : a \mapsto true\}$
- **for** $\dfrac{e_1 : a_1 \quad e_n : a_n}{e_\rho : a_\rho}$ with conditions $C_1, \ldots, C_m$ $\in \rho$
  - **if** $C_1 \wedge \cdots \wedge C_m = true$
  - **then**
    - **if** MEMBERSHIPTEST$(A, e_1 : a_1, \rho) = true$ $\wedge \cdots \wedge$ MEMBERSHIPTEST$(A, e_n : a_n, \rho) = true$
    - **then return** $(true)$

$memo \leftarrow memo \setminus \{e : a \mapsto true\}$
$memo \leftarrow memo \cup \{e : a \mapsto false\}$
return $(false)$

An Upper Bound Complexity for Membership Test of Graph Shaped Data So, the membership test for graph shaped data presented here is an extension of the membership test for tree shaped data—the result of the validation of nodes is memorized and used to end validation of cyclic structures. The complexity of the membership test of a node in a graph shaped document is hence the same as the complexity of a node validation for tree shaped data (see the end of section 6.2) plus the costs for memorizing the each nodes validation result. If the data structure used to represent nodes has the ability to store meta data of the validation process, the overhead is constant, hence complexity of graph validation based on simulation has the same complexity as validation of tree shaped data. If, as in the pseudo code example above, a look-up table is used for memorization, the factor for updating and reading the look-up table has to be added to the cost of the validation of a node. However, a hash table can provide this with logarithmic (over the number of nodes in the data graph) overhead, hence the cost of graph validation with look-up table based memorization is the same as the costs for tree shaped data (section 6.2)—polynomial in the size of the data graph.

6.3 A Calculus Relating Automata and R_2G_2

Yet the presented automaton model is well suited as execution model for recognition of trees and graphs on regular rooted graph languages, it is not convenient as language definition formalism for the end user of applications of such problems—e.g. for document schema authors and programmers. XML Schema, DTD, Relax NG and R_2G_2 are arguably appropriate formalisms for this task.

The following set of rules describe an algorithm for the generation of an automaton, as formerly introduced, for a given R_2G_2 instance. The rules are strictly defined along the structure of the R_2G_2 syntax, hence an implementation could be a recursive descend function along the abstract syntax tree of an R_2G_2 instance. Adapting the algorithm to Relax NG or DTD as input languages is not difficult—the rules have to be adapted to the abstract syntax components of those languages. Adapting the algorithm to XML Schema as input language requires some additional processing doe to object oriented modelling features, but application to the tree grammar based core of XML Schema is comparable to the translation of R_2G_2.

The rules have a local aspect, in that they are driven by the abstract syntax tree and there is no context sensitive property affecting the applicability of the rules. Additionally to the local behaviour of the rules with respect to the abstract syntax tree, two global environments are altered and queried by the algorithm: the automaton, constructed while processing the abstract syntax tree, accepting exactly the language generated by the grammar at the end and a relation relating type names of the grammar to labels and start states of the automata parts implementing the content model of the given type name. In implementation of the rules, it is possible to split the context sensitive processing (look up of automaton components for type name definitions) and the context free processing (i.e. the recursive descend along the abstract syntax tree of the R_2G_2 definitions) in different phases.

The rules have the following general structure:

$$\frac{\begin{array}{ccc} & \Delta_A & \\ & \Delta_L & \\ t_1 \mapsto (i_1, o_1) & \cdots & t_n \mapsto (i_n, o_n) \end{array}}{T(t_1, \ldots, t_n) \mapsto (i, o)} \quad \text{(EXAMPLE)}$$

By $T(t_1, \ldots, t_n)$ a term of the structure of the R_2G_2 syntax definition is denoted, t_i are its sub-terms. The rules hence relate the necessary automata construction operations with the automata construction

operations of its sub-terms. Rule applications are functional, mapping to a tuple of states, denoted by $(i,o),(i_i,o_i)$. Some states in the tuples above the line may occur in the tuple below the line, but the tuples above the lines, i.e. the tuples of the applications of the sub-terms t_i are independent from each other. In the situations, where the states are irrelevant, the tuple ε is used. An implementation may return any tuple of states or nothing here (i.e. `null` values).

By Δ_A, a condition on the automaton A is denoted. This may imply editing the automaton in an implementation. The conditions are always of the form $A' \subseteq A$—there are always positive expressions about components of the automaton, never negative ones, hence a concrete implementation of the algorithm in an imperative programming language can implement the condition as an addition of the components of A' to the components of A.

By Δ_L querying or altering the global environment L is denoted. $L \subseteq N \times \Sigma \times S$ is a ternary relation between non terminals as found in the R_2G_2 instance, symbols or term labels, and automata states.

As type names on the right hand side of an R_2G_2 rule may be used, even if their declaration (i.e. the rule with that type name on the left hand side) is still pending, more than one pass over the abstract syntax tree of the R_2G_2 instance is necessary in an implementation of the algorithm. The rule based abstraction of the algorithm neglects the necessity of multiple passes.

$$\frac{\begin{array}{c}(\{\},\{\},\{(s,l,c,e)\},\{e\},\{\}) \subseteq A \\ L(N_r) = (?,l_r,c_r,?) \\ \rho_j \;=\; \texttt{element}\, N_j \;\rightarrow l[\cdots] \\ A \;|\; \rho_1 \;\mapsto\; (i_1,o_1) \qquad \cdots \qquad A \;|\; \rho_n \;\mapsto\; (i_n,o_n)\end{array}}{A \;|\; \rho_1,\cdots,\rho_n,\texttt{root} = N_r \;\mapsto\; (?,?)} \qquad (\text{GRAMMAR})$$

The automata construction rule for grammars describe the relationship between the results of automata construction rule applications for all R_2G_2 rules in a given grammar. The result state tuple of application of this rule has no meaning and is hence undefined. An implementation may safely return e.g. null-values or any dummy tuple here. The result state tuples of the automata construction algorithm applied to the rules (which neither have use or meaning, as it will be shown in automata construction rule for R_2G_2 rules) are not considered for any construction and can hence be ignored. Apart of recursively applying automata construction to all rules, the root declarations are treated in the grammar case.

$$\frac{\begin{array}{c}(\{s,e\},\{l\},\{(s,l,s_{re},e)\},\{\},\{e_{re}\}) \subseteq A \\ L(N) = (l,s_{re}) \\ A \;|\; re \;\mapsto\; (s_{re},e_{re})\end{array}}{A \;|\; \texttt{element}\, N \rightarrow l[\, re \,] \;\mapsto\; (s_?,e_?)} \qquad (\text{RULE})$$

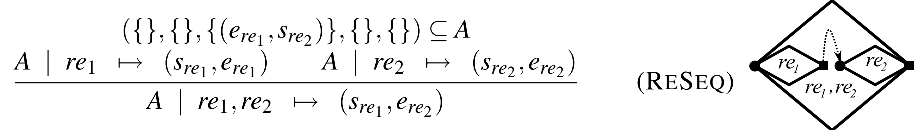
An application of the automata construction rule to an R_2G_2 rule results intentionally in a meaningless state tuple, which is no problem, as the tuple is not used—recall, that these rule application may only occur in the context of a grammar rule application ignoring the result state tuple nevertheless. The look-up table or global environment L relates the given type name on the left hand side of the R_2G_2 rule to the label of the type term on the right hand side and to the start state of the automaton part realizing the content model of the right hand side type term. For an application of the automaton construction rule to an R_2G_2 rule the automaton contains an edge (s,l,s_{re},e), two states not introduced elsewhere, s and e and the state e_{re} can be declared as final state.

$$\frac{\begin{array}{c}(\{s,e\},\{l\},\{(s,l,s_{re},e)\},\{\},\{e_{re}\}) \subseteq A \\ A \;|\; re \;\mapsto\; (s_{re},e_{re})\end{array}}{A \;|\; l[\, re \,] \;\mapsto\; (s,e)} \qquad (\text{TYPETERM})$$

The type term automata construction rule is similar to the previously introduced “Rule” rule, except that the look-up table is not involved, as a type term by itself is not associated to a non terminal or type name. Recall, that type terms may occur at the right hand side of a rule, as well as as child terms of type terms—the first case is caught by applications if the “Rule” rule, the second case by applications of the “Type term” rule. The result tuple consists of two new states.

$$\frac{(\{s,e\},\{\},\{(s,l,c,e)\},\{\},\{\}) \subseteq A \qquad L(N) = (l,c)}{A \mid N \;\mapsto\; (s,e)} \qquad (\textsc{Typename})$$

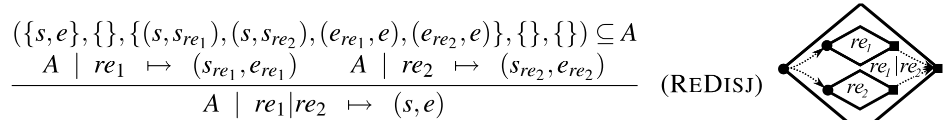
The “Typename” rule is applicable to type names occurring in type terms. For each such type name, the label and content model start state introduced by the corresponding application of the “Rule” rule is being used to construct an edge from a new state s to a new state e. The new states form the result tuple of the rule application.



$$\frac{(\{\},\{\},\{(e_{re_1},s_{re_2})\},\{\},\{\}) \subseteq A \qquad A \mid re_1 \;\mapsto\; (s_{re_1},e_{re_1}) \qquad A \mid re_2 \;\mapsto\; (s_{re_2},e_{re_2})}{A \mid re_1,re_2 \;\mapsto\; (s_{re_1},e_{re_2})} \qquad (\textsc{ReSeq})$$

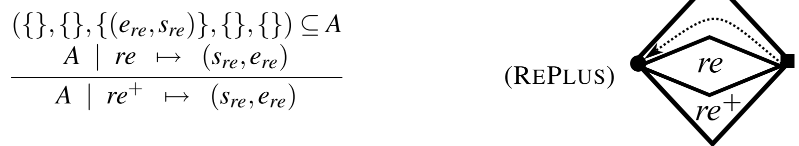
The automaton construction for a sequence of two regular expressions (recursively of arbitrary many expressions) is being handled by a “ReSeq” rule application. The result tuples of the automaton construction rules applications to the two regular expressions to be sequenced are used to (1) return a result tuple consisting of the first state of the first expression’s tuple and the last state of the second expression’s tuple, and (2) to tie the two automata parts of the two expressions together using an ε-edge.

All the automata construction rules for regular expressions are similar—the result of automata construction of sub-expressions is being connected in some way using ε-edges, sometimes not involving new states. The explanation of the rules will be presented in a graphical way from now on—in this visualisation, the automaton part being constructed is represented as a diamond shaped polygon, sub automata as well and contained inside of the bounds of the automaton. The first state of the tuple is being depicted as a black circle, the second as a little black square. ε-edges are depicted by dotted arrows. The text in the graphics has mere documentary character.



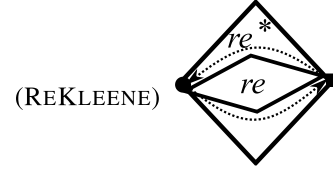
$$\frac{(\{s,e\},\{\},\{(s,s_{re_1}),(s,s_{re_2}),(e_{re_1},e),(e_{re_2},e)\},\{\},\{\}) \subseteq A \qquad A \mid re_1 \;\mapsto\; (s_{re_1},e_{re_1}) \qquad A \mid re_2 \;\mapsto\; (s_{re_2},e_{re_2})}{A \mid re_1|re_2 \;\mapsto\; (s,e)} \qquad (\textsc{ReDisj})$$

$$\frac{(\{\},\{\},\{(s_{re},e_{re})\},\{\},\{\}) \subseteq A \qquad A \mid re \;\mapsto\; (s_{re},e_{re})}{A \mid re^{?} \;\mapsto\; (s_{re},e_{re})} \qquad (\textsc{ReOpt})$$



$$\frac{(\{\},\{\},\{(e_{re},s_{re})\},\{\},\{\}) \subseteq A \qquad A \mid re \;\mapsto\; (s_{re},e_{re})}{A \mid re^{+} \;\mapsto\; (s_{re},e_{re})} \qquad (\textsc{RePlus})$$

$$\frac{(\{\},\{\},\{(s_{re},e_{re}),(e_{re},s_{re})\},\{\},\{\}) \subseteq A \quad A \mid re \mapsto (s_{re},e_{re})}{A \mid re^* \mapsto (s_{re},e_{re})}$$ (ReKleene)



An Upper Bound Complexity for The Non-Deterministic Automaton Generated out of an R_2G_2 Instance
The cost of constructing an automaton out of a grammar is polynomial in the size of the grammar, the size of the resulting automaton, given in the number of edges, is in the order of $O(N)$ for N as the size of the grammar (i.e. abstract syntax items of the parsed grammar), the time complexity is $O(N)$ as well. To reason about the complexity, it is good to evaluate the cost of each automaton generation rule. The total cost is then the sum of all generation rule applications, as each rule application consumes a part of the input grammar, no backtracking is needed (as the choice of the applicable rule is unambiguous).

ReDisj A disjunction of two regular expressions adds the cost of four ε-edges to the cost of the translation of the two regular expressions. It can be assumed, that construction of the edge has constant complexity.

ReOpt An optional regular expression adds the cost of one ε-edges to the cost of the translation of the regular expression.

RePlus A plus-adorned regular expression adds the cost of one ε-edges to the cost of the translation of the regular expression.

ReKleene A Kleene-star-adorned regular expression adds the cost of one ε-edges to the cost of the translation of the regular expression.

ReSeq Sequencing two regular expressions adds no cost to the cost of the two regular expressions.

Typename Translating a type name to an automaton component costs as much as the construction of an ε-edge. Looking up the target of the edge in the look-up table can be assumed to be able in logarithmic time with respect to the number of type names in an appropriate data structure, like e.g. a hash table.

Typeterm Translating a type term corresponds to the construction of an edge. It can be assumed, that construction of the edge has constant complexity, which adds up to the translation of the regular expression for the content model.

Rule A rule constructs no new content, but has to alter the look-up table—again, in an appropriate data structure, this can be assumed to be able in logarithmic time with respect to the number of type names, which adds up to the translation of the right hand side type term.

Grammar Translating a grammar into an automaton costs as much as adding up the costs of the translation of all rules.

6.4 Some Set Theoretic Computations on Hyper Graph Automata

For the purpose of static type checking, it is necessary to analyse some set properties on languages, namely (1) emptiness of a language, (2) the subset property between two languages and to (3) calculate

the intersection of two languages.

6.4.1 The Emptiness Test

The emptiness test finds out, if for an automaton, there may be any data instance accepted by this automaton, if therefore the language accepted by the automaton is non empty. For an automaton to accept finite trees, it is obviously necessary to find paths along the hyper edges ending in final states. For infinite trees or graphs containing loops, this property can be relaxed, as such data instances can be accepted by loops without final state in the automaton. Automata constructed from R_2G_2 definitions arguably always accept non empty languages for three reasons: (1) As they have a root transition by definition (based on the mandatory `root` declaration). (2) Along the breadth axis of the automata there is always either an end state at the end of each path or the path is a loop containing an end state. This is due to the fact, that the last state constructed by regular expression decomposition is always an end state, or a looping ε-edge is added to an end state terminated path to express repetition. (3) Along the depth axis there is either an end state due to empty content, or a transition to a state representing another grammar rule. This state again is part of a non empty breadth axis and either of a final state terminated depth axis or of a depth axis recursively fulfilling reason 3. A depth axis loop without final state can therefore only accept infinite trees or graphs containing an appropriate loop.

Nevertheless, automata representing only empty languages exist in practise: intersection of two automata can lead to an automaton accepting only empty languages, e.g. by construction of an automaton without start transition or by construction of an automaton with root transitions with all outgoing edges ending only in branches without loops and final states. Detection of automata representing empty languages is important for type checking.

An algorithm for detection of emptiness for a given automaton is sketched now:

Algorithm 6.4.1: ISEMPTY($A = (Q, \Sigma, \Delta, S, F)$)

```
memoisation ←create a lookup table of truth values with index over Q
comment: memoisation is defined in each call of ISEMPTY().

procedure RECURSIVETRANSITIONTEST(δ = (s,l,c,e))
  return (DEPTHTEST(c) ∧ BREADTHTEST(e))

procedure RECURSIVETRANSITIONTEST(δ = (s,e))
  return (BREADTHTEST(e))comment: handling of ε-transitions.


procedure BREADTHTEST(v)
  if v ∈ memoisation
    then return (memoisation[v])
  memoisation[v] ← true
  for (v,l,c,e) ∈ Δ
       ⎧if RECURSIVETRANSITIONTEST((v,l,c,e)) = false
    do ⎨       ⎧memoisation[v] ← false
       ⎪  then ⎨return (false)
       ⎩       ⎩exit
  return (true)

procedure DEPTHTEST(v)
  if v ∈ memoisation
    then return (memoisation[v])
  memoisation[v] ← false
  for (v,l,c,e) ∈ Δ
       ⎧if RECURSIVETRANSITIONTEST((v,l,c,e)) = false
    do ⎨       ⎧memoisation[v] ← false
       ⎪  then ⎨return (false)
       ⎩       ⎩exit
  memoisation[v] ← true
  return (true)

main
  for δ ∈ S
       ⎧if RECURSIVETRANSITIONTEST(δ) = false
    do ⎨  then ⎧return (false)
       ⎩       ⎩exit
  return (true)
```

An Upper Bound Complexity for the Emptiness Test The complexity of the emptiness test is linear in the size of the automaton. This is given by the fact, that the graph structure of the automaton is traversed to check if the paths in depth and breadth are closed. When graph traversal comes to a state already visited, the memoization stops the recursion at the (apparently cyclic) branch and returns the value of the last computation at this state. Hence, no edge is traversed twice. The Cost for the look-up of the memoized states is logarithmic in the size of the automaton, but if the states are extended

with some mean of annotation of the traversal, the look-up can be replaced by a constant check of the annotation.

6.4.2 Intersection of Regular Rooted Graph Automata

Calculating the intersection of two regular languages is a common exercise in text books about theoretical computer science and automata theory and it is also of high practical use. Given e.g. two language definitions for two versions of a data format, the intersection reflects a kind of conservative transitional data format providing guaranteed backward and forward compatibility. In type checking of the Xcerpt query language, non empty intersection can play an important role for checking selection constructs: given a query with multiple occurrences of the same variable, the occurrences may have different types. If the types have empty intersection, no data exists conforming the type constraint of the variables, therefore the selection may never select any valid data with respect to the types and is therefore arguably useless. Note, that different type annotations may either occur due to a query programmers annotation or due to type inference. Checking consistency of such concurrent type annotations in Xcerpt query terms can also be handled using an emptiness test for the type intersection.

Intersection of Regular (String) Languages Using DFAs The presented approach is a classical text book approach as found in [26] and [44]. It serves as introduction to a technique of calculating intersection and will be modified to non-deterministic and then to regular graph automata.

It is easily possible to construct the intersection of L_1 and L_2, if union and complement are defined, as generally $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ holds. A direct construction is presented, as neither union, nor complement is presented by now, and is not strictly necessary for type checking of Xcerpt later on. A direct construction is achieved by simulating parallel execution of the two deterministic finite automata representing L_1 and L_2. This corresponds to the construction of the product automaton:

Let deterministic automata be defined as 5-tuples $(Q, \Sigma, \Delta, s, F)$ with Q as the states of the automaton, Σ as the alphabet of the corresponding language, s the start state and $F \subseteq Q$ as the final states. The transitions $\Delta \subsetneq Q \times \Sigma \times Q$ are defined such that for every $v \in Q$ and for every $l \in \Sigma$ there is a transition $(v, l, v') \in \Delta$ and no other transition $(v, l, v'') \in \Delta$ with $v', v'' \in Q$ and $v' \neq v''$.

For L_1 accepted by $A_1 = (Q_1, \Sigma_1, \Delta_1, s_1, F_1)$ and L_2 accepted by $A_2 = (Q_2, \Sigma_2, \Delta_2, s_2, F_2)$, the intersection $L_1 \cap L_2$ is accepted by $A_\cap = (Q_1 \times Q_2\ ,\ \Sigma_1 \cap \Sigma_2\ ,\ \Delta_\cap\ ,\ (s_1, s_2)\ ,\ F_1 \times F_2)$ where $\Delta_\cap((p, p'), a, (q, q')) = (\Delta_1(p, a, q), \Delta_2(p', a, q'))$.

See Figure 6.2 for an example on how to get a product of two automata.

An algorithm for the construction of an automaton $A_\cap = (Q_\cap, \Sigma_\cap, \Delta_\cap, s_\cap, F_\cap)$ from two automata $A_1 = (Q_1, \Sigma_1, \Delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma_2, \Delta_2, s_2, F_2)$ is presented now:

Algorithm 6.4.2: INTERSECTIONDFA(A_1, A_2)

$\Sigma_\cap \leftarrow \Sigma_1 \cap \Sigma_2$
$s_\cap \leftarrow (s_1, s_2)$
$Q_\cap \leftarrow Q_1 \times Q_2$
$F_\cap \leftarrow \{(v_1, v_2) \in Q_\cap \mid v_1 \in F_1 \wedge v_2 \in F_2\}$
for $(v_1, l, v'_1) \in \Delta_1$
  do $\begin{cases} \textbf{for } (v_2, l, v'_2) \in \Delta_2 \\ \quad \textbf{do } \{\Delta_\cap \leftarrow ((v_1, v_2), l, (v'_1, v'_2)) \end{cases}$

Figure 6.2: The product automaton on the right accepts the intersection of the language of the two automata on the left.

Extending the Approach to Non-Deterministic Finite Automata The presented approach has the drawback to require deterministic automata, that may have exponential size of a corresponding non-deterministic automaton. The automaton model focused on in this deliverable are usually non-deterministic ones. Fortunately, the approach can be extended to non-deterministic automata without exponential blowup in time or space.

The difference between deterministic and non-deterministic automata in a nutshell is (1) non-deterministic automata may have spontaneous state transitions along so called ε-edges without consumption of an input symbol, and (2) while each symbol in Σ has exactly one outgoing transition from each state in deterministic automata, any number of such edges may occur in the non-deterministic case.

Let non-deterministic automata be defined as 5-tuples (Q,Σ,Δ,s,F) with Q as the states of the automaton, Σ as the alphabet of the corresponding language, s the start state and $F \subseteq Q$ as the final states. The transitions are defined as $\Delta \subseteq ((Q \times \Sigma \times Q) \cup \Delta_\varepsilon)$ with $\Delta_\varepsilon \subseteq (Q \times Q)$.

To simulate the parallel execution of two automata in a product automaton with an epsilon transition (a,e) in one automaton, it is necessary to provide an epsilon edge for any product state (a,v) to the corresponding state (e,v). This reflects the possibility of a spontaneous transition every time the automaton with $a \in Q$ is in state a, independent of the state of the other automaton.

To handle the arbitrary amount of edges with one label from a state, no further change is necessary, as the deterministic algorithm already relates *all* edges of one automaton with all edges of the other one, as long as the transition labels match. In the deterministic case, by definition only one edge per state and label exists, therefore the same algorithm behaves as defined for the deterministic case.

An algorithm for the construction of an automaton $A_\cap = (Q_\cap,\Sigma_\cap,\Delta_\cap,s_\cap,F_\cap)$ from two non-deterministic automata $A_1 = (Q_1,\Sigma_1,\Delta_1,s_1,F_1)$ and $A_2 = (Q_2,\Sigma_2,\Delta_2,s_2,F_2)$ is presented now:

Algorithm 6.4.3: INTERSECTIONNFA(A_1,A_2)

$\Sigma_\cap \leftarrow \Sigma_1 \cap \Sigma_2$
$s_\cap \leftarrow (s_1, s_2)$
$Q_\cap \leftarrow Q_1 \times Q_2$
$F_\cap \leftarrow \{(v_1, v_2) \in Q_\cap \mid v_1 \in F_1 \wedge v_2 \in F_2\}$
for $(v_1, l, v_1') \in \Delta_1$
  do $\begin{cases} \textbf{for } (v_2, l, v_2') \in \Delta_2 \\ \quad \textbf{do } \Delta_\cap \leftarrow \Delta_\cap \{((v_1, v_2), l, (v_1', v_2'))\} \end{cases}$
for $(v_1, v_1') \in \Delta_1$
  do $\begin{cases} \textbf{for } v_2 \in Q_2 \\ \quad \textbf{do } \Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1', v_2))\} \end{cases}$
for $(v_2, v_2') \in \Delta_1$
  do $\begin{cases} \textbf{for } v_1 \in Q_1 \\ \quad \textbf{do } \Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1, v_2))\} \end{cases}$

Extending the Approach to Graph Automata The main difference of string- and graph automata is the shape of the transitions—triples for string automata and quadruples for graph automata. Fortunately, the calculation of an automaton accepting the language intersection of two automata, is easily derivable from the string automaton case. Informally, the only difference is the handling of the third state.

Algorithm 6.4.4: INTERSECTIONNDFTA(A_1,A_2)

$\Sigma_\cap \leftarrow \Sigma_1 \cap \Sigma_2$
let $(a_1, l_1, c_1, e_1) = s_1$
let $(a_2, l_2, c_2, e_2) = s_2$
$s_\cap \leftarrow ((a_1, a_2), l, (c_1, c_2), (e_1, e_2))$
$Q_\cap \leftarrow Q_1 \times Q_2$
$F_\cap \leftarrow \{(v_1, v_2) \in Q_\cap \mid v_1 \in F_1 \wedge v_2 \in F_2\}$
for $(v_{a_1}, l, v_{c_1}, v_{e_1}) \in \Delta_1$
  do $\begin{cases} \textbf{for } (v_{a_2}, l, v_{c_2}, v_{e_2}) \in \Delta_2 \\ \quad \textbf{do } \Delta_\cap \leftarrow \Delta_\cap \{((v_{a_1}, v_{a_2}), l, (v_{c_1}, v_{c_2}), (v_{e_1}, v_{e_2}))\} \end{cases}$
for $(v_1, v_1') \in \Delta_1$
  do $\begin{cases} \textbf{for } v_2 \in Q_2 \\ \quad \textbf{do } \Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1', v_2))\} \end{cases}$
for $(v_2, v_2') \in \Delta_1$
  do $\begin{cases} \textbf{for } v_1 \in Q_1 \\ \quad \textbf{do } \Delta_\cap \leftarrow \Delta_\cap \cup \{((v_1, v_2), (v_1, v_2))\} \end{cases}$

Figure 6.3 illustrates the cross product of two automata. The automata accept the languages defined by the grammar (for the upper left automaton)

```
root A;
element A = a[ A* ];
```

and the grammar (for the lower left automaton)

```
root A;
element A = a[ A|B ];
element B = b[ ];
```

The resulting automaton accepts the language represented for example by the grammar

```
root A;
element A = a[ A ];
```

The resulting intersection automaton contains some unreachable states and transitions, that could easily be removed using some minimization algorithm or simply by applying a reachability algorithm. As this is not essential to the tractability of type checking later on, automata minimization will not be considered.

An Upper Bound Complexity for Intersection of two NDFTA The complexity of calculating the intersection corresponds to calculating the cross product of the two automata A_1 and A_2. For $M = |A_1|$ and $N = |A_2|$ as the sizes of the automata (e.g. the number of edges), this gives a time and memory complexity in the order of $O(M \times N)$.

6.4.3 Automata Based Subset Test for two Regular Rooted Graph Languages

Given a regular language, testing if it is a subset of another regular language, is an important task e.g. in type checking. If e.g. it is possible to infer the type of a variable used in the output or construction part of a query (maybe the type is implied by a selection), this variable is well typed with respect to a given type, if the inferred type is a subtype of the type given by the programmer.

Another very practical use case is schema checking for special document schemata: if one wants to make a schema for HTML documents of a certain shape, e.g. a web page supporting the corporates look and feel by using certain navigation elements, testing that this schema represents a subset of HTML is desirable.

The approach for subset testing presented here is based on the simulation preorder between two automata. A simulation preorder is a relation between state transition systems associating systems which behave in the same way in the sense that one system simulates the other. Formally, given a state transition system with states S, a simulation preorder is a binary relation $R \subseteq S \times S$ such that if $(p,q) \in R$, then for each transition $p \xrightarrow{a} p'$ there is a transition $q \xrightarrow{a} q'$ such that $(p',q') \in R$.

For string language automata (DFA's or NFA's) A_1 and A_2, simulation preorder is specialised in such a way, that A_1 and A_2 are in simulation preorder—written $A_1 \preceq A_2$ later on— if each initial state of A_1 simulates in an initial state of A_2 and for each final state of A_1 there is a final state in A_2 in which it simulates. For automata defined as $A = (S,T,F,s_0,\Sigma)$ where S is the set of states, $T \subseteq S \times \Sigma \times S$ is the set of transitions, $s_0 \in S$ is the start state and $F \subseteq S$ is the set of final states, the definition of the simulation preorder over label equality can be written as:

$$
\begin{array}{rcl}
A_1 \preceq A_2 & \text{iff} & \forall (s,l,e) \in T_{A_1} \exists (s',l',e') \in T_{A_2}.(s,l,e) \preceq (s',l',e') \\
(s,l,e) \preceq (s',l',e') & \text{iff} & l = l' \wedge \\
 & & \forall (e,\bar{l},\bar{e}) \in T_{A_1} \exists (e',\bar{l}',\bar{e}') \in T_{A_2}.(e,\bar{l},\bar{e}) \preceq (e',\bar{l}',\bar{e}')
\end{array}
$$

Extending the definition of automata simulation to graph automata simulation is strait forward: the recursive $\preceq$ condition is tested along both dimensions of the hyper edges as used in the tree automata:

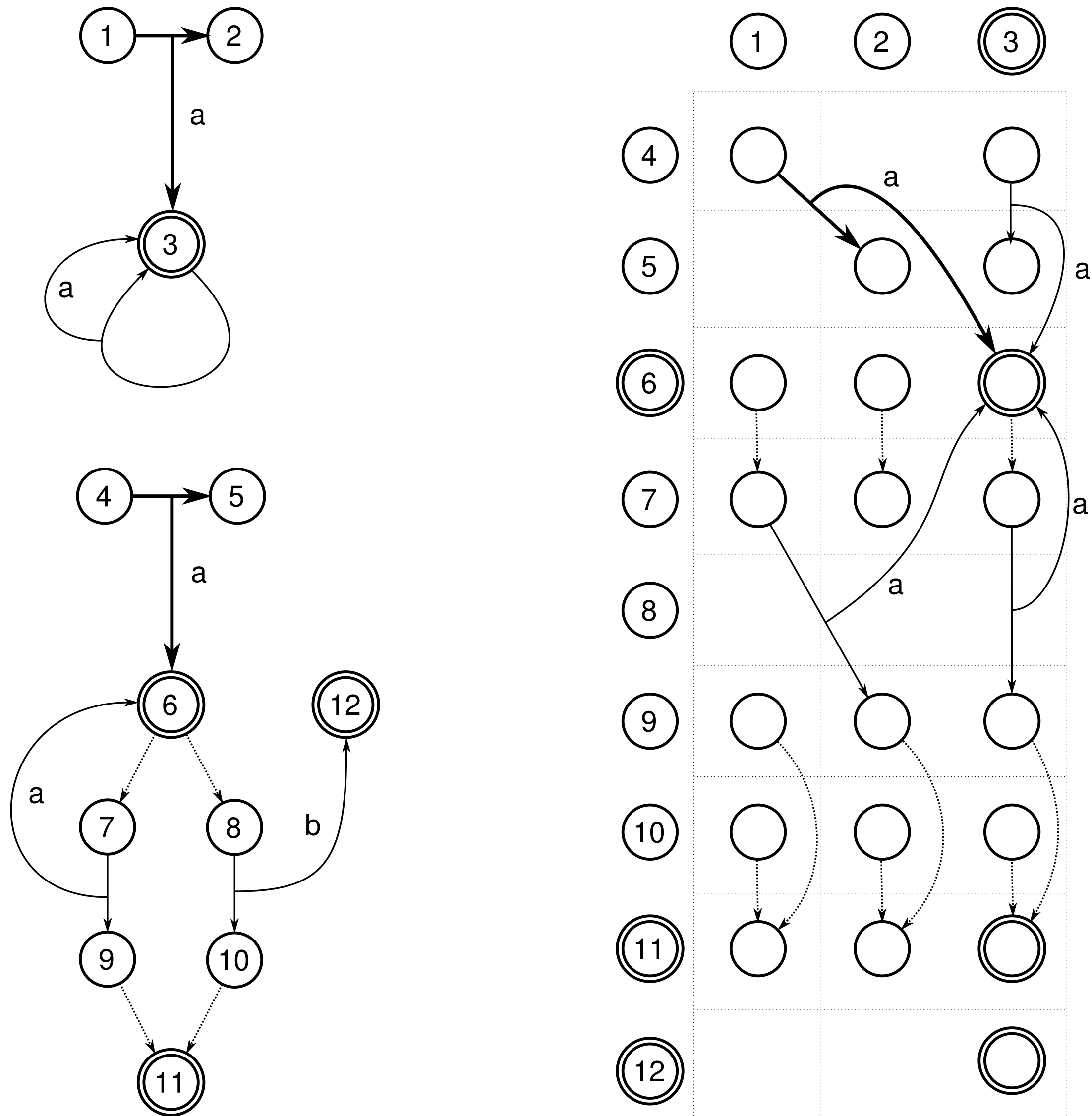


Figure 6.3: The product automaton on the right accepts the intersection of the graph language of the two automata on the left.

$$
\begin{aligned}
A_1 \preceq A_2 \quad &\Rightarrow \quad \forall (s,l,c,e) \in T_{A_1} \exists (s',l',c',e') \in T_{A_2}.(s,l,c,e) \preceq (s',l',c',e') \\
(s,l,c,e) \preceq (s',l',c',e') \quad &\Rightarrow \quad l = l' \ \wedge \\
&\quad \forall (e,\bar{l},\bar{c},\bar{e}) \in T_{A_1} \exists (e',\bar{l}',\bar{c}',\bar{e}') \in T_{A_2}.(c,\bar{l},\bar{c},\bar{e}) \preceq (c',\bar{l}',\bar{c}',\bar{e}') \ \wedge \\
&\quad \forall (e,\vec{l},\vec{c},\vec{e}) \in T_{A_1} \exists (e',\vec{l}',\vec{c}',\vec{e}') \in T_{A_2}.(e,\vec{l},\vec{c},\vec{e}) \preceq (e',\vec{l}',\vec{c}',\vec{e}')
\end{aligned}
$$

6.4.3.1 An Algorithm for Subset Graph on Tree Automata

As a sketch for implementation and for complexity analysis of the presented simulation relation on tree automata, the following algorithm is proposed. The algorithm is applied to two automata A_1 and A_2:

- a two dimensional matrix of truth values of size $|T_{A_1}| \times |T_{A_1}|$ is initialized in such a way, that for each transition pair $(\tau_1, \tau_2) \in T_{A_1} \times T_{A_1}$ the corresponding field in the matrix is *true*, if the labels of τ_1 and τ_2 are identical and $false$ otherwise.
- set each matrix field with value *true* to $false$, if $((s_1,l,c_1,e_1),(s_2,l,c_2,e_2))$ is the corresponding transition pair and either s_1, c_1 or e_1 is a final state but not the corresponding s_2, c_2 or e_2.
- modify the matrix until a fix point is reached by
  - set each matrix field with value *true* in the matrix with corresponding transition pair $((s_1,l,c_1,e_1),(s_2,l,c_2,e_2))$ to $false$, if for any transition $\tau_1 = (e_1,\vec{l},\vec{c_1},\vec{e_1})$ in A_1 there is no corresponding transition $\tau_2 = (e_2,\vec{l},\vec{c_2},\vec{e_2})$ in A_2 such that the field (τ_1,τ_2) in the matrix is *true*.
  - set each *true* field in the matrix with corresponding transition pair $((s_1,l,c_1,e_1),(s_2,l,c_2,e_2))$ to $false$, if for any transition $\tau_1 = (c_1,\bar{l},\overline{c_1},\overline{e_1})$ in A_1 there is no corresponding transition $\tau_2 = (c_2,\bar{l},\overline{c_2},\overline{e_2})$ in A_2 such that the field (τ_1,τ_2) in the matrix is *true*.
- If for any transition $\tau_1 \in A_1$ there is no corresponding transition $\tau_2 \in A_2$ such that (τ_1,τ_2) in the matrix is *true*, then the language accepted by A_1 is not a subset of the language accepted by A_2.

Example 6.5
The condition $A_1 \subseteq A_2$ is to be tested using the presented algorithm.

A run of the algorithm is visualized with a table representing the matrix. The edges of A_1 are used as column labels and the edges of A_2 as row labels. Final states are emphasized using a **bold** font. The cells contain a series of ones (1) and zeros (0) representing the truth values *true* and $false$ a field has in various stages of the computation. Note, that if a 0 occurs in the cell, the 0 is the ultimate value of this cell, as the algorithm only changes *true* values, in case of conflicts, to $false$ values. 4 states of computations are represented, so either a cell contains $1,1,1,1$ and is thereby *true*, or it contains less entries where the last state is 0, e.g. $1,1,0$. The stages represented are:

1. after performing the label check,
2. after checking, that final states in transitions of A_1 fall on final states of corresponding transitions of A_2
3. first iteration of checking following transitions in both dimensions (two cells changed truth value)

$$
\begin{array}{lll}
A_2 = (& \{ & 1,2,3,4,5,6,7,8,9\}, \\
 & \{ & (1,a,3,\mathbf{2}), \\
 & & (3,a,3,\mathbf{4}), \\
 & & (4,a,3,\mathbf{6}), \\
 & & (\mathbf{6},b,\mathbf{6},7), \\
 & & (7,c,\mathbf{8},\mathbf{9}), \\
 & & (6,b,5,\mathbf{6}), \\
 & & (5,a,3,\mathbf{6}), \\
 & & (3,a,3,\mathbf{6})\}, \\
 & \{ & 2,6,8,9\}, \\
 & & (1,a,3,\mathbf{2}), \\
 & \{ & a,b,c\})
\end{array}
$$

$$
\begin{array}{lll}
A_1 = (& \{ & 10,11,12,13,14,15\}, \\
 & \{ & (10,a,12,\mathbf{11}), \\
 & & (12,a,12,\mathbf{14}), \\
 & & (\mathbf{14},b,15,\mathbf{14}), \\
 & & (15,a,12,\mathbf{13})\}, \\
 & \{ & 11,13,14\}, \\
 & & (10,a,12,\mathbf{11}), \\
 & \{ & a,b\})
\end{array}
$$

Figure 6.4: Two automata used to demonstrate the sub-language test.

Figure 6.5: Visual representation of the two automata in example 6.4

4. second (and last) iteration of checking following transitions in both dimensions (no cells changed truth value)

$\downarrow A_2 \quad A_1 \rightarrow$	$(10,a,12,\mathbf{11})$	$(12,a,12,\mathbf{14})$	$(\mathbf{14},b,15,\mathbf{14})$	$(15,a,12,\mathbf{13})$
$(1,a,3,\mathbf{2})$	1,1,1,1	1,1,0	0	1,1,1,1
$(3,a,3,4)$	1,0	1,0	0	1,0
$(4,a,3,\mathbf{6})$	1,1,1,1	1,1,1,1	0	1,1,1,1
$(\mathbf{6},b,7,\mathbf{6})$	0	0	1,1,0	0
$(7,c,\mathbf{8},\mathbf{9})$	0	0	0	0
$(\mathbf{6},b,5,\mathbf{6})$	0	0	1,1,1,1	0
$(5,a,3,\mathbf{6})$	1,1,1,1	1,1,1,1	0	1,1,1,1
$(3,a,3,\mathbf{6})$	1,1,1,1	1,1,1,1	0	1,1,1,1

After the last iteration, the columns are checked for consistency, i.e. each column should contain at least one $true$ cell, so a table cell with $1,1,1,1$. As this is the case, A_1 is an automaton accepting a sub-language of the language represented by A_2.

An Upper Bound Complexity for Subset Test The presented algorithm shows, that the subset test has an upper bound of polynomial time and space complexity. The space complexity is determined by the matrix that is of the size of the product of the number of transitions of both automata, i.e. $O(|\Delta_1| \times |\Delta_2|)$. The time complexity is the sum of initializing the matrix (including label test and final state condition) and the iterative refinement of the matrix. The refinement process must terminate, as either no change is made to the matrix and then the refinement is over, or at least one cell changes truth value from $true$ to $false$. Truth values are never altered from $false$ to $true$ again. Assuming the worst case, that on each iteration process just one cell is altered, we need $|\Delta_1| \times |\Delta_2|$ iterations, each iteration has a complexity of $O(|\Delta_1| \times |\Delta_2|)$. The final step is the consistency check of the columns, which also takes $O(|\Delta_1| \times |\Delta_2|)$ time. The total costs therefore are $O(\ (k + (|\Delta_1| \times |\Delta_2|)) \times (|\Delta_1| \times |\Delta_2|)\ )$ with k as factor for initialisation cost and consistency check of one cell.

Chapter 7

A Model for Regular Languages of Multisets

The analysis of sets of multisets in the framework of language theory as introduced by Chomsky has first been considered in [40]. In this work Rohit J. Parink analyses properties of context free languages when order of the symbols of words is ignored. He concludes that, under irrelevance of symbol order, regular and context free languages are of same expressiveness, i.e. for each context free language, there is a regular language representing the same language *modulo* order.

The analysis of multisets, or unordered sequences, is an important topic in the context of this deliverable, as R_2G_2 has been conceived to model graph shaped data with ordered sequences **and unordered multisets** of sibling nodes, usually as children of a parent node. The languages without symbol order analyzed by Rohit J. Parikh are exactly the class of languages used by R_2G_2 for modelling multisets of nodes. Moreover, the language class in question is general enough to capture existing models of unordered node sequences, as e.g. modelled in XML Schema.

7.1 Introduction to Multisets and Multiset Languages

7.1.1 Multisets

A multiset is a set, container or bag in which multiple occurrences of the same value may occur. This differs from the usual set in the possibility of multiplicity of the same values.

Definition 7.1 (Syntax of Multisets)
In the following, a multiset M will be denoted in the usual set notation, as $M = \{v_1, \ldots, v_n\}$ with $v_i \in \Sigma$ where Σ is the set of defined values for M, and v_i and v_j with $i \neq j$ can be the same element in Σ. Σ has (by now) to be a finite set of symbols.

- $\{\}$ is a multiset
- if $\{$ *content* $\}$ is a multiset and v is a valid defined value, then $\{ v, content \}$ is a multiset.

Hence, $M \in \bigcup_{i \in \mathbb{N}} \Sigma^i$.

Note, that $M \neg \subseteq \Sigma$, as M is not a set, the subset relation is not even well typed here. This is due to the fact, that there may be $i \neq j$ such that $v_i = v_j$ for $v_i, v_j \in M$.

As multisets do not impose an order on their elements, two multisets with different notation according to the syntax may be considered equivalent. As an example, the multiset $\{v_1, v_2, v_1\}$ is the same as the multiset $\{v_2, v_1, v_1\}$. This can be defined by interpreting the multiset member separator (in this case the comma) as an associative commutative function.

Definition 7.2 (Semantics of Multisets)

- $\{\ v, content\ \} \ = \ \{\ content, v\ \}$
- $\{\ v_1, (v_2, content)\ \} \ = \ \{\ (v_1, v_2), content\ \}$

An important property is the multiplicity of the symbols of Σ in a given multiset. The *Parikh Mapping*, see section 7.2, defines this multiplicity.

7.1.2 Multiset Languages

A multiset language is a set of multisets, hence the multiset language L over Σ is $L \in \bigcup_{i\in\mathbb{N}} \Sigma^i$. What was called "*set of defined values*" in the definition of multisets is often called *vocabulary* in linguistics or *symbols* in language theory.

A language of multisets may be finite or infinite, hence a finite notation for languages of multisets is necessary to formalize them.

7.1.2.1 Existing Approaches to Formalize Multiset Languages

Different approaches have been proposed to formalize sets of multisets, among them so called *multiplicity lists* [17], $\mathcal{L}$-formulae [39] and so called *counting constraints* [67].

Definition 7.3 (Multiplicity Lists)
A multiplicity list is a regular type expression of the form $s_1(n_1 : m_1) \cdots s_k(n_k : m_k)$ where $k \geq 0$ and $s_1, \ldots, s_k$ are distinct symbols. By n_i the minimum and by m_i the maximum number of occurrences is expressed.

A problem of the interval notation is, that it is not possible to express gaps in the multiplicity of the occurrence of a symbol. It is e.g. not possible to express, that the multiplicity of a symbol should be at least n and at most m but not p with $n < p < m$. Further, it is e.g. not possible to express, that the multiplicity of the symbols should be an odd or an even number or a multiple of a given factor.

Definition 7.4 ($\mathcal{L}_{\geq}$-Formulae)
A $\mathcal{L}_{\geq}$-Formulae is an expression φ of the shape

$$\varphi ::= true|false|a = i|a \geq i|\neg\varphi|\varphi \vee \varphi \quad a \in \Sigma, i \in \mathbb{N}$$

and the language of *varphi* is $\mathcal{L}(\varphi)) = \{w|w \models \varphi\}$ with $w \models a = i$ iff the multiplicity of w is i, $w \models a \geq i$ iff the multiplicity of w is greater or equal i, $w \models \neg\varphi$ iff not $w \models \varphi$, $w \models \varphi_1 \vee \varphi_1$ iff $w \models \varphi_1$ or $w \models \varphi_2$, never $w \models false$ and always $w \models true$.

$\mathcal{L}_{\geq}$-Formulae are closed under union, intersection and complement.

Compared to multiplicity lists, $\mathcal{L}_{\geq}$-Formulae are more expressive, as they allow the expression of gaps in symbol multiplicity intervals. However, gaps have to be expressed, there is no abstraction over the concept of the gap. If e.g. odd multiplicity has to be expressed, it is needed to declare a gap of one after each valid symbol. Such multiplicities are commonly expressed e.g. using regular expressions, in the case of odd multiplicity for the symbol s, the regular expression $s, (s, s)*$ can be used to express it.

Definition 7.5 (Counting Constraints (a.k.a. Presburger Constraints))
A counting constraint is a formula φ of the shape:

$$\varphi, \psi ::= (Exp_1 = Exp_2)|\neg\varphi|\varphi \vee \psi|\exists N.\varphi Exp_i ::= n|N|Exp_1 + Exp_2$$

where n is a natural number, N is natural number variable. Later on, $\varphi(N_1, \ldots, N_m)$ will denote the counting constraint with the free variables $N_1, \ldots, N_m$. Counting constraints are decidable, i.e. for every $\varphi(N_1, \ldots, N_m)$ it is decidable, if there exists n_i for all free variable such that $\varphi(n_1, \ldots, n_m)$ is true.

In contrast to $\mathcal{L}_{\geq}$-Formulae, counting constraints are able to express repeating interval/gap multiplicities by the use of variables and sums for multiplicities. As an insight, consider the odd multiplicity of a symbol—let the multiplicity of that symbol be S, then $S = 1 + N + N \wedge N \geq 0$ for N and S as natural numbers expresses the fact, that S is an odd number.

Yet counting constraints are decidable, the deciding complexity in general is high [27]. In practice it turned out, that many efficient approaches based on heuristics exist [28], making type checking using counting constraints feasible.

7.1.2.2 Motivating a new Approach

Type and schema declarations for XML based languages are usually based on regular expressions, content models are ordered sequences of elements. For unordered content usually, as introduced earlier in this section, different formalisms are used. The new approach suggested in this deliverable is to use regular expressions with unordered semantics as multiset language formalism. This releases the user from the burden to learn different formalisms for ordered and unordered content. Further, it gives rise to a simple treatment of ordered types under unordered queries as presented now:

Often, when querying, and sometimes when modelling, the order is irrelevant. When querying XML using XPath it is generally possible to query data based on the sequential order (using the so called `following`, `preceding`, `following-sibling` and `preceding-sibling` axis), but in many applications the order of the elements is ignored, especially when querying database like documents. In Xcerpt, explicitly ordered or unordered query patterns are used to query documents. An unordered query can be considered to have a type with unordered content specification, which itself is a language of multisets. This is motivated by the fact, that it is arguably reasonable to interpret the type of a query as the type representing all data instances that can be queried by the query term under the assumption of a given type or schema for the queried data. If no type or schema is given for the queried data, the most general type may safely be assumed. The type of a query is a subset of the type of the queried data. When a query has multiple occurrences of the same variable, it gets necessary to calculate the variable intersection to infer the type of a query, as shown in section 8.3.2. When constructing unordered content, e.g. in Xcerpt, out of queries to ordered typed data, it is easy construct content that is of a type that is the unordered interpretation of an ordered content model. As an example consider the following Xcerpt rule:

```
CONSTRUCT
  unorderedData{ all var C }
FROM
  orderedData[[ var C ]]
END
```

To infer the type of the result of of that rule, it is necessary to give an unordered interpretation of the type of the content queried by the query pattern `orderedData[[ var C ]]`. Unfortunately, such

a type may not be represented precisely using multiplicity lists or $\mathcal{L}_{\geq}$-Formulae. As an example, think of a regular content of the shape $(C,C)*$ which models a content with an even number of elements of type C. An unordered interpretation of such a type should arguably reflect the even multiplicity of it's C-typed members. Neither $\mathcal{L}_{\geq}$-Formulae nor multiplicity lists allow this. With counting constraints or Presburger arithmetic, the desired set of multisets can be expressed.

R_2G_2 uses regular expressions with unordered interpretation to model unordered content models. For the operational semantics, counting constraints or Presburger arithmetic formulae, very much in the spirit of Scheaves automata [67], are used. The main difference to Scheaves automata and Scheaves logic lies on the surface—unordered content models in R_2G_2 are not specified using a special formalism, but with regular expressions.

7.2 Counting Constraints

Languages of Multisets A language of multisets over (finite) alphabet Σ is defined as a (possibly infinite) set of finite multisets over alphabet Σ.

Parikh Mapping For a word w of a finite alphabet Σ with m symbols (i.e. $|\Sigma| = m$), the Parikh Mapping $\#(w) = (n_1, \ldots, n_m)$ is a vector, where n_i denotes the number of occurrences of the i's symbol of Σ (as Σ is finite, it is enumerable, it is hence sensible to talk of a symbol at position i) in the word w. For example, for $\Sigma = \{a,b,c,d\}$ and $w =$ "*cabba*", the Parikh mapping $\#(w) = (2,2,1,0)$. A language of multisets can also be defined as a set of vectors as returned by the Parikh mapping.

Counting Constraint A counting constraint is a formula φ of the shape:

$$\varphi, \psi ::= (Exp_1 = Exp_2) | \neg\varphi | \varphi \vee \psi | \exists N.\varphi$$

$$Exp_i ::= n | N | Exp_1 + Exp_2$$

where n is a natural number, N is natural number variable. $\varphi(N_1, \ldots, N_m)$ denotes the counting constraint with the free variables $N_1, \ldots, N_m$. Counting constraints are decidable, i.e. for every $\varphi(N_1, \ldots, N_m)$ it is decidable, if there exists n_i for all free variable such that $\varphi(n_1, \ldots, n_m)$ is true.

For a language $\mathcal{L}$ of multisets over Σ, a counting constraint φ with free variables for every symbol in Σ can be used to test membership of a word (or multiset) using the Parikh mapping: $w \in \mathcal{L}$, when

$$\#(w) = (n_1, \ldots, n_{|\Sigma|}) \wedge \varphi(n_1, \ldots, n_{|\Sigma|})$$

The expression can be understood as follows: $\#(w)$ yields a Parikh mapping, which is the multiplicity of each symbol of the language in the given word w. The given multiplicity is related to variables $(n_1, \ldots, n_{|\Sigma|})$, one for each symbol of the set of symbols of the language. The word w is in the language modelled by φ, if instantiating the variables (i.e. $\varphi(n_1, \ldots, n_{|\Sigma|})$) according to the Parikh mapping holds.

Defining Languages of Multisets with Regular Expressions Given a regular expression r, $\mathcal{L}(\{r\})$ is defined as the language of multisets for r such that, for a word w there exists a permutation $w' \in \mathcal{P}(w)$ such that $w' \in \mathcal{L}(r)$ (w' is in the regular (string) language $\mathcal{L}(r)$).

7.3 A Calculus for Translation of Regular Expressions to Counting Constraints

For a given regular expression, it is possible to derive a counting constraint, such that the counting constraint accepts exactly the language of multisets defined by (the permutation of) all words accepted by the regular expression.

As an example, consider the regular expression `(A,B)*,C` : The language of multisets defined by this regular expression is the set of all multisets with the same amount of `A`s and `B`s and exactly one `C` for e.g. an alphabet $\Sigma = \{A,B,C\}$. A counting constraint for this language could be

$$\exists A \exists B \exists C \ . \ A = B \wedge C = 1$$

The following set of rules describes non deterministically a simple algorithm for the translation of regular expressions interpreted as definitions of languages of multisets to counting constraints. The regular expression is given in curly braces (i.e. $\{re\}$) in accordance to the notation in R_2G_2 and for technical reasons to distinguish the root of the abstract syntax tree of (recursively defined) regular expressions. The domain of the rules is $\Gamma, N \ / \ re \mapsto C$ where N is a natural number variable or a natural number to be used in the counting constraint, *re* is the regular expression, C is the, maybe partially specified, counting constraint, and Γ is a mapping of symbols as used in the regular expression to sets of natural number variables. To ease the formal handling in the rules, the mapping Γ is used as a set of equations, where the equations have the symbol on the left hand side and the set of natural number variables as the sum on the right hand side. The role of N is a bit unconventional, as it is not a variable for a natural number, it is a meta-variable for either a natural number (indeed, only the number 1 occurs) or for a natural number variable. It is hence sometimes needed to construct a natural number variable—fortunately only new variables are needed, they are constructed using the symbol N^{new}. The rules relate one expression of the domain above the line with many expressions of the domain below the line, where usually the expressions below the line are decompositions of the expression above the line.

For a regular expression with the atoms (or symbols) $a_1, \ldots, a_m$, the general scheme of the resulting counting constraint is

$$\begin{array}{l} \varphi(X_{a_1}, \ldots, X_{a_m}) \\ \quad ::= \\ \quad \exists Y_{1_1}, \ldots, Y_{1_p}, \ldots, Y_{m_1}, \ldots, Y_{m_q}. \\ \quad X_{a_1} = Y_{1_1} + \cdots + Y_{1_p} \wedge \ldots \wedge X_{a_m} = Y_{m_1} + \cdots + Y_{m_p} \\ \quad \wedge \varphi(\ldots, Y_{i_j}, \ldots) \end{array}$$

The variables X_{a_i} are then bound to all the variables in the mapping Γ for the symbol a_i. This is formalized in the root rule (the last in the following set of rules), applied to the regular expression, the resulting counting constraint represents the language of multisets defined by the regular expression.

Atoms (or symbols) in regular expressions are at the leaf level of the rule based constraint construction. Indeed, the rule based construction spans a tree structure equally shaped to the abstract syntax tree of the regular expression, as there is exactly one rule for the decomposition of one abstract syntax tree node. So, when an atom occurs, the multiplicity of this symbol, represented by the natural number meta-variable N, has to be propagated to the mapping Γ, which is to be used in the end at root level.

$$\frac{\begin{array}{c}(a, \{\ldots, N, \ldots\}) \in \Gamma \\ \Gamma, N \ / \ a \mapsto \emptyset\end{array}}{} \qquad \text{(ATOM)}$$

For a sequence of two regular expressions, the multiplicity of the current context is passed to the two components, the partial constraints of the two parts are connected using a conjunction, the resulting mappings of the two regular part-expressions are also merged. This reflects the fact, that the multiplicity of the regular expression rs implies that the expression parts r and s have to occur also in the same multiplicity in a valid multiset.

$$\frac{\Gamma_r \cup \Gamma_s, N \;/\; rs \mapsto C_r \wedge C_s}{\Gamma_r, N \;/\; r \mapsto C_r \qquad \Gamma_s, N \;/\; s \mapsto C_s} \tag{SEQ}$$

For a disjunctive regular expression it's multiplicity has to be divided between the two options. This is reflected by the sum $N = M + P$, where N is the multiplicity of the disjunction, and M and P are the multiplicity of the two components of the disjunction. The sum is a new part of the counting constraint. Note, that M and P are both new variables constructed using the N_{new} constructor.

$$\frac{\Gamma_r \cup \Gamma_s, N \;/\; r|s \mapsto C_r \wedge C_s \wedge N = M + P}{\Gamma_r, M = N_{new} \;/\; r \mapsto C_r \qquad \Gamma_s, P = N_{new} \;/\; s \mapsto C_s} \tag{DISJ}$$

For an optional regular expression occurring N times, the optional sub-expression can at most also occur N times. This is expressed using a new multiplicity variable M for the sub expression and the inequality $M \leq N$.

$$\frac{\Gamma, N \;/\; r? \mapsto C \wedge M \leq N}{\Gamma, M = N_{new} \;/\; r \mapsto C} \tag{OPT}$$

The Kleene star is a bit tricky: the sub regular expression r could be repeated arbitrary often. It's multiplicity is independent of the multiplicity of $r*$, the sub expression hence gets a new multiplicity variable. On the other hand, if $r*$ does not occur at all in the valid word, then the sub expression r cannot occur neither. This is reflected by the new constraint part $\neg(N = 0 \wedge M \neq N)$—if N is 0 then M is also 0, otherwise anything for M and N is OK.

$$\frac{\Gamma, N \;/\; r* \mapsto C \wedge \neg(N = 0 \wedge M \neq N)}{\Gamma, M = N_{new} \;/\; r \mapsto C} \tag{KLEENE}$$

The regular expression plus construct has, like the Kleene star, also to consider the two cases, that either the whole expression $r+$ does not occur at all in the valid word, or that it occurs. If it does not occur, then the part expression neither occurs, otherwise the sub expression at least occurs as often as the expression.

$$\frac{\Gamma, N \;/\; r+ \mapsto C \wedge ((N > 0 \wedge M \geq N) \vee (N = 0 \wedge M = 0))}{\Gamma, M = N_{new} \;/\; r \mapsto C} \tag{PLUS}$$

The root of a regular expression—more precisely of it's abstract syntax tree—is used to finalize the constraint by adding the information of the symbol mapping.

The multiplicity meta-variable is set to one, as a valid word with respect to the regular expression fully fits exactly once in the regular expression.

The mapping Γ is used in the previous rules to capture from all the variable expressions the multiplicities relating them to the corresponding symbol itself. The mapping set gets new members in the Atom rule and possibly alters the members in the sequence and disjunction rule (whenever two occurrences of the same symbol are represented in two mappings to be merged, the member becomes the sum of the two occurrences).

In the root rule the members of the mapping are now interpreted as sums (as which they syntactically occur), and are all added as conjunction to the constraint. All (free) right hand side variables are existentially bound, the symbols from the mapping (on the left hand side of the mapping equations) are now interpreted as variables for the multiplicity of their corresponding symbol, they are the only free variables of the constraint. Every binding of the free variables which yields a solution of the constraint, gives the Parikh mapping of multisets in the language modelled by the constraint.

$$\frac{\{\},1\ /\ \{r\} \mapsto \exists \ldots, N_{i_j}, \ldots . C \wedge a_1 = N_{1_1} + \cdots + N_{1_p} \wedge \cdots \wedge a_m = N_{m_1} + \cdots + N_{m_q}}{\{a_1 = N_{1_1} + \cdots + N_{1_p}, \ldots, a_m = N_{m_1} + \cdots + N_{m_q}\},1\ /\ r \mapsto C}\ (\text{REGEXPROOT})$$

7.3.1 Example of a Regular Expression Translated to a Counting Constraint

The calculus for translation of regular expressions to counting constraints will be applied to the following example expression:

$$((a,b)^*,c)|(d,e^?)$$

In words, this regular expression models a language of multisets with the following properties:

- the set either contains a c or a d.
- If there is no c in the set, then there is neither a b.
- Further, If there is no c (and hence there is a d) in the set, then there is at most one a contained in the set.
- if there is no d (and hence there is a c), there are as many bs as as in the set.

Let's see, what kind of counting constraint the calculus produces of the regular expression:

$$\dfrac{\{\},1/\ ((a,b)^*,c)|(d,a^?) \ \mapsto \exists O \exists P \exists M \exists N \ .\ a = O+P \wedge b = P \wedge c = M \wedge d = N \wedge \neg(M = 0 \wedge M \neq P) \wedge 0 \leq N \wedge 1 = M+N}{\dfrac{\{a = O+P, b = P, c = M, d = N\},1/\ ((a,b)^*,c)|(d,a^?) \ \mapsto \neg(M = 0 \wedge M \neq P) \wedge 0 \leq N \wedge 1 = M+N}{\dfrac{\{a = P, b = P, c = M\},M/\ ((a,b)^*,c) \ \mapsto \neg(M = 0 \wedge M \neq P)}{\dfrac{\{a = P, b = P\},M/\ (a,b)^* \ \mapsto \neg(M = 0 \wedge M \neq P)}{\dfrac{\{a = P, b = P\},P/\ a,b \ \mapsto \emptyset}{\dfrac{\{a = P\},P/\ a \ \mapsto \emptyset}{} \quad \dfrac{\{b = P\},P/\ b \ \mapsto \emptyset}{}}} \quad \dfrac{\{c = M\},M/\ c \ \mapsto \emptyset}{}} \quad \dfrac{\{a = O, d = N\},N/\ (d,a^?) \ \mapsto 0 \leq N}{\dfrac{\{d = N\},N/\ d \ \mapsto \emptyset}{} \quad \dfrac{\{a = O\},N/\ a^? \ \mapsto 0 \leq N}{\dfrac{\{a = O\},O/\ a \ \mapsto \emptyset}{}}}}}$$

The counting constraint is the consequence above the top line

$$\exists O \exists P \exists M \exists N \ .\ a = O+P \wedge b = P \wedge c = M \wedge d = N \wedge \neg(M = 0 \wedge M \neq P) \wedge 0 \leq N \wedge 1 = M+N$$

and arguably fulfills our verbal requirement.

7.4 Some Set Theoretic Computations on Counting Constraints

As for hyper graph automata, for the purpose of static type checking, it is necessary to analyse the same set properties on languages modelled using counting constraints: (1) emptiness test, (2) computation of language intersection, and (3) subset test.

Testing Emptiness of a Multiset Language defined by Counting Constraints Using Counting Constraints it is possible to define empty languages due to unfulfillable constraints. The declaration $\exists : Aa\&Bb : A = B \wedge A \leq 1 \wedge B \geq 2)$ has unsatisfiable constraints and declares therefore an empty language. Hence, the emptiness test for a language can be reduced to the satisfiability test of the counting constraints. In the context of R_2G_2, the counting constraints generated as a regular expression never models an empty language. nevertheless, in the context of type checking, empty languages may result as intermediate step of the type checking process, i.e. the intersection[1] of two languages defined using Counting Constraints, hence the emptiness check is of practical relevance.

As solutions have to be in $\mathbb{N}$ (because the variables are defined variables in $\mathbb{N}$—variables represent multiplicities of symbols in the end, and non integer multiplicities of symbols make no sense), a lower bound is always available for the variables (e.g 0 or given by a constraint $n \leq X$). There is not necessarily an upper bound for the solutions (note, that languages can be infinite sets of multisets), but as all multisets are finite, there are always smallest ones (e.g. with the least amount of symbols). It is always possible to predict an upper bound for the smallest word, such that if no solution is found within the lower and the upper bound for all variables, the language must be empty. As a reason for this, consider, that Presburger arithmetic formulas are decidable.

Given upper and lower bounds in $\mathbb{N}$ for all variables, finding a solution is a finite domain problem, that always terminates. Unfortunately, the upper bound therefore is $(u-l)^{|Vars|}$ with $|Vars|$ for the number of variables, u as upper bound and l as lower bound. For all linear equation parts and for some non linear ones, fortunately there exist solvers with polynomial complexity. Combining them with search over the non tractable sub part of the problem possibly yields reasonable average case behaviour.

An evidence for a non empty set is given by the multiset that results when applying the solution to the counting constraints of the tested language.

Membership Test with Counting Constraints The membership test finds out, if a multiset w is contained in the set of multisets defined by the counting constraints C (the set of multisets may also be called the language $\mathcal{L}(C)$). The test can be achieved by instantiating the free variables of C with the corresponding number of occurrences of the corresponding symbols in w.

The test can easily be expressed as a specialisation of the emptiness test: if a multiset w is in $\mathcal{L}(C)$,

there must be a solution, such that $w = T$ under C. For $w = a_1^{m_1} \oplus \cdots \oplus a_n^{m_n}$ and $T = a_1^{M_1} \oplus \cdots \oplus a_n^{M_n}$, therefore $w = T$ is $w = a_1^{m_1} \oplus \cdots \oplus a_n^{m_n} = T = a_1^{M_1} \oplus \cdots \oplus a_n^{M_n}$ which is to be interpreted as $m_1 = M_1 \wedge \cdots \wedge m_n = M_n$. Solving $w = T \wedge C$ with the solver used for emptyness test, yields that w is a valid multiset in $\mathcal{L}(M)$, if there is a solution.

Intersection of Languages declared using Counting Constraints The intersection of two Languages $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ is an extension of the membership test: The intersection of $M_1 = (T_1, C_1)$ and $M_2 = (T_2, C_2)$ is the language, where $T_1 = T_2$ under $C_1 \wedge C_2$. The problem can be reformulated as $C_1 \wedge C_2 \wedge M_{1_1} = M_{2_1} \wedge \cdots \wedge M_{1_n} = M_{2_n}$ where $T_i = M_{i_1} \oplus \cdots \oplus M_{i_n}$.

[1] Intersection of two languages defined using Counting Constraints is introduced later

Part III

Type Checking

Chapter 8

Type Checking using Regular Rooted Graphs as Data Paradigm

This chapter can be considered the core of this deliverable. After a brief introduction about types, programming and query languages, a typing approach for Xcerpt based on the automaton model presented of Chapter 6 (and hence based on R_2G_2) is presented.

8.1 Types and Query- and Programming Languages

In Programming and Query languages types are used to give an approximation of the values certain components and constructs of the language can obtain. Usually this approximation is a super set of the concrete values to which the constructs are evaluated during run time. One use of the approximations is to provide specialized and appropriate memory representations for data instances. A dual use of those approximations is to explain, find or prevent errors due to possibility or necessity of invalid values at run time. Different approaches to the recognition of such errors exist:

1. The errors can be detected at run time by *validating* the data instances against the types.
2. The errors are detected before run time, e.g. at compile time by analysing the compatibility of the types.

8.1.1 Dynamic Typing and Type Checking in Programming Languages

In dynamically typed languages the **values** have an associated type and the type correctness is checked at run time. The type check is usually explicitly programmed in the core libraries and, if needed or wished, implemented by the users of the given programming language in their own programs. The concept of dynamic typing and dynamic type checking is mostly inspired by practise—the explicit type check is easy to include in the run time system of a programming language and no static type system has to be conceived for the programming language. Most common scripting languages like Perl, Python, Ruby, etc. are conceived with the so called *dynamic typing* facilities. Hence, as well as static type checking, dynamic typing can prevent type related *system corruption* in the sense, that operations boiling down to predefined functions interacting with the system are checked for type correct application at run time. This distinguishes dynamically typed languages from untyped languages.

In contrast to statically typed languages, that prevent even execution of wrongly typed programs, the error may occur relatively *late*, hence arguably along the execution line, prior or parent functions may be considered to have been applied violating types without warning.

The use of static type systems usually hinder the application of some programming techniques, that may be semantically well founded and intended by the programmer, but not well typed within the framework of the type system. By example, most type systems have a notion of well typed lists or arrays, if they contain just elements of one type. Nevertheless, it may be convenient to have mixed type lists for some applications. With dynamic typing the restrictions of what can be programmed are usually lower than with common wide spread statically typed programming languages, yet type errors can still be discovered easier than with untyped languages.

8.1.2 Static Typing and Type Checking in Programming Languages

In statically typed languages the **program fragments**, e.g. the variables or the data constructors or the function invocations, have an associated type and the type correctness can be checked prior to run time, usually at compile time. Examples of statically typed programming languages are SML, Haskell and many others.

A common approach in many programming languages is to deny the compilation of ill typed programs, where ill typed programs are such programs, where multiple occurrences of the same variable have different types. The term *different types* is often, e.g. in object oriented languages, relaxed to different types that are not in a subtype, i.e. subset, relationship.

If a program is not ill typed, it can be assured, that all function invocations are well typed, hence there often no need for dynamic type checking at run time. The type information can hence be discarded at run time, the run time system may be completely untyped yet program evaluation will not yield any type errors. Hence a veteran of static type checking, Robin Milner states *"Well-typed programs do not go wrong"* [37], many run time errors are still possible, e.g. hanging a program in an infinite loop, division by zero and many more. A more appropriate exclamation (yet arguably not of practical relevance) about typing has been stated by S. Kahrs: *"Well-going programs can be typed"* [32].

8.1.3 Combined Static and Dynamic Typing

Some programming languages provide reflection frameworks or meta programming, enabling a program to reason about types, values and methods at run time. In Java e.g. the reflection API provides the ability to invoke methods, read object attributes and the type name of an object by invoking reflection methods. This makes it necessary to keep the type information of the objects at run time.Arguably, for *late binding* [5], a common feature of most object oriented languages, the objects need to be annotated with their type, as it is necessary to dispatch method invocations at run time according to the proper subtype an object may have.

Combined statically and dynamically typed languages as Java, C++ and C-Sharp have in common with statically typed programming languages, that ill typed programs are detected at compile time. With dynamically typed languages they have in common, that at rum time the types of the values is checked when late binding is necessary. The kind of check performed hence may never fail, it is a mere decision process which concrete implementation of a method has to be called in the given context.

The advantage of late binding is higher extensibility of the code applying late binding, as further object types unknown at implementation time of the code in question will be usable as long as they fulfill the necessary interfaces. This advantage has not to be *paid* by sacrificing static type safeness, as

the dynamic check may never fail at run time. The disadvantage of late binding is the higher overhead at run time.

Further dynamic features as e.g. reflection mostly share the same advantages and disadvantages as late binding—they are defined in a statically typed context, using statically typed abstractions of the language features. In the context of the reflection, the invocation may fail, as any dynamic programming may fail due to type errors at run time, but in the context of the hosting language, the failure is handled in a type safe way—e.g. in Java with appropriate exceptions.

8.1.4 From Typed Programming languages to typed Query Languages

In database theory querying is often formalized in an abstract manner as algebra of **selection**, **join**, and **projection**. [4] Less formally, querying—and hence a query language—is a composition of selecting some instances of a set of given data, further combining or filtering of potential instances by joining selections, and projecting intended parts of the selected data instances to result instances (maybe constructing new content for the result instances).

Types, as approximations of values, can be of potential use for error detection and prevention in query languages as well as it is of use for programming languages. The use of types in this context can also be decomposed as the principles of queries:

Selecting of some data instances of which a common type will be assumed has to be done in a way *compatible* with the data's type. A selection can hence be given a type. A common approach here [63][29] is to consider the type of the selection construct to be a type representing all data instances that the selection may select from the set of all possible data instances in the data formalism. A reasonable information about a selection under the condition of a given data type for the queried data could hence be "may this selection yield *any* result?"

Joining of different selections is often involved with sharing multiple occurrences of the same variable, hence comparing somehow the values of selected instances of multiple selections. This may be done e.g. using equivalence relations (for example "are there equivalent values in the selected instances of two selections?") or using any relation and operation on values (for example "is the sum of some values greater than a given bound?"). The idea of typing variables, functions and operations in general purpose programming languages for error detection or prevention is applicable in the same way here.

Projecting parts of the selected and 'joined' instances and construction of results yield new data instances that form a set—the answer set—and hence a type. A typed query program could be annotated with an intended result type, an ill typed query would hence be a query that *may* produce invalid results with respect to the given result type.

A query language with the presented typing philosophy is of practical relevance. Assume e.g. a query program in the context of a data warehouse: such queries tend to be long running processes due to the sheer amount of data, yet the data warehouse schema is often fairly simple (compared to the data complexity). If the query has an ill typed selection, it can quickly be rejected reducing development costs. The type tests of joins may hint potential run tie problems, e.g. due to type incompatible function application, it may further reduce queries that never yield results. As an example for the usefulness of type checking of projections and constructions, assume e.g. a query producing an HTML document as result: the intended result type could be given as HTML, a query that *may* produce invalid HTML would hence be ill typed.

8.2 Type Systems for Xcerpt

The query language Xcerpt exists as an untyped query language for the Web and the Semantic Web, yet different approaches for extending Xcerpt with types have been proposed.

8.2.1 XcerptT—Descriptive Typing for Xcerpt

XcerptT [14] [65] is a type system for a substantial fragment of Xcerpt. The type system is called a *descriptive type system*, i.e. a typing approximates the semantics of a program (in an untyped programming language). This means finding an approximation of the semantics of the given program. The "counterpart" of descriptive typing in typing terminology is *prescriptive typing*. In prescriptive typing the types are usually defined aligned to the semantics of the language, type checking is then not related to the language's semantic anymore. In our case, for a given Xcerpt program and a type of data (i.e. the set of data objects to which the program may be applied) the type system provides a type of the program results (i.e. a super-set of the set of the program results). This is type inference; if a type of expected results is given then type checking can be performed by checking if the obtained type of results is a subset of the the given one.

The given types for results and for the data queried (and in the end also the internal representation of types) is a regular tree language. It is also possible to use DTDs as type definitions.

Two different algorithms are presented in the in the context of XcerptT. The first one has polynomial complexity and is based on a slight restriction of regular tree grammars called *'Proper Type Definitions'*. Proper type definitions are regular tree grammars where there are not two distinct types in the regular expression content models, that model elements with identical label. The second algorithm handles arbitrary type definitions, but has exponential worst case complexity.

In contrast to the approach presented in this deliverable, the XcerptT approach always calculates the types as precisely as possible with respect to the expressiveness of the given type declaration formalism (i.e. proper type definitions or arbitrary regular tree grammars). The approach in this deliverable is to use unrestricted type declaration formalisms and to retain acceptable complexity—atleast for ordered queries and ordered type declarations[1]—by relaxing the type checking and inferences precision. Another difference of the type checking in XcerptT and this deliverable is the use of unordered content models: the modelling approach for unordered content models in XcerptT is the co called *multiplicity list*, as also presented in section 7.1.2.1. Multiplicity lists have been shown to be closed under intersection, which is arguably enough expressiveness for type checking in most situations, but they are not closed under union, which is arguably desirable for schema languages, e.g. for the creation of new schemata by uniting two given schemata.

8.2.2 Prescriptive typing: from CLP to Xcerpt

In the work of Francois Fages and Emmanuel Coquery [21], the adaption of a prescriptive type system for Constraint Logic Programming (i.e. CLP) to Xcerpt is presented. The type system for CLP is based on types with a sub-typing relation forming a lattice over the types with a most general type at the top and an empty type at the bottom. As CLP is based on a normal term concept, where terms and function symbols with a given label have fixed arity, the adaption to Xcerpt comprises an extension for the treatment of semi structured data. So, opposed to typing of CLP programs, where language constructs may be typed with one type along a path in the sub type relation lattice, language constructs

[1]For unordered type declarations the approach of type checking using R_2G_2 in this deliverable has exponential complexity.

in Xcerpt may be typed with various types at the same time. This is the consequence of having repetition constructs, optionality and disjunctions in type declarations based on regular tree grammars.

In contrast to the XcerptT of section 8.2.1 approach, this approach is able to treat whole programs with rules and chaining of them, as this property has already been inherited from the CLP typing. The specification of type checking in the adaption of CLP type checking to Xcerpt type checking is directly based on decomposition of regular expressions. This distinguishes the work from the approach chosen in this deliverable, where type checking is made based in an automaton model for ordered data and queries. Unordered type declarations are not considered in the CLP to Xcerpt typing adaption, however the problem itself is approached, as unordered query patterns can be type checked against)ordered) type declarations, which is comparable. The solution of type checking of unordered queries is expressed in the spirit of a declarative semantics of unordered types, where typing is valid, if there is an instance of the type of which an arbitrary permutation can be matched by the query pattern.

8.2.3 Typing with R_2G_2—Differences to the Former Approaches

The R_2G_2–based typing approach for Xcerpt[2] as presented in the remainder of this section, has similarities as well as differences to the formerly presented two approaches.

A **common feature** of all approaches is relying on **regular tree languages for ordered type declarations**. The typing algorithms however are expressed differently: while XcerpT and the prescriptive typing of E. Coquery directly rely on the use of regular expressions and regular grammars, the approach presented here is based on regular tree automata. Using directly **grammars and regular expressions in the formalization of type checking has the advantage of being simple** and 'inexpensive in formal definitions'—no intermediate formalism that is not visible to the user is necessary. However, **when it comes to implementation, the automaton approach is arguably an advantage**, as breadth and depth are treated in similar ways—both are conceptually just traversal of transitions: The use of automata for processing of the word problem is a standard method known to virtually any computer scientist and many programmers, extending finite automata techniques to tree automata is a well known technique and the use of the hyper graph based automata as presented in chapter 6 arguably makes the 'knowledge transition' from words to trees rather smooth. When it comes to type checking, the approach presented here is considered as an extension of the word problem, i.e. query language expressions are considered as words that are extended by constructs of incertitude, variables and incompleteness—hence type checking is boiled down to a *beefed-up* word problem test with automata.

The treatment of **unordered data models**, as presented in chapter 7, is **different in all three approaches**: while XcerptT follows the simple and pragmatic approach of using **multiplicity lists**, the prescriptive typing approach based on the adaption of CLP typing has **no modelling support** for unordered data models and the approach presented here relies on **unordered interpretations of regular expressions**. The use of multiplicity lists is computationally desirable, as membership test and relevant set operations (i.e. subset test, intersection, emptiness test) can be obtained in polynomial time. The downside, is the restricted expressiveness of the approach, some constraints on languages expressible with regular expressions are not expressible using multiplicity lists, possibly forcing the user in practice in using ordered data models where unordered models would be more appropriate. The use of unordered interpretations of regular expressions pragmatically overcomes the modelling restrictions of multiplicity

[2] Note, that the typing approach for Xcerpt can easily be adapted to other Web query languages as e.g. XQuery—Web query languages in general follow an approach of having selection, projection and construction. These features can also be found in Xcerpt. Adaption to other query languages to the typing process could e.g. in the simplest form be achieved by seeing Xcerpt as an intermediate language to which other query languages may be translated (Consider the work in [35] as an example of such a translation.) and on which type checking is performed.

lists, however at a super-polynomial cost. In experiments the use of constraint solving techniques has shown good results and always fast solutions of the relevant set operations, it has not been possible to find regular expressions modelling problems to fall in the so called *thick tail*, the class of problems that are indeed hard to solve with the given constraint solver.[3]

The approach presented in this deliverable extends the former approaches further on by a modelling technique for graph shaped data and for serialisations of graph shaped data by means of spanning trees and typed references. This feature leads indeed to almost no change in to the operational semantics of validation or type checking—no more than cycle detection and type compatibility in the cyclic case is needed. However it is considered to be a relevant extension of current schema languages, motivated by the availability of typed references in general purpose programming languages.

8.2.4 The Syntax of R_2G_2 typed Xcerpt

A typed Xcerpt program is an Xcerpt program, where every language term **may** be typed. The "may" implies, that any Xcerpt program of the current untyped syntax is also a valid program in typed Xcerpt syntax. Programs hence to not have to be completely type annotated to benefit of many type based services. The rational behind this is, that the programmer should not be forced into annotating all of the program with type information, when he does not like to, types can be inferred out of partial type annotation. A programmer may hence be willing to annotate the type of an input resource or of the output resource, as he wants to ensure valid input or output.

8.2.4.1 Typed Data Terms

Apart of Xcerpt programs, data terms may be type annotated, hence they can be validated using the type checking algorithm. Data terms are structurally the base of query and construct terms, hence of Xcerpt.

data-term := (oid "@")? *ns-label list* (*type-annotation*)?
type-annotation := "^^" (typename | *type-disjunction*)
type-disjunction := "(" (typename | typeterm) ("|" (typename | typeterm))* ")"
ns-label := (*ns-prefix* ":")? label
ns-prefix := label | '"' iri '"'
list := *ordered-list* | *unordered-list*
ordered-list := "[" *data-subterms*? "]"
unordered-list := "{" *data-subterms*? "}"
data-subterms := *data-subterm* ("," *data-subterm*)*
data-subterm := *data-term* | '"' string '"' | number | "^" oid

Consider example 8.1 for type declaration in R_2G_2 syntax along with the (totally) type annotated data term:

In example 8.1 only type names are used for annotation. If e.g. the element of type `City` would have been annotated with a type term, it would be as follows:

```
city[ "Munich"^^String ]^^(city[ String])
```

[3]A finite domain constraint solver from GNU Prolog has been used. Even if the problem is not a finite domain problem, there is always either a solution within a finite bound or no solution, this is implied by the fact, that Presburger arithmetic formulae are decidable. It is however an open question how to statically obtain the upper bound for the solution space. No further investigation in this direction has been made along this deliverable, as solvers for Presburger arithmetic constraints exist, it may however be research of interest for the finite domain solving community.

```
type AddressBook=addresses{Address*};
type Address=addr{Name, Street,
                  Nr, ZIP?, City? };
type Name=name[String];
type Street=street[String ];
type City=city[String];
type Nr=nr[Integer];
type ZIP=zip-code[Integer];
```

```
addresses{
 addr{
  name[
   "Sacha Berger"^^String
  ]^^Name,
  street[
   "Oettingen"^^String
  ]^^Street,
  city[
   "Munich"^^String
  ]^^City,
  nr[ 68^^Integer ]^^Nr,
  zip[ 80639^^Integer ]^^ZIP
 }^^Address
}^^AddressBook
```

Figure 8.1: An example presenting a valid Xcerpt term with type annotation along with the corresponding type annotation.

8.2.4.2 Typed Xcerpt Construct Terms

Xcerpt construct terms extend data terms by (1) variables, (2) grouping constructs (e.g. `all`) and (3) default settings for optional parts.

Variables are type annotated as if they would be data terms. For variables occurring in the `group-by` clause of the grouping constructs, the given type **may** be a primitive data type implementing order, hence being informative to the ordering algorithm of the grouping of constructed results.

Grouping constructs like `all` or `some` are not type annotated, as they represent sequences of elements, and the type annotation is inherently conceived for elements. However, the contained term can be annotated, as well as the variables occurring in the (optional) `group-by` clause.

Optional parts in Xcerpt construct terms denote parts, that are only to be constructed, if there are variable bindings for the scoped optional variable. As optional may span sequences of terms, the optional part is not type annotated, neither the (optional) `with-default` clause.

```
addresses[ all ( addr{ var N^^Name
                     , var C^^City
                     , optional (var Z^^ZIP)
                       with-default ( zip[0^^Integer]^^ZIP )
                     , var NR^^Nr
                     , var S^^Street }^^Address
               ) ordered-by (var N^^String)
         ]^^AddressBook
```

8.2.4.3 Typed Xcerpt Query Terms

Query terms arguably are extensions of Construct terms, in the sense that they have constructs for (1) incertitude and (2) negation, they are restrictions of them, in the sense that they have no (1) grouping constructs and no (2) `with-default` clause for variables tagged as optional variables.

Incertitude in order or breadth s given in Xcerpt by different brace or bracket styles in terms, they have no influence on type annotation of terms.

Incertitude in depth—the `desc` construct—is type annotated as a term. As it contains itself a term and hence the type annotation lexically looks like a type annotation of a type annotation, the *descendant tagged* term has to be set in parenthesis, if the descendant construct is to be typed. The query term `desc a[[]]` e.g. may be used to query hyperlinks at arbitrary depth in an HTML document. For it's typed variant `desc( a[[]]^^A)^^A` the type `A` is considered to be the type of hyperlinks.

Negation in Xcerpt is expressed using the `not` or `without` constructs. Their type annotation paradigm is the same as for the descendant construct.

Optional query parts may also span sequences and are hence not type annotated.

8.2.4.4 Typed Xcerpt Queries

Queries are query conjunctions or disjunctions of queries or query terms. Tey are term structured themselves, they can be type annotated the same way as other terms.

```
and( html[[]]^^HTML ,
     or( var X^^HTML,
         var Y^^HTML )^^HTML
   )^^HTML
```

8.2.4.5 Typed Xcerpt Rules

A Rule is typed (implicitly, i.e. without special syntactic construct) by the type of the query and the type of the construct term. A rule is hence not type annotated.

8.2.4.6 Typed Xcerpt Programs

An Xcerpt program per se has no type. Each rule of a program is typed, this also holds for goal rules. The type of a program can be seen as the disjunction of the types of all goals in the program. A program is hence not type annotated.

8.3 Type checking and Type Inference for R_2G_2 Typed Xcerpt Programs

Typing Rules for Query Terms The following rules describe the algorithm used to infer total type annotation for a (possibly partly type annotated) query term. The algorithm is a function which returns the possible types out of a given set of types for a query term under a given automaton A. The automaton itself is constructed out of R_2G_2 declarations as presented in section 6.3.

As a function, the algorithm's signature is denoted as $\Gamma, A|t, \tau \mapsto \tau'$, where t is an Xcerpt program term, i.e. a program, a rule or a part of a rule. A is the automaton representing the types declared to be used in the program and is constant in the whole context of the typing algorithm application. By Γ a global variable environment used to represent types of variables is denoted. It consists of tuples (v, T) where v is the variable (or it's name) and T the type of v.

Depending on the part of the Xcerpt program to be typed, τ, resp. τ' is of different kind. In general, for Xcerpt query or construct terms, τ is a set of transitions as defined in A, more precisely in A_Δ. As the type of a rule is given by the type of it's query part and it's construct part, the types for rules have also to be of *two-folded* nature—the type of a rule is a tuple of sets of transitions representing the types for the input and for the output.[4] For the type inference for sequences of query or construct terms, τ is a set of states as found in A, more precisely in A_S, and for sequences of construct terms in the context of a grouping construct (i.e. the `all` construct), τ is a set of tuples of states as found in A, more precisely in A_S.

Note, that the functions for typing terms and those for typing sequences (ordered and unordered) will rely recursively and interlaced on each other.

8.3.1 Paying Attention to Type Annotation in Queries

The typing algorithm as presented does not forcibly need type annotated programs, yet in can consider, even incomplete, type annotation. When type annotation is available, it is propagated to the set of assumptions about a program segment. In the case of a query term this means, that only types inferred for that term that have non empty intersection with the type annotation are considered to be possible types. The rational behind this is, that two types with non empty intersection have common members, only such members are arguably candidates for a *valid* query evaluation.

By $\overline{T}$ the user provided type annotation, i.e. the transitions realizing the type annotation in the automaton A, is denoted. The types of T and $\overline{T}$ get filtered to the types of T that have non empty intersection with a member of $\overline{T}$, the resulting set T'' is then used for the actual typing t. The result of this typing is them also the result of the typing of the type annotated term $t : \overline{T}$. The set T'' relies on checking the non empty intersection of two languages represented by two types, $\mathcal{L}(\overline{\tau}) \cap \mathcal{L}(\tau) \neq \emptyset$. This can be calculated with the algorithms for intersection and emptiness check on automata as presented in sections 6.4.1 and 6.4.2 applied on two variations of A, varying in the starting transitions—set to $\overline{\tau}$, resp. τ.

$$\frac{\begin{array}{c} T'' = \{\ \tau | \tau \in T \wedge \overline{\tau} \in \overline{T} \wedge \mathcal{L}(\overline{\tau}) \cap \mathcal{L}(\tau) \neq \emptyset\ \} \\ \Gamma, A \mid t, T'' \ \mapsto\ T''' \end{array}}{\Gamma, A \mid t : \overline{T}, T \ \mapsto\ T'''} \ (\text{QUERYTYPEANNOTATION})$$

8.3.2 Typing Ordered Queries

The typing of ordered and unordered sequences of terms is inherently different—ordered data is typed based on the use of automata as presented in chapter 6, while unordered data is typed on the basis of counting constraints as presented in chapter 7.

8.3.2.1 Ordered Total Query Patterns

When the typing function is applied to a query term $l[\ cnt\ ]$ with ordered, total sequence of sub terms and the type disjunction T consisting of a set of transitions from A, the following rule applies:

[4] Note, that a finer type variation would use a set of transition or type tuples to relate more precisely each input type to an output type. pros and cons of such a rule type has not been analysed along this deliverable.

$$\frac{\begin{array}{c} T' = \{\ (s,l,c,e) | (s,l,c,e) \in T \wedge c \in S'\ \} \\ S = \{\ c | (_,l,c,_) \in T\ \} \\ \Gamma,A \mid [\ cnt\ ],S \ \mapsto\ S' \end{array}}{\Gamma,A \mid l[\ cnt\ ],T \ \mapsto\ T'} \quad \text{(TotOrdQueryTerm)}$$

The result of the application of this rule is the set of transitions $T' \subseteq T$. T' consists only of transitions with label l—as found on the query term to type, and with a content start state against which the sequence of sub terms can be typed. This is achieved by passing the content start states S of the transitions with matching label l to the typing of ordered total sequences with the content sequence *cnt* of the query term, and using the result S' in the restriction of T to T'.

When the typing function for ordered sequences is applied to an ordered, total empty sequence with a set of states as input, the following rule applies:

$$\frac{S' = \{\ s | s \in S \wedge \exists s' \in \{s\} \cup \varepsilon\text{-reachable-states}(s,A)\ .\ s' \in F_A\ \}}{\Gamma,A \mid [\,],S \ \mapsto\ S'} \quad \text{(TotOrdEmptyList)}$$

The set $S' \subseteq S$ is generated by checking the input states if S for their membership in F_A, the set of final states of A. Note, that it is common to all the rules, except the `desc` rule, that they restrict the set of types used as input of the check. As the automata may contain ε-transitions, it is also necessary to traverse all possible ε-transitions from the states in S that are not final, as they may lead to final states. The ε-reachable-states(s,A) function returns a subset of A_Σ, such that each state of the subset can be reached from $s \in A_\Sigma$ when following paths starting at s made up only of ε-transitions. The rule has no rule application above the line, is hence not further recursively applying typing.

Type checking non-empty sequences is achieved in a recursive *divide-and-conquer* way, where the first element of the sequence is checked as well as the rest of the sequence. The typing of the sequence $[\ n_1, n_2, \ldots, n_m)\ ]$ receives the set of states S as input and returns $S' \subseteq S$. The transitions T for checking the head n_1 of the sequence are those that have a state of S as start state, or that have a start state that can be reached along ε-transitions from a state found in S. Further on the transitions in T need to have an (horizontal) end state in $\vec{S}'$, the set of states that an application of the typing function to the tail of the sequence returns. This application receives $\vec{S}$, the set of (horizontal) end states of T', the result of typing the first element of the sequence.

$$\frac{\begin{array}{c} S' = \{\ s | s \in S \wedge \exists t \in T\ .\ t \in \varepsilon\text{-reachable-transitions}(s,A)\ \} \\ \vec{S} = \{\ e | (_,_,_,e) \in T'\ \} \\ T = \{\ (s',l,c,e) | (s',l,c,e) \in \varepsilon\text{-reachable-transitions}(s,A) \wedge s \in S \wedge e \in \vec{S}'\ \} \\ \Gamma,A \mid n_1,T \ \mapsto\ T' \qquad A \mid [\ n_2,\ldots,n_m\ ],\vec{S} \ \mapsto\ \vec{S}' \end{array}}{\Gamma,A \mid [\ n_1,n_2,\ldots,n_m)\ ],S \ \mapsto\ S'} \quad \text{(TotOrdList)}$$

At a first glance, operationally, this looks like a hopelessly recursive dependency with need for fix-point approximation. Unfortunately, the types of elements of a sequence are not independent, this is due to the type of the sequence. Hence, checking the type of the first element influences the type of the rest of the sequence and the type of the rest of the sequence influences the type of the head. Fortunately it turns out, that from an operational point of not more than one refinement of the head is needed: The key here is, that the typing algorithm refines a given type disjunction (in form of a set of types) by reducing the set to possible types. After performing this reduction on the head element of the sequence, the tail of the sequence is typed with the outcome of the head typing. The tail then may reduce the input set of possible (type) states. The typing of the head can now be refined to just use the types represented by

transitions with (horizontal) end states that are contained in the result of the typing of the tail. Retyping the tail is now not necessary, as it will yield exactly the same set of states. Arguably, the second typing application to the head is not necessary, as it is already confirmed, that the input types used will succeed, a kind of type propagation would be enough, but for the sake of briefness, the typing itself is used for the type propagation.

8.3.2.2 Ordered Partial Query Patterns

Typing of an ordered partial query pattern $l[[\ cnt\ ]]$ is very similar to the typing of an ordered total query pattern, where the only distinction is the application of the typing rule for ordered partial sequences for the sequence of sub terms.

$$\frac{\begin{array}{c} T' = \{\ (s,l,c,e) | (s,l,c,e) \in T \wedge c \in S'\ \} \\ S = \{\ c | (_,l,c,_) \in T\ \} \\ \Gamma,A \mid [[\ cnt\ ]], S\ \mapsto\ S' \end{array}}{\Gamma,A \mid l[[\ cnt\ ]], T\ \mapsto\ T'} \quad \text{(PARTORDQUERYTERM)}$$

Typing of partial ordered empty sequences is similar to typing of total ordered empty sequences:

$$\frac{S' = \{\ s | s \in S \wedge \exists s' \in \{s\} \cup reachable-states(s,A)\ .\ s' \in A_F\ \}}{\Gamma,A \mid [[\ ]], S\ \mapsto\ S'} \quad \text{(PARTORDEMPTYLIST)}$$

The only difference is the extension of non final states—instead of just following paths made up of ε-transitions, now paths made up of ε-transitions and steps along horizontal transition steps of non-epsilon-transitions can be used to reach final states. This is given by the $reachable-states(s,A)$ function.

Unsurprisingly, the typing of (non-empty) ordered partial sequences is similar to the total case. The only difference is to not just consider transitions with start states found in S (possibly connected using ε-transitions) for the typing of the first sequence element, but any transition reachable by horizontal traversal starting at the states of S. The rational behind this is, that an incomplete pattern matches data terms where in between the matched (i.e. bound to query terms) sub terms arbitrary many other sub terms may occur. These sub terms, if valid with respect to the given type, would hence be accepted by corresponding automata transitions.

$$\frac{\begin{array}{c} S' = \{\ s | s \in S \wedge \exists t \in T\ .\ t \in reachable-transitions(s,A)\ \} \\ \vec{S} = \{\ e | (_,_,_,e) \in T'\ \} \\ T = \{\ (s',l,c,e) | (s',l,c,e) \in reachable-transitions(s,A) \wedge s \in S \wedge e \in \vec{S}'\ \} \\ \Gamma,A \mid n_1, T\ \mapsto\ T' \qquad \Gamma,A \mid [[\ n_2,\ldots,n_m\ ]], \vec{S}\ \mapsto\ \vec{S}' \end{array}}{\Gamma,A \mid [[\ n_1,n_2,\ldots,n_m\ ]], S\ \mapsto\ S'} \quad \text{(PARTORDLIST)}$$

8.3.2.3 Descendants

A descendant query term consists of a term n adorned with the `desc` keyword. The query semantics is, that it either matches a data term at the level of the term sequence that is to be simulated by the query term sequence containing the descendant query term, or any term containing at arbitrary nesting depth a term simulated by n. A typing strategy for this semantics is, that the descendant term is a placeholder for any term of not only types valid for typing n, but also of the types t that contain at arbitrary depth

types against which n may be typed. This is restricted by the fact, that the type t must also be valid with respect to the given position of the descendant term, must hence be in the set of the types used as input of a typing rule application on the descendant term. Note, that the descendant construct itself may not be type annotated by the user, yet the typing process may associate type information to it, valuable for query optimization.

The typing of a descendant term receives the set of transitions T as input and returns a restriction $T' \subseteq T$ such that each transition $t \in T'$ either is a valid transition for n, or a valid transition for n can be reached from the content model start state of t in breadth or depth. For the typing of n all transitions of T are used plus all transitions reachable in breadth or depth from the content model start states of the transitions in T.

$$\frac{\begin{array}{c}\vec{T} = T \cup \{\, t | t \in \mathsf{breadthAndDepthReachableTransitions}(c,A) \wedge (_,_,c,_) \in T \,\} \\ T' = \{\, t | t \in T \wedge (\, t \in \vec{T}' \vee \exists \vec{t}' \in \vec{T}'.(_,_,c,_) = t \wedge \vec{t}' \in \mathsf{breadthAndDepthReachableTransitions}(c,A) \,\} \\ \Gamma,A \mid n,\vec{T} \;\mapsto\; \vec{T}'\end{array}}{\Gamma,A \mid \mathtt{desc}\; n,T \;\mapsto\; T'} \quad \text{(DESCENDANT)}$$

8.3.2.4 Variables

The semantics of an unrestricted variable is to match any data term as long as this is consistent with the positional mapping of data terms to query terms in the simulation unification. From the typing view, positional information is considered at the level of sequences, resulting in a set of (positionally) appropriate types in T. As a consequence of no further restricting the matching, the result of the application of the variable typing function rule with a given set of transitions T is the unaltered set T.

$$\frac{(X,T_X) \in \Gamma \wedge \forall \tau_X \in T_X.\exists \tau \in T.(\tau_X) \cap (\tau) \neq \emptyset}{\Gamma,A \mid \mathtt{var}\; X,T \;\mapsto\; T_X} \quad \text{(QUERYVARIABLE)}$$

In addition to the structural properties of queries, variables only match data terms in a query in such a way, that multiple occurrences of the same variable match identical data terms (where 'identity' in this context is given, if terms bi-simulate using ground query term simulation). The same property holds for multiple instances of the same variable in a conjunction of query terms. A typing view to this property is, that the terms matched by different occurrences of the same variable are forcibly of the same type. Nevertheless, the variable occurrences may have different type annotations, as long as they have non empty intersection. The global environment Γ is used for this purpose. It ensures, that all types assigned to a variable have non empty intersection with at least one type of any other occurrence of the same variable. In implementations of the rule, this usually means, that independently of applying structural typing to the whole rule, the environment needs to be checked, possibly resulting in the necessity to repeat the typing phase, as the altered type of the variable may affect sibling element types or types of elements on the same path as the variable (and hence possibly the types in the whole structure). This process can again influence the variable types, requiring a fixpoint based iteration approach. As in worst case, each variable (not each occurrence) is typed with all possible types and in each iteration not more than one type in one variable gets eliminated until one variable reaches an empty (and hence invalid) state, the repetitions is in the order of the size of the automaton times the number of variables (which is in the order of the size of the query). As however such a strong relationship between the types of different occurrences of the same variable and of the structure are usually quite unlikely, it can be assumed, that not more than one iteration (or a fixed small number of iterations) is needed. A non-reached fixpoint after an iteration is e.g. worth a warning, that the query is structurally valid, however the satisfiability

of the different occurrences of a variable (the one necessary for not reaching the fixpoint) cannot be ensured, which means, that it is possible, that the query never matches any valid data.

Variables with variable restrictions are evaluated while querying by only binding data terms to the variable, that match a given query term n. They are hence evaluated like regular query terms, but in addition the matching data is bound to the given variable. As a consequence, for typing, the type of the variable is propagated from the typing of the restricting query term.

$$\frac{\Gamma,A \mid t,T \;\mapsto\; T' \qquad \Gamma,A \mid \texttt{var}\,X,T' \;\mapsto\; T''}{\Gamma,A \mid \texttt{var}\,X \rightarrow t,T \;\mapsto\; T''} \text{ (QUERYVARIABLERESTRICTED)}$$

8.3.3 Typing Unordered Queries

When typing unordered query terms, counting constraints are used to check the consistency of the language represented by the given types and the language of terms that the unordered (total or partial) sequence of query terms may match. This is achieved by checking the intersection of the two languages for emptiness.

The intersection of languages represented by counting constraints is constructed by building a conjunction of the counting constraints representing the two languages and adding additional constraints restricting different variables from the two languages representing multiplicities of equal symbols to be equal.

Note, that in the following set of rules, the grammar G, on which the automaton A is based, is also assumed to be globally available. It however has not been integrated in the rule signature, to reduce formal buzz in the cases of ordered content.

8.3.3.1 Unordered Total Query Terms

When the typing function is applied to a query term $l\{\ cnt\ \}$ with unordered, total sequence of sub terms and the type disjunction T, the following rule applies:

$$\frac{\begin{array}{c} T' = \{\ (s,l,c,e) \mid (s,l,c,e) \in T \wedge c \in S'\ \} \\ S = \{\ c \mid (_,l,c,_) \in T\ \} \\ \mathcal{T} = \{\ (c,re) \mid \tau \in T \wedge \tau_{trans} = (_,l,c,_) \wedge \text{``}\texttt{element}\,\tau_{name}{=}l\langle re\rangle'' \in G\ \} \\ \Gamma,A \mid \{\ cnt\ \},\mathcal{T} \;\mapsto\; S' \end{array}}{\Gamma,A \mid l\{\ cnt\ \},T \;\mapsto\; T'} \text{ (TOTUNORDQUERYTERM)}$$

The term is typed with $T' \subseteq T$, which is obtained very much in the way of typing total ordered query terms. The only addition is, that typing unordered sequences is done with a set of types represented as tuples of a state and a regular expression, where the state is the start state of the automaton part implementing the regular expression—the unordered sequence typing rule's signature is the rational for that. The regular expressions of the type declarations of the types in T have hence to be accessible. This is not difficult, as the type name of a term is available in that context as well as the grammar, it is hence just a look-up of the typing rule with the given type name and extraction of the regular expression from the right hand side type term.[5]

[5]The look up of the grammar rule is formalized as a pattern matching with "$\texttt{element}\,\tau_{name}{=}l\langle re\rangle''$ where $\langle$ and $\rangle$ depict arbitrary braces, because it is irrelevant if the grammar rule models an ordered or an unordered data model.

Typing an unordered sequence of query terms relies on the use of counting constraint based type annotations, as mentioned at the beginning of the section. As the counting constraint language representation, as introduced in chapter 7, are generated out of regular expressions, the signature of the typing rule for unordered sequences needs a set of regular expressions as given type formalism. The regular expressions are passed in along with the automaton state implementing the start state of the expression, as the typing of terms—the typing rule from which sequence typing is applied—expects those states as result of the recursive typing process. The starting state of the automata implementation of the regular expression is hence just needed to relate the regular expression to the type occurrence it originates from.

$$\frac{\begin{array}{c}\tau_{i_j} \in T_i' \wedge re = (\tau_{1_{1_{name}}}|\cdots|\tau_{1_{j_{name}}}|\cdots),\cdots,(\tau_{m_{1_{name}}}|\cdots|\tau_{m_{k_{name}}}|\cdots) \\ S' = \{\ s|(s,\mathfrak{re}) \in \mathcal{T} \wedge \mathsf{countingConstraints}(re) \cap \mathsf{countingConstraints}(\mathfrak{re}) \neq \emptyset\ \} \\ T = \{\ \tau|\tau_{trans} \in \varepsilon{-}\mathsf{reachable}-\mathsf{ransitions}(s) \wedge (s,_) \in \mathcal{T}\ \} \\ \Gamma,A \mid n_1,T\ \mapsto\ T_1' \qquad \cdots \qquad n_m,T\ \mapsto\ T_m'\end{array}}{\Gamma,A \mid \{\ n_1,\ldots,n_m\ \},\mathcal{T}\ \mapsto\ S'}\ (\textsc{TotUnordSeq})$$

The rule is applied to a sequence $n_1,\ldots,n_m$ of query terms and a set of automaton state / type name tuples $\mathcal{T}$. The automata states in $\mathcal{T}$ denote starting states of the automata components implementing the corresponding regular expressions. From those states, all reachable transitions are gathered using the $\varepsilon{-}\mathsf{reachableTransitions}$ function, the resulting set T of transitions is use for typing the nodes n_i. The nodes are checked against all types, as they may be at any position in the query term, hence matching to any of the types in T. The results of the node typing T_i' are then used to construct a regular expression, by using the type names $\tau_{i_{j_{name}}}$ of the types T_i', building a group of options $(\ldots|\tau_{i_{j_{name}}}|\ldots)$ for each type T_i' and sequencing the option groups. The rational behind this regular expression is, that it represents a reasonable type approximation of sequences matched by the given query term sequence. For this regular expression (called re), a counting constraint is then generated and checked against non-emptiness of the intersection with the regular expressions $\mathfrak{re}$ found in tuples in $\mathcal{T}$, the set of input types. For non-empty intersections, the state corresponding to the regular expression $\mathfrak{re}$ is then finally included in the result set S'.

8.3.3.2 Unordered Partial Query Term

The typing of partial unordered query terms is almost identical to the typing of total unordered query terms, the mere difference is the application of the typing rule for partial unordered sequences instead of the rule for total unordered sequences:

$$\frac{\begin{array}{c}T' = \{\ (s,l,c,e)|(s,l,c,e) \in T \wedge c \in S'\ \} \\ S = \{\ c|(_,l,c,_) \in T\ \} \\ \mathcal{T} = \{\ (c,re)|\tau \in T \wedge \tau_{trans} = (_,l,c,_) \wedge \text{“}\mathtt{element}\ \tau_{name}\ =\ l\langle re\rangle\text{”} \in G\ \} \\ \Gamma,A \mid \{\{\ cnt\ \}\},\mathcal{T}\ \mapsto\ S'\end{array}}{\Gamma,A \mid l\{\{\ cnt\ \}\},T\ \mapsto\ T'}$$

(PartUnordQueryTerm)

The typing of partial unordered sequences is very similar to typing of of total unordered sequences: Instead of calculating the intersection of the two languages, another operator is used—it is called extended intersection and defined now:

While normal language intersection of counting constraint languages are constructed by conjunction of the two constraint sets and adding equality constraints for the variables representing the same

symbol's multiplicities in both languages, the equality is exchanged by less-equal ($\leq$), such that the symbol's multiplicities on the side of the partial query term's counting constraint language definition may be smaller, than on the side of the given type. Symbol multiplicities of one language not matched by variables on the other side get zero multiplicity on the *other* side, like for the intersection case, but again not with a less-equal relation. The *'inequality'* relation reflects the fact, that more symbols in valid data instances may occur than in a query pattern they match in, but not the other way round.

$$\frac{\begin{array}{c}\tau_{i_j} \in T_i' \wedge re = (\tau 1_{1_{name}} | \cdots | \tau 1_{j_{name}} | \cdots), \cdots, (\tau m_{1_{name}} | \cdots | \tau m_{j_{name}} | \cdots) \\ S' = \{ \ s | (s, \mathfrak{re}) \in \mathcal{T} \wedge \mathsf{countingConstraints}(re) \cap^{\leq} \mathsf{cc}(\mathfrak{re}) \neq \emptyset \ \} \\ T = \{ \ \tau | \tau_{trans} \in \varepsilon{-}\mathsf{reachableTransitions}(s) \wedge (s, _) \in \mathcal{T} \ \} \\ \Gamma, A \mid n_1, T \ \mapsto \ T_1' \qquad \cdots \qquad \Gamma, A \mid n_m, T \ \mapsto \ T_m' \end{array}}{\Gamma, A \mid \{\{ \ n_1, \ldots, n_m \ \}\}, \mathcal{T} \ \mapsto \ S'} \ (\textsc{PartUnordList})$$

8.3.4 Typing of Construct Terms

Typing of Xcerpt construct terms in special and data in general is comparable to typing of queries. Similarities are, that

- structural properties of the construction is checked for validity based on the automaton model used for internal type representation.
- the typing is based on the semantics of the construct terms, which manifests itself in a set of data terms valid with respect to the given type.

Differences in the typing of construct terms to the typing of query terms are, that

- construct terms have no incompleteness.
- construct terms have grouping or repetition components.
- no typing for variables is deduced, the typing deduced in query parts is *used*.
- while query terms semantically represent sets of data terms[6] and a well typed data term is one, with non empty intersection of this set and the set of data terms represented by the type, a well typed construct term in contrast represents a set of data terms[7] that must be a subset of the given type, as otherwise the construct term **may** yield invalid results.

Typing of construct terms is presented mostly with the same formalism as typing for queries. A small formal extension is used on the rules for typing the sequence of nodes in the scope of a grouping construct like `all`—the signature of such rules is $A \mid lst, S \ \mapsto \ SE$, where the only difference to the signature of former rules is, that the result of typing is a set of tuples of states $SE \subseteq S \times S$ where A is an automaton and S the set of states occurring in A as presented formerly for query terms.

[6] A query term can be seen as a representation of a set of data terms—of the data terms that can be matched by the query term.

[7] A construct term can be seen as a representation of a set of data terms—of the data terms that can be constructed by the construct term for arbitrary well typed substitution sets as input. A well typed substitution set is one, where variables are always substituted by data terms valid with respect to the type of the variable.

8.3.4.1 Ordered Total Construct Terms

Typing a construct term with label l, formalized in the rule ORDCONSTRUCTTERM, requires l to be the label of at least one transition in the set of transitions/types T. Further on, the content model *cnt* of the term has to be valid with respect to the content model of one of the possible types. The result of application of the typing rule is the set $T' \subseteq T$, consisting of all types for which the construct term validation passes. Note, that the rule is identical to the typing rule for a totally ordered query term. Indeed, such a query term (without any incompleteness like descendants or incomplete subterms at arbitrary depth and without variables, semantically represent the same set of data terms as such a construct term—the represent a set consisting of exactly one data term syntactically equivalent to those query and construct terms! This property is called *referential transparency*[11]. Referential transparency means, that the same term within a given context has the same meaning (or represents the same data).

$$\frac{\begin{array}{c} T' = \{ \ (s,l,c,e) | (s,l,c,e) \in T \wedge c \in S' \ \} \\ S = \{ \ c | (_,l,c,_) \in T \ \} \\ \Gamma,A \mid [\ cnt\ ],S \ \mapsto \ S' \end{array}}{\Gamma,A \mid l[\ cnt\ ],T \ \mapsto \ T'} \quad (\text{ORDCONSTRUCTTERM})$$

The ordered sequence of construct terms of a construct term are typed in a similar way as the ordered sequence of query terms—starting with the empty sequence, it can successfully be typed by a final state. The set of states used as input S is restricted to S', the final states of the automaton A (fond in F_A)

$$\frac{S' = \{ \ s | s \in S \wedge \exists s' \in \{s\} \cup \varepsilon\text{-reachable-states}(s,A) \ . \ s' \in F_A \ \}}{\Gamma,A \mid [\,],S \ \mapsto \ S'} \ (\text{ORDCONSTRUCTEMPTYLIST})$$

Type checking non-empty sequences of construct terms is similar to type checking of non-empty total sequences of query terms. It is achieved in a recursive *divide-and-conquer* way, where the first element of the sequence is checked as well as the rest of the sequence. The typing of the sequence $[\ n_1,n_2,\ldots,n_m\ ]$ receives the set of states S as input and returns $S' \subseteq S$. The transitions T for checking the head n_1 of the sequence are those that have a state of S as start state, or that have a start state that can be reached along ε-transitions from a state found in S. Further on the transitions in T need to have an (horizontal) end state in $\vec{S}'$, the set of states that an application of the typing function to the tail of the sequence returns. This application receives $\vec{S}$, the set of (horizontal) end states of T', the result of typing the first element of the sequence.

$$\frac{\begin{array}{c} S' = \{ \ s | s \in S \wedge \exists t \in T \ . \ t \in \varepsilon\text{-reachable-transitions}(s,A) \ \} \\ \vec{S} = \{ \ e | (_,_,_,e) \in T' \ \} \\ T = \{ \ (s',l,c,e) | (s',l,c,e) \in \varepsilon\text{-reachable-transitions}(s,A) \wedge s \in S \wedge e \in \vec{S}' \ \} \\ \Gamma,A \mid n_1,T \ \mapsto \ T' \qquad \Gamma,A \mid [\ n_2,\ldots,n_m\ ],\vec{S} \ \mapsto \ \vec{S}' \end{array}}{\Gamma,A \mid [\ n_1,n_2,\ldots,n_m\ ],S \ \mapsto \ S'} \ (\text{ORDCONSTRUCTLIST})$$

Variables are not restricted in construct terms in the sense, that multiple occurrences of the same variable in a construct term have no influence on each other, as they receive their bindings or typing from the query part of a rule—all occurrences of the same variable are typed equally in the query. However, it is important to ensure, that the type of the variable in the global environment does not exceed the type propagated to the typing procedure. A variable type in Γ violating this requirement indicates, that

substitutions for this variable may exist, that would result in invalid instances of the construct term. This requirement is considered by checking that each type associated to the given variable is contained in the set of types propagated to the construct variable rule application—hence the type for the variable X in Γ has to be a subset (or equal) to T. However, it is possible to restrict the type of the variable based in the global environment's type—if it is a real subset of T, it can be returned as the consequence of the rule application to the variable.

$$\frac{(X,T_X) \in \Gamma \wedge \forall \tau_X \in T_X.\exists \tau \in T.(\tau_X) \subseteq (\tau)}{\Gamma,A \mid \texttt{var}\ X,T \ \mapsto\ T_X} \ (\textsc{OrdConstructVariable})$$

The Xcerpt grouping construct `all`[8] is rather complex, as it involves three typing rules and s special case in the typing rule signature—the case, where application of the typing function returns a set of tuple of states. A grouping construct can be considered as a sequence of nodes in another sequence (hence the grouping construct is a sub-sequence of a sequence of nodes). The node sequence of the grouping construct is to be repeated arbitrary often, at least once. Hence, the content model of the construct term containing a grouping construct among it's child node sequence needs to support repetition appropriate to capture the repeatability property of the grouping construct. As typing in this deliverable is operationally based on automata, some means of repetition in automata has to be found for paths in automata appropriate for typing grouping constructs. Repetition is modelled by loops in automata. A grouping construct is hence valid with respect to a type or a state (remember, that sequences are typed using automata states instead of transitions), when there is a path from this state such that the whole grouping sequence can be validated along this path, and when the last state of this path is either as well the first state, or the first state is reachable via ε-edges from this last state. For this reason the typing of grouping sequences returns a set of tuples of states, consisting of the (start)state with which the grouping sequence is typed, as well as the end state. The end state is (1) used to check the cycle to the start state, and (2) used as start state for the validation of the following sibling nodes of the grouping construct. The start state is one, that has been reached while validating the sequence of sibling nodes to the left of the grouping construct.

The typing rule is not applied strictly to the grouping construct, but to the sequence consisting of the grouping construct as first element and the following siblings. Note, that the pattern this rule applies to, hooks in in the typing of construct term sequences in general.

The grouping construct `all` $(n_{1_1},\ldots,n_{1_p}),n_2,\ldots,n_m)$ is hence typed with the states S such that the sequence of nodes $(\ n_{1_1},\ldots,n_{1_p}\ )_{\texttt{all}}$, which are the nodes in the context of the grouping construct, can be typed using S. The result is the set of state tuples SE. The nodes following the grouping construct are typed with the set of nodes $\vec{S}$ where each $e \in \vec{S}$ comes from a tuple $(s,e) \in SE$, if s is ε-reachable from e.

$$\frac{\begin{array}{c}\vec{S} = \{\ \vec{e} | (s,e) \in SE \wedge \vec{e} \in \{e\} \cap \varepsilon\text{-reachable-states}(e,A)\ \} \\ S' = \{\ s | (s,e) \in SE \wedge (\{e\} \cup \varepsilon\text{-reachable-states}(e,A)) \cup \vec{S}'\ \} \\ \Gamma,A \mid (\ n_{1_1},\ldots,n_{1_p}\ )_{\texttt{all}},S \ \mapsto\ SE \qquad \Gamma,A \mid [\ n_2 \ldots,n_m\ ],\vec{S} \ \mapsto\ \vec{S}'\end{array}}{\Gamma,A \mid [\ \texttt{all}\ (n_{1_1},\ldots,n_{1_p}),n_2,\ldots,n_m)\ ],S \ \mapsto\ S'} \ (\textsc{OrdConstructGrouping})$$

When a grouping construct sequence is empty, the state its validation starts with is per definition also the state it ends with. Note, that an empty grouping construct sequence never occurs in real programs,

[8] Other grouping constructs like `some` have been kept out for the sake of brevity, their typing is not substantially different to the typing of `all`.

it is however a valid pattern in the recursive decomposition of a program when typing it—the empty grouping construct sequences the recursions base case.

$$\frac{SE = \{\ (s,s) | s \in S \}}{\Gamma, A \mid (\ )_{\texttt{all}}, S \ \mapsto \ SE} \quad (\textsc{OrdConstructGroupingList})$$

The typing of a non empty grouping construct sequence is similar to the typing of construct term sequences. The difference lies in the return type (the set of state tuples) and how it is constructed. The set of tuples consists of tuples, such that the tuples (s,e) consist of a state s with which the sequence is validated (this is identical to the typing of general sequences), and of the state e that is propagated from the validation of the tail and which is the end state of the validation of the sequence. In contrast to general construct term sequence validation, the end state of the validation does not have to be member of the set of final states of the automaton A, as possibly the grouping sequence is followed by sibling nodes in it's containing sequence of nodes.

$$\frac{\begin{array}{c} SE = \{\ (s,e) | s \in S \wedge (s',e) \in \vec{S}E \wedge \exists (s',l,c,e) \in T'\ .\ (s',l,c,e) \in \varepsilon\text{-reachable-transitions}(s,A)\ \} \\ \vec{S} = \{\ e | (_,_,_,e) \in T'\ \} \\ T = \{\ (s',l,c,e) | (s',l,c,e) \in \varepsilon\text{-reachable-transitions}(s,A) \wedge s \in S \wedge (\vec{s},e) \in \vec{S}E\ \} \\ \Gamma, A \mid n_1, T \ \mapsto \ T' \qquad \Gamma, A \mid (\ n_2, \ldots, n_m\ ], \vec{S} \ \mapsto \ \vec{S}E \end{array}}{\Gamma, A \mid (\ n_1, n_2, \ldots, n_m\ )_{\texttt{all}}, S \ \mapsto \ SE}$$

(OrdConstructGroupingList)

8.3.4.2 Unordered Construct Terms

Briefly, typing of unordered construct terms is similar to typing of unordered total query terms, with two differences: (1) an unordered construct term may contain grouping constructs that have to be treated, and (2) the calculated type when type checking of a construct term has to be a subset of the type used for annotation (in contrast to non-empty intersection between calculated and annotated type in the case of the query). The treatment of grouping constructs (exemplary shown on the Xcerpt grouping construct `all`) unfortunately introduces another modification of the rule signature: the rule for the abstract syntax component `all` receives a set of types as input and returns **one** regular expression.

$$\frac{\begin{array}{c} T' = \{\ (s,l,c,e) | (s,l,c,e) \in T \wedge c \in S'\ \} \\ S = \{\ c | (_,l,c,_) \in T\ \} \\ \mathcal{T} = \{\ (c,re) | \tau \in T \wedge \tau_{trans} = (_,l,c,_) \wedge \text{“}\texttt{element}\,\tau_{name}{=}l\langle re\rangle\text{''} \in G\ \} \\ \Gamma, A \mid \{\ cnt\ \}, \mathcal{T} \ \mapsto \ S' \end{array}}{\Gamma, A \mid l\{\ cnt\ \}, T \ \mapsto \ T'} \quad (\textsc{UnordConstructTerm})$$

At the abstract syntax level of the unordered construct term (not at the level of it's sequence of child terms), typing is **identical** to the typing of the unordered query term. It is however not so surprising when looking at what exactly happens: The term is typed with $T' \subseteq T$. Types of T with equal label l as the construct term in question are further considered for typing of the sequence of child terms. Recall, that my τ_{name} the type name, as used in the grammar, of a type or transition is meant. Respectively, τ_{trans} represents a transition for a type name.

$$\frac{\begin{array}{c} re = \mathsf{re\text{-}component}(n_1, T_1'), \cdots, \mathsf{re\text{-}component}(n_m, T_m') \\ S' = \{\ s | (s, \mathfrak{re}) \in \mathcal{T} \wedge \mathsf{countingConstraints}(re) \subseteq \mathsf{countingConstraints}(\mathfrak{re})\ \} \\ T = \{\ \tau | \tau_{trans} \in \varepsilon\text{-}\mathsf{reachable\text{-}transitions}(s) \wedge (s, _) \in \mathcal{T}\ \} \\ \Gamma, A \mid n_1, T \ \mapsto\ T_1' \qquad \cdots \qquad n_m, T \ \mapsto\ T_m' \end{array}}{\Gamma, A \mid \{\ n_1, \ldots, n_m\ \}, \mathcal{T} \ \mapsto\ S'}$$

(UNORDCONSTRUCTLIST)

Typing an unordered sequence of child nodes of a construct term is done by first typing all child nodes, then checking that the regular expression *re*, which gets constructed of the results of typing the child terms, models an unordered language that is a subset (or equal) to the language modelled by regular expressions provided in the set of tuples $\mathcal{T}$. States from the set of tuples $\mathcal{T}$ are returned as valid sequence types, only if they were associated to a regular expression modelling an unordered super-set language to *re*. The regular expression *re* is constructed by concatenating the results of applying the auxiliary function re-component to the consequences of typing all child nodes n_i. re-component is defined as follows:

$$\begin{array}{rcl} \mathsf{re\text{-}component}(l[\cdots], \{\tau_1, \ldots, \tau_m\}) & = & (\tau_{1_{name}} | \cdots | \tau_{m_{name}}) \\ \mathsf{re\text{-}component}(l\{\cdots\}, \{\tau_1, \ldots, \tau_m\}) & = & (\tau_{1_{name}} | \cdots | \tau_{m_{name}}) \\ \mathsf{re\text{-}component}(\mathtt{all}(\cdots), re) & = & re \end{array}$$

re-component returns the regular expression associated to the typing of an `all` construct or the disjunction of all types (more precisely all type names) returned as consequence of typing any other construct term.

$$\frac{\begin{array}{c} re = (\mathsf{re\text{-}component}(n_1, T_1''), \ldots, \mathsf{re\text{-}component}(n_m, T_m''))^+ \\ \Gamma, A \mid n_1 \vec{T} \ \mapsto\ \vec{T}_1'' \qquad \Gamma, A \mid n_m \vec{T} \ \mapsto\ \vec{T}_m'' \end{array}}{\Gamma, A \mid \mathtt{all}\ (n_1, \ldots, n_m), T \ \mapsto\ re} \text{ (UNORDCONSTRUCTGROUPING)}$$

Typing a sequence of terms enclosed by the grouping construct `all` is done by checking all those sub-terms and returning a regular expression constructed similar to the one used in the typing rule for typing sequences—the only difference is the appended "plus" to indicate, that a grouping construct produces arbitrary many (but at least one) instances of the type of the enclosed enclosed sequence of construct terms.

$$\frac{(X, T_X) \in \Gamma \wedge \forall \tau_X \in T_X. \exists \tau \in T. (\tau_X) \subseteq (\tau)}{\Gamma, A \mid \mathtt{var}\ X, T \ \mapsto\ T_X} \text{ (UNORDCONSTRUCTVARIABLE)}$$

Typing of variables in unordered construct terms is identical to the typing in ordered construct terms.

8.3.5 Typing Rules

The typing of a rule is done by connecting the typing of the query and the construct part. Connecting the types is mere propagation of the global environment Γ as obtained by the typing of the query to the typing of the construct part. Depending of the case, where typing of programs and rule chaining is considered, the type of the rule itself is relevant or not—a rule is seen like a transformation of data from the type of the query part to the type of the construct part.

$$\frac{\Gamma, A \mid c, T_c \ \mapsto\ T_c' \qquad \Gamma, A \mid q, T_q \ \mapsto\ T_q'}{\Gamma, A \mid c \leftarrow q, T_c \leftarrow T_q \ \mapsto\ T_c' \leftarrow T_q'} \text{ (RULE)}$$

8.3.6 Typing Programs

For rule based languages like Xcerpt, typing of a program arguably means first typing of all rules. However, rules interact by so called rule chaining, which means, that the query parts of a rule is evaluated against the consequences of all rules, i.e. against the data constructed by the other rules. Typing Xcerpt programs considering rule chaining has been conceived first in work by W. Drabent and A.Wilk [65]. Different aspects of rule typing have been considered in this work, due to the fact, that in some cases termination of the typing can not be ensured—if the consequence of the rules restricts the type of the queries, the restricted queries can again restrict the consequences of the construction and hence the query type may be further restricted. Approximations may be defined in such a way, that termination can be guaranteed, e.g. by recognizing cyclic type restriction and aborting after a fixed number of cycles.

8.3.7 Coverage of Current Xcerpt Constructs

The presented typing rules do not cover all of Xcerpt's current features. As Xcerpt is in active development, features currently come and go. The features typed in the preceding sections are either generally useful when typing query languages for the Web or are principally new to the way typing is done in e.g. programming languages. Features of Xcerpt currently available or under development, that have not been covered here are:

1. conjunctive and disjunctive queries
2. resource specifiers
3. conditions (a.k.a. `where`-clauses)
4. optional variable binding modifiers
5. query negation modifiers
6. function applications
7. rule chaining

Handling of conjunction and disjunction in queries (1) can be seen mostly as separate typing of the different components but sharing a common environment for the variables, where variables occurring in different conjuncts have to have non empty intersection and variables in different components of a disjunction get typed with the union of the types in both disjunctive components. Resource specifications (2) occur at the root level of a query. They by this do not have a logical impact on the contained query or construct terms. However they provide a prominent place to indicate information about the type of the data intended to be queries, e.g. a resource on where to find a schema or type declaration. From a prescriptive view, modifiers—optional (4) and negation (5)—can be treated as regular terms, as their occurrence makes no sense, if they are not applicable at their given occurrence. At the same time they should be treated as optional content in the given data model, as it makes no sense to negate or treat as optional obligatory content. In the ordered automaton model, optionality can be achieved by optionally skipping a (non ε) horizontal transition step, for unordered content models a corresponding optionality has to be introduced in the expression constructed from the Xcerpt term sequence. Conditions (3) do not use the Xcerpt term syntax, they are more in the spirit of (un)equations of variables and function applications. As such they can be handled by traditional typing (e.g. in the spirit of typing for functional programming languages) adapted to R_2G_2 type declarations. The necessary adaption is to use type intersection instead of type equality when checking variable type conformance in the environment. Rule

chaining (7) has not been considered in this deliverable. It is, among the current Web query languages, a very special feature applicable to Xcerpt but not to most of the common Web query languages. An important property to consider about a type system treating chaining in Xcerpt is subject reduction: is a language with chaining well typed, or maybe even type-able, at every stage of evaluation of the chaining. In functional and imperative programming languages, this means usually after application of β-reduction. For logic languages or deductive rule languages like Xcerpt, this means the application of substitutions.

Part IV

Outlook & Conclusion

Chapter 9

Outlook

Many promising continuations of the practical work in this deliverable have shown up. Some of them are summarized now.

9.1 Type Based Querying—an Extension to Simulation Unification

The traditional use of types in programming- and query languages is to enhance security, performance, documentation or verbosity of errors. For many languages it holds, that for a well typed program the program is equivalent to what the program would be after removing all type information and running it in a *sibling language* without type support. In traditional settings, types arguably do not alter the semantics of (well typed) programs, they may just help finding ill typed programs, the can considered to be *passive* at run time.

For Web querying in general, for Xcerpt in special, types on the side of selection constructs can be used to ensure, that the query is *reasonable* for data of given type to query, where an unreasonable query is one, that never matches any data of the given type. Types hence represent a set that may not have empty intersection with the set of the data the given selection construct matches with. As an example, the rather vague Xcerpt query term in figure 55 is expected to query HTML Table element in its given context. As the query may query arbitrary data, it is obviously well types. However, as nothing restricts the variable `TABLE` from matching with arbitrary content, the query will most likely not fulfill the author's desire. The type information is very restrictive (and it can be assumed that it is not reasonable for the given query under traditional *passive* type semantic) about what the query actually queries. It is possible, that in the given situation, the programmer really expected to query HTML tables and was inspired, that the given query would more or less fulfill his requirement.

```
CONSTRUCT
 result{ all var TABLE }
FROM
 in document("http://example.com") desc var TABLE^^Table
END
```

Code Example 55 An Xcerpt query querying arbitrary data in an HTML document—however the type annotation indicates, that the author had something else in mind...

A proposed extension to the query semantic in general, and to simulation unification in the case of Xcerpt, is to use type information of well typed queries to restrict the query, to be an *active* member of the querying process. As type information is usually given and mostly designed in an unambiguous way, sometimes involving complex rules with many details, they can be a powerful tool for the query author to exactly specify the elements he is interested in. In the example in figure 55, the type would hence restrict the rather generic query pattern to match only with HTML tables, even if the untyped query would match other elements additionally (in the given case very likely, e.g. the content of the HTML table).

9.1.1 Extending Ground Query Term Simulation With Active Type Querying

To use active types to express queries in Xcerpt, the query evaluation has to be altered. The query evaluation in Xcerpt is based on a non standard unification, called *Simulation Unification*. Simulation unification is the method binding sub-terms of queried data to variables in a query pattern, for a query pattern matching the data. A preliminary concept of Simulation unification is the so called *ground query term simulation*.

Ground query term simulation is well defined in [43]. A query term is called *ground*, if it does not contain (sub-term, label, namespace, or positional) variables. $\mathcal{T}^g \subsetneq \mathcal{T}^q$ denotes the set of all ground query terms, where $\mathcal{T}^q$ is the set of all query terms, and $\mathcal{T}^d \subsetneq \mathcal{T}^g$ denotes the set of all data terms. In essence, a ground query term simulates a data term, if (1) the labels match, and (2) there is a mapping of the child terms of the query term to the child terms of the data term, such that the query sub-terms simulate data sub-terms. The properties of the mapping are defined according to partiality or order specification of the query term, e.g. the mapping for a total unordered query term to an unordered data term has to be bijective.

Ground query term simulation is extended, such that (3) for a query term with active type annotation (which is a disjunction of types) to simulate a data term, the data term has to be valid with respect to at least one of the given data types.

Operationally, this can be achieved by applying the type checking algorithm with the given type of the top level query term to the incoming data term at evaluation time and then, if the data term is valid with respect to the given type, applying the query algorithm. Type checking a data term is reasonable, as data terms are syntactically a subset of query terms and their query term semantics is to match exactly the data term they are. As in a query term various type annotations may be given for the sub-terms (e.g. in form of a complete type annotation as a consequence of type checking the query term), the type annotations of the child query terms have to be checked against the type annotations of the data terms. For this purpose, the intersection of the disjunction of types of query and data (sub)term are checked for non emptiness.

9.1.2 Extending Simulation Unification with Active Type Querying

Based on ground query term simulation and standard unification, simulation unification is defined in [43] as the building block of Xcerpt, responsible for the evaluation of the query parts of Xcerpt rules. Simulation Unification is an algorithm that, given two terms t_1 and t_2, determines variable substitutions such that the ground instances of t_1 and t_2 simulate. The outcome of Simulation Unification is a set of substitutions called simulation unifier. Informally, a simulation unifier for a query term t^q and a construct term t^c is a set of substitutions Σ, such that each ground instance $t^{q'}$ of t^q in Σ simulates into a

ground instance $t^{c'}$ of t^c in Σ. Each substitution is a mapping of the free variables in t_q and t_c to a ground term.

To extend simulation unification with active type support, ground query term simulation has to be replaced with ground query term simulation extended with active types as introduced in section 9.1.1. Further on, the set of substitutions Σ has to be restricted to consist only of substitutions, where each mapping of variable to ground term fulfills the requirement, that the type of the variable has non empty intersection with the type of the ground term.

9.2 Optimizing Xcerpt Query Evaluation Based on Type Information

This work has been published [12] and presented at *The First International Conference on Web Reasoning and Rule Systems*.

With the vast data size on the Web and Semantic Web, reducing costs of data transfer and query evaluation for Web queries is crucial. To reduce costs, it is necessary to narrow the data candidates to query, simplify complex queries and reduce intermediate results. This can indeed be achieved using type information in queries and/or in the queried data. A static approach to optimization of web queries has been has been proposed [12] based on the typing proposed in this deliverable. By static optimization the rewriting of queries, such that, under the assumption of querying valid data (with respect to the type of the schema) the rewritten query returns the same results as the original query. A set of rules which achieves the desired optimization by schema and type based query rewriting is proposed. The approach consists in using schema information for removing incompleteness (as expressed by *'descendant'* constructs and disjunctions) from queries. The approach is presented on the query language Xcerpt, though applicable to other query languages like XQuery.

Incomplete query constructs have proved to be both essential tools for expressing Web queries and a great convenience for query authors able to focus better on the parts of the query he or she is most interested in. Though some evaluation approaches, e.g., [15] (usually limited to tree-shaped data) can handle certain incomplete queries (viz., those involving `descendant` or `following`) efficiently, most approaches suffer from lower performance for evaluating incomplete queries than for evaluating queries without incompleteness. The latter is particularly true for query processors with limited or no index support (a typical case in a Web context where query processors are often used in scenarios where data is transient rather than persistent).

In Web queries, incompleteness occurs in three forms: breadth, depth, and order. In [12] mostly breadth and depth have been addressed though also order incompleteness is briefly considered.

1. Incompleteness in depth allows a query to express restrictions of the form "there is a path between the paper and the author" without specifying the path's exact shape. The most common construct for expressing depth incompleteness is XPath's `descendant` or Xcerpt's `desc`, an unqualified, arbitrary-length path between two nodes.

2. Incompleteness in breadth allows a query to express restrictions on some children of a node without restricting others ("there is an author child of the paper but there may be other unspecified children"). Breadth incompleteness is an essential ability of all query languages. Indeed, in many languages breadth completeness is much harder to express than incompleteness—a reason why many query authors prefer to write queries with breadth incompleteness where queries with completeness would be more efficient.

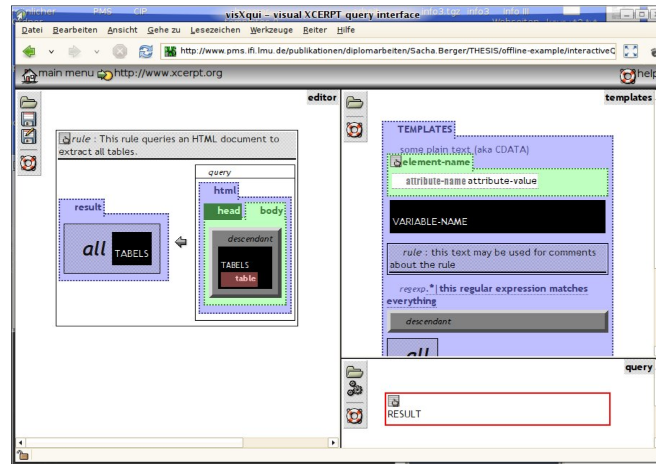


Figure 9.1: An example session of the visXcerpt Web and Semantic Web query editor. Queries (i.e. their patterns or terms) are visualized as nested boxes, possibly folded in the spirit of tabs. The editing window consists of three panels—on the left the current program being edited or used for querying, on the right at the top a set of templates to use for copy and pasting in the editor panel and on the bottom a parameter window for the query execution. The query results are shown in a pop up window using a very similar term visualisation as the visXcerpt queries use.

3. Incompleteness in order allows a query to express that the children order of a node is irrelevant ("there is an author child of the paper and a title child of the same paper but don't care about their order").

The approach presented in [12] however does not provide a fully applicable optimization approach. First of all, the problem of finding an equivalent optimal query under schema constraint is undecidable. However, heuristic approaches to rewriting queries such that the resulting query should be more efficient in many cases for a given query engine exist. The work in [12] presents rewriting rules that can be used to add or remove incompleteness under schema constraints such that the resulting query is equivalent. The current choice of rule to apply depends of a heuristic which is not presented in the work, as highly dependant on the requirements of the query execution system.

9.3 Integrating Types Into visXcerpt—an Interactive Visual Development Environment for Xcerpt

visXcerpt is a visual dialect of Xcerpt integrated with a development environment. It has first been presented in [9] and [13]. When programming in visXcerpt, the programmer uses templates to drag, drop, copy and paste queries, here and there editing various labels. This way a steep learning curve is expected to be flattened, as the user may derive his work from existing work, i.e. learning by example. Consider example 9.1 for a typical visXcerpt session

However, when constructing queries from existing example queries or documents, there is always a restricted variety of templates available and nothing prevents combination of invalid components. Using

types in such a visual environment could be greatly beneficial. Types can be used for two purposes:

1. Based on the current editing context and it's type, applicable templates can be constructed and provided for query construction.

2. Based on types invalid editing operations can be deactivated in certain contexts. As an example pasting an invalid (or at least invalid in a given context) element into a container can be prevented. On the other hand required attributes or content can be enforced or alternatively incompleteness can be provided.

The use of types in programming environments is known, e.g. from well known integrated development environments like Microsoft Visual Studio or Eclipse. Construction of valid XML and SGML documents based on DTD information has also been applied in practice for some while, e.g. in the Emacs text editor. However, schema driven query construction is a new field hot applied in practice by now.

Chapter 10

Conclusion

The contributions of this deliverable can be summarized as

1. a Web Schema language to model (serialisations of) graph structured data, including a new modelling approach for unordered data or multisets, based on unordered interpretations of regular expressions, and

2. a typing approach for web query languages based on unrestricted regular tree grammars with former extension.

The first contribution, the type and schema language R_2G_2 has been implemented for integration, e.g. in Web query or transformation languages, but also for stand alone use as schema language for XML or Xcerpt data terms. The modelling approach for unordered data based on unordered interpretation of regular expressions (1) is implemented as a prototype based on constraint solvers in GNU Prolog [25]. Typing of Web query languages (2) has been implemented based on (1) for the Web query language Xcerpt. Currently two implementations exist, a prototypical one supporting ordered and unordered query terms, and a second implementation for integration into the prototype of Xcerpt currently under development. The second version offers no support for unordered data models by now.

The need and usefulness of an extension of current type and schema languages for the Web arises from

- the lack of modelling support for graph shaped documents and their serialisations, and

- the lack of modelling techniques for unordered content models with the expressiveness of regular expressions for ordered data.

The relevance of the need of graph shaped data on the Web is paid tribute to by prominent movements like the Semantic Web with e.g. RDF at it's base, a formalism arguably modelling graph shaped data. However, many standard Web applications can benefit of modelling possibilities for valid graph shaped data. Modelling of graph shaped data and their serialisations has been achieved by adding typed references in the grammar language and cycle detection in the automata execution.

Unordered content models are either not available in Web Schema languages (i.e. DTD) or of less expressiveness than the modelling techniques for ordered data (i.e. XML Schema, Relax NG). Regular expressions provide (beside the ability to model the order of elements) a powerful mechanism to express the inter-dependency of elements and/or their optionality in content models. As shown in small use cases in chapter 3 this is of practical relevance. As a consequence, many schemata of data formats on the Web

dictate order, where indeed order is not relevant. It is however difficult to give clear numbers or hard facts about the *misuse of order*, as the lack of such modelling techniques leads to an *"as it is"* resignation of the schema developers, where the current technical possibilities are used as they are.

As first shown in [16] unrestricted regular tree grammars are computationally expensive in some situations for type checking. Various work [6] has shown arguably *"pessimistic"* results about cost or decidability of typing of Web query languages under general regular tree language constraints, however the approach has been followed in this deliverable. It is not at all the goal to contradict to the excellent theoretical results of the above mentioned work—the approach followed in this deliverable relies heavily on sacrificing precision of the typing to achieve acceptable performance and complexity (at least in the ordered data model case). The premise was not to restrict the expressiveness of the modelling languages, as they are then usually easier to understand—the restrictions are often not based on syntactical properties of the grammar formalism, but on properties of derivable automata or on features of underlying algorithms. Another advantage of using unrestricted grammars for the user is the value as a documentation of such grammars have—an easy to read and expressive grammar is valuable documentation about the data format. If the grammar of a data format has to be widened, generalized or expressed in a verbose way due to restrictions of the schema formalism used to model them, the value as a documentation of the schema is reduced.

While the typing approach presented here has been tailored for the Web and Semantic Web query language Xcerpt, it can easily be adapted to other Web or Semantic Web query languages like e.g. Xquery. The main requirement is the usefulness of R_2G_2 for the given query language as a type declaration language. A possible approach to achieve typing is the translation of the given query language to Xcerpt for the purpose of typing (investigation in the translation of Xcerpt to Xquery and vice versa has been done in [35]), but the rules may also easily be adapted to other query languages—as long as the concept of construction, projection and selection is applicable (which arguably is a central conceptualisation of all query languages).

So, what are the benefits of a (statically) typed Web query language over a non typed one? The most obvious one is help for the developer of queries, in providing prior to run time (e.g. compile time) information about errors. One could assume, that errors can easily be found by test running a program, however, this may only find errors along the execution paths used while test running—often errors may stay hidden for long time until triggered. When type checking, only a certain class of errors may be found (i.e. errors related to type conflict), but they are all found, as the type checking analyses the whole program. Further on types often serve as valuable documentation about the code.

Another arguably important benefit of typed queries is potential optimization of the evaluation based on type information. This can happen at run time or at compile time. As a rule of thumb, this kind of optimization is profitable whenever the schema is quite complex and the type information is rather selective about the data fragments, while queries are rather brief or less selective. Run time optimization could e.g. rely on checking the types of validated data against the expected results without deeply checking structures or data values in the case of type mismatch. An approach in which brief investigation has been done along this deliverable is static optimization of queries based on type annotations (see section 9.2, as well as [12]). The idea here is to rewrite queries based on their type annotation such that their selectivity is increased or their evaluation cost is reduced. A set of rewriting rules for queries under schema constraints yielding semantically equivalent queries (under the assumption, that the queried data is valid with respect to the given schema) have been proposed in this work. However, the choice of which rules to apply to achieve optimized queries can only be based on heuristics specialized for the evaluation engine of the query language in question. No such heuristics have been given by now.

Design of type systems is a difficult task and while this deliverable gives a rather concrete guide to building a type system for Xcerpt, many tasks can still be performed around the research for typing

Xcerpt and Web query languages.

Acknowledgements.

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

Bibliography

[1] J. S. 34. *Standard Generalized Markup Language (SGML)*. International Organization for Standardization (ISO), 1986. ISO 8879:1986.

[2] J. S. 34. *Document Schema Definition Languages (DSDL)*. International Organization for Standardization (ISO), 2004. ISO/IEC 19757.

[3] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[6] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *Symposium on Principles of Database Systems*, 2001.

[7] U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and J. Johannes. Modular Web Queries—From Rules to Stores. In *Proc. Int'l. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2007.

[8] U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and P.-L. Patranjan. A Generic Module System for Web Rule Languages: Divide and Rule. In *Proc. Int'l. RuleML Symp. on Rule Interchange and Applications*, 2007.

[9] S. Berger. Conception of a graphical interface for querying xml. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2003.

[10] S. Berger. *Regular Rooted Graph Grammars: A Web Type and Schema Language*. Ph.d. thesis, University of Munich, 2008.

[11] S. Berger, F. Bry, and T. Furche. Xcerpt and visxcerpt: Integrating web querying. In *Proceedings of Programming Language Technologies for XML, Charleston, South Carolina (14th January 2006)*, 2006.

[12] S. Berger, F. Bry, T. Furche, and A. J. Häusler. Completing Queries: Rewriting of Incomplete Web Queries under Schema Constraints. In *Proceedings of First International Conference on Web Reasoning and Rule Systems, Innsbruck, Austria (7th–8th June 2007)*, volume 4524 of *LNCS*, pages 319–328, 2007.

[13] 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 29th Intl. Conference on Very Large Data Bases, Berlin, Germany (9th–12th September 2003)*, 2003.

[14] S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive Typing Rules for Xcerpt. In *PPSWR*, pages 85–100, 2005.

[15] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery Processor powered by a Relational Engine. pages 479–490, New York, NY, USA, 2006. ACM Press.

[16] A. Brüggemann-Klein. *Formal Models in Document Processing*. Habilitation, Institut für Informatik, Universität Freiburg, Freiburg, Germany, 1993.

[17] F. Bry, W. Drabent, and J. Maluszynski. On Subtyping of Tree-Structured Data: A Polynomial Approach. In *Principles and Practice of Semantic Web Reasoning, Second International Workshop, PPSWR 2004, St. Malo, France, September 6-10, 2004, Proceedings*, volume 3208 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2004.

[18] F. Bry and S. Schaffert. A gentle introduction into xcerpt, a rule-based query and transformation language for xml. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Sardinia, Italy (14th June 2002)*, 2002.

[19] H.-J. Bürckert, A. Herold, D. Kapur, J. Siekmann, M. Stickel, M. Tepp, and H. Z. and. Opening the AC-Unification Race. *J. of Automated Reasoning*, 4(4):465–474, 0 1989.

[20] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 1997:1–25, 2001".

[21] H. Cirstea, E. Coquery, W. Drabent, F. Fages, C. Kirchner, L. Liquori, B. Wack, and A. Wilk. Types for REWERSE reasoning and query languages, 2005.

[22] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 12th 2007.

[23] R. L. Costello, B. Poisson, and M. Utzinger. XML Schemas: Best Practices. Technical report, xml-dev Mailinglist, 2006.

[24] E. Derksen, P. Fankhauser, E. Howland, G. Huck, I. Macherius, M. Murata, M. Resnick, and H. Schöning. XQL (XML Query Language). Technical report, August 1999. W3C Recommendation, `http://www.w3.org/TR/xquery/`.

[25] D. Diaz and P. Codognet. The GNU prolog system and its implementation. In *SAC (2)*, pages 728–732, 2000.

[26] J. e. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[27] M. J. Fischer and M. O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *SIAMAMS: Complexity of Computation: Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics*, volume 7, pages 27–41, 1974.

[28] V. Ganesh, S. Berezin, and D. Dill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design (FMCAD '02), LNCS 2517*, number 2517 in LNCS, pages 171–186. Springer, November 2002.

[29] H. Hosoya and B. C. Pierce. Xduce: A statically typed xml processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[30] International Organization for Standardization. *Document Description and Processing Languages – Regular Language Description for XML (RELAX) – Part 1: RELAX Core*, October 2000. ISO/IEC DTR 22250-1.

[31] International Organization for Standardization. *Information technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG*, November 2003. ISO/IEC 19757-2:2003.

[32] S. Kahrs. Well-Going Programs Can Be Typed. In M. Hofmann, editor, *Typed Lambda Calculi and Applications*, number 2701 in LNCS, pages 167–179. Springer, June 2003.

[33] C. Kirchner, H. Kirchner, and A. Santana. Anchoring modularity in HTML. In *First International Workshop on Automated Specification and Verification of Web Sites (WWV 2005)*, pages 139–151, 2005.

[34] S. Kraus. Use cases für xcerpt: Eine positionelle anfrage- und transformationssprache für das web. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2004.

[35] B. Linse. Automatic translation between xquery and xcerpt. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2006.

[36] D. Lugiez and S. D. Zilio. XML Schema, Tree Logic and Sheaves Automata. *Applicable Algebra in Engineering, Communication and Computing*, 17(5):337–377, 2006.

[37] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[38] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[39] F. Neven and T. Schwentick. XML Schemas without Order, 1999.

[40] R. J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

[41] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[42] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I congres de Mathematiciens des Pays Slaves*, pages 92–101, 1929.

[43] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich, 2004. PhD Thesis, Institute for Informatics, University of Munich, 2004.

[44] U. Schöning. *Theoretische Informatik – kurzgefasst*. Spektrum Akademischer Verlag in Elsevier, 1997.

[45] H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in trees for free, 2004.

[46] Unidex. *Turing Machine Markup Language*, March 2001. http://www.unidex.com/turing/tmml.htm.

[47] W3 Consortium. *Extensible Stylesheet Language Transformations (XSLT) Version 1.0*, November 1999. W3C Recommendation, http://www.w3.org/TR/xslt.

[48] W3 Consortium. *XML Path Language (XPath) Version 1.0*, November 1999. W3C Recommendation, http://www.w3.org/TR/xpath.

[49] W3 Consortium. *XHTML(TM) 1.0 The Extensible HyperText Markup Language (Second Edition)*, January 2000. W3C Recommendation, http://www.w3.org/TR/xhtml1/.

[50] W3 Consortium. *Extensible Stylesheet Language (XSL) Version 1.0*, October 2001. W3C Recommendation, http://www.w3.org/TR/2001/REC-xsl-20011015/.

[51] W3 Consortium. *TREX — Tree Regular Expressions for XML*, February 2001. W3C Recommendation, http://www.thaiopensource.com/trex/.

[52] W3 Consortium. *XML Fragment Interchange*, February 2001. W3C Recommendation, http://www.w3.org/TR/xml-fragment.

[53] W3 Consortium. *XML Linking Language (XLink) Version 1.0*, June 2001. W3C Recommendation, http://www.w3.org/TR/xlink/.

[54] W3 Consortium. *Scalable Vector Graphics (SVG) 1.1 Specification*, January 2003. W3C Recommendation, http://www.w3.org/TR/SVG/.

[55] W3 Consortium. *XHTML(TM) 2.0*, May 2003. W3C Working Draft, http://www.w3.org/TR/xhtml2/.

[56] W3 Consortium. *XML Information Set (Second Edition)*, February 2004. W3C Recommendation, http://www.w3.org/TR/xml-infoset/.

[57] W3 Consortium. *XML Schema Part 0: Primer Second Edition*, October 2004. W3C Recommendation, http://www.w3.org/TR/xmlschema-0/.

[58] W3 Consortium. *XML Schema Part 1: Structures Second Edition*, October 2004. W3C Recommendation, http://www.w3.org/TR/xmlschema-1/.

[59] W3 Consortium. *XML Schema Part 2: Datatypes Second Edition*, October 2004. W3C Recommendation, http://www.w3.org/TR/xmlschema-2/.

[60] W3 Consortium. *Document Object Model (DOM)*, January 2005. W3C Architecture domain, http://www.w3.org/DOM/.

[61] W3 Consortium. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, August 2006. W3C Candidate Recommendation , http://www.w3.org/TR/xml/.

[62] W3 Consortium. *SPARQL Query Language for RDF*, June 2007. W3C Candidate Recommendation , http://www.w3.org/TR/rdf-sparql-query/.

[63] W3 Consortium. *XQuery 1.0: An XML Query Language*, January 2007. W3C Recommendation, `http://www.w3.org/TR/xquery/`.

[64] W3 Consortium. *XQuery 1.0 and XPath 2.0 Formal Semantics*, January 2007. W3C Recommendation, `http://www.w3.org/TR/xquery-semantics/`.

[65] A. Wilk and W. Drabent. On Types for XML Query Language Xcerpt. In F. Bry, N. Hence, and J. Maluszynski, editors, *Principles and Practice of Semantic Web Reasoning*, number 2901 in LNCS, pages 128–145. Springer, 2003.

[66] A. Wilk and W. Drabent. A Prototype of a Descriptive Type System for Xcerpt. In *Principles and Practice of Semantic Web Reasoning, 4th International Workshop, PPSWR 2006, Budva, Montenegro, June 10-11, 2006, Revised Selected Papers*, volume 4187 of *Lecture Notes in Computer Science*, pages 262–275. Springer, 2006.

[67] S. D. Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 135–146, New York, NY, USA, 2004. ACM.