I4-D13

Prototypical Implementation of Xcerpt
Xcerpt 1.0 Prototype and Beyond

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/P (report and prototype)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D13/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Sacha Berger and Jakob Henriksson
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	28 February 2007
Actual submission date:	31 August 2007

Abstract

Web query languages promise convenient and efficient access to Web data such as XML, RDF, or Topic Maps. Xcerpt is one such Web query language with strong emphasis on novel high-level constructs for effective and convenient query authoring, particularly tailored to versatile access to data in different Web formats such as XML or RDF. However, without an implementation all the convenience matters little. Therefore, this deliverable introduces prototypes, APIs, and Web services that are available to Xcerpt users *now*. It also discusses a few issues related to its implementation as well as an outlook towards Xcerpt 2.0, a re-engineering of Xcerpt that is still heavily under development, but expected to provide improved performance and coverage of the language specification.

The first part of this deliverable details Xcerpt 1.0, the first prototype for Xcerpt. In particular, it contains the first published description of the Xcerpt API and its companion Web service available at `http://rewerse.net/I4/software/Xcerpt/`. The second part introduces revised principles of Xcerpt 2.0 as well as some language extensions not covered well in Xcerpt 1.0, viz. functions and modules, and their implementation

Keyword List

prototype,Xcerpt,evaluation,algorithm,API,functions,modules

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

Prototypical Implementation of Xcerpt
Xcerpt 1.0 Prototype and Beyond

Uwe Aßmann[3], Sacha Berger[1], François Bry[1], Hatice Serap Durmaz[1], Tim Furche[1], Clemens Ley[1], Benedikt Linse[1], Jakob Henriksson[3], Jendrik Johannes[3], Sebastian Schaffert[2], and Maximilian Scherr[1]

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

[2] Knowledge-based Information Systems, Salzburg Research, Salzburg, Austria
http://www.salzburgresearch.at/

[3] Fakultät für Informatik, Technische Universität Dresden
http://www-st.inf.tu-dresden.de/

31 August 2007

Abstract
Web query languages promise convenient and efficient access to Web data such as XML, RDF, or Topic Maps. Xcerpt is one such Web query language with strong emphasis on novel high-level constructs for effective and convenient query authoring, particularly tailored to versatile access to data in different Web formats such as XML or RDF. However, without an implementation all the convenience matters little. Therefore, this deliverable introduces prototypes, APIs, and Web services that are available to Xcerpt users *now*. It also discusses a few issues related to its implementation as well as an outlook towards Xcerpt 2.0, a re-engineering of Xcerpt that is still heavily under development, but expected to provide improved performance and coverage of the language specification.

The first part of this deliverable details Xcerpt 1.0, the first prototype for Xcerpt. In particular, it contains the first published description of the Xcerpt API and its companion Web service available at http://rewerse.net/I4/software/Xcerpt/. The second part introduces revised principles of Xcerpt 2.0 as well as some language extensions not covered well in Xcerpt 1.0, viz. functions and modules, and their implementation

Keyword List
prototype,Xcerpt,evaluation,algorithm,API,functions,modules

Contents

Overview of this Deliverable

The purpose of this deliverable is, first and foremost, to give an overview over the existing tools around Xcerpt, their state and availability, techniques and algorithms used for implementing, where these are relevant and novel. Second, we give an outlook on Xcerpt 2.0, the next generation implementation of Xcerpt.

The first part is devoted to descriptions of the existing Xcerpt 1.0 prototype as well as its Java-API and Web demonstrator. Chapter 1 introduces the command-line prototype for Xcerpt 1.0 (that is available *now* from Sourceforge as well as `http://rewerse.net/I4/software/Xcerpt/`). After a brief tutorial on using the command-line interface, we summarize the structure of the source code to make integration and extensions of Xcerpt easier. In particular, we discuss the implementation of the constraint solver and the memoization matrix supporting efficient simulation unification.

In Chapter 2, we detail the operational semantics of Xcerpt 1.0, viz. the rules governing constraint solving that implement the novel form of unification (used for pattern and formula matching) in Xcerpt, called simulation unification.

Easier access to Xcerpt is provided in Chapter 3 in form of an object-oriented API providing much the same features as database APIs for relational databases or XQuery such as JDBC or XQJ. We give a gentle introduction to using Xcerpt this way in form of a tutorial and also provide a sample application: a Web service the allows for easy demonstration and experimentation with Xcerpt using the Xcerpt API for accessing and presenting query results.

> **Xcerpt 1.0 Demonstrator:** `http://rewerse.net/I4/software/Xcerpt/`
> A new demonstrator for Xcerpt that allows you without any software installation to freely test Xcerpt's features. A few basic examples as well as an online tutorial are provided to get you started. Please not, that this interface is not tested extensively. In particular, don't enter large amounts of data in any of the form fields. Rather try using access to Web resources in Xcerpt itself.

The remainder of this deliverable is more forward-looking: It introduces revised principles for Xcerpt 2.0 and follows by picking out two particular issues related to the Xcerpt 2.0 prototype that we have been working on recently. Both issues are actually results of cooperations between REWERSE I3 and I4 working groups. In particular, Chapter 6 provides a look at modules in Xcerpt that is a result of a close cooperation with I3 on a framework for modules in rule languages (for more details on the generic composition framework and its use for modules see recent I3 deliverables and [41, 4]).

Chapter 1

Xcerpt 1.0 Prototype

As part of this deliverable, a prototypical runtime system for evaluating Xcerpt programs has been implemented. This runtime system (from now on called "the prototype") serves both as a testbed for new features and algorithms, and as a means to implement and test Xcerpt queries. Being a prototype, this implementation lacks some features that are desirable for practical applications (like the negation constructs `not` and `without`) and evaluation speed was not one of the primary goals (although evaluation is reasonably fast in many cases).

The runtime system is implemented in the functional language Haskell which, due to its purely functional approach, is particularly well suited for the purpose of prototypical implementations. Haskell allows to program at a very high level of abstraction and thus to stay close to the more formal definition of the evaluation algorithm(s) in Deliverable I4-D11.

The following sections illustrate various aspects of the prototype and its evaluation. Since the complete implementation is rather extensive (approximately 6500 lines of code), this chapter only highlights important aspects while the complete source code is provided in electronic form at `http://www.xcerpt.org`. Most of the code presented here is furthermore simplified over the real implementation for presentation purposes. The descriptions here are thus rather meant as a *guide* to the source code than as a standalone description and in most parts require to have the source code at hand. The documentation in this chapter is structured according to the module structure of the source code. Each section starts with a small illustration of the (sub-)module hierarchy.

The source code of the prototype is copyright of the authors and made available under the GNU General Public License (GPL), a copy of which is contained in the source archive. It uses several packages from third parties, particularly the HaXML and HXML XML parsers, and an implementation of the HTTP protocol. HaXML is available under GNU Library General Public Licence (LGPL), and HXML and HTTP under BSD license. All components are Open Source and may be distributed freely. The code is compiled with the *Glasgow Haskell Compiler* (GHC) and runs on both Unix and Windows systems. Makefiles for `make` on Unix are provided.

1.1 Usage of the Prototype

The Xcerpt prototype consists of two callable Unix or Windows binary programs:

- `xcerpt` (or `xcerpt.exe`) implements the command line interpreter
- `convert` (or `convert.exe`) converts between different Xcerpt syntaxes (i.e. XML and Xcerpt).

xcerpt can operate in two modes: either with a program as argument (evaluation mode), or in interactive command mode. The first mode of operation is most frequently used and simply evaluates the given program, which can be read either from a file or from standard input. The second mode of operation serves mainly debugging purposes and allows to test various aspects of the program evaluation (like unification).

1.1.1 Command Line Switches

The program xcerpt supports the following command line options. As is common on Unix systems, all options are prefixed by - and provided in both a short and a long form:

short option	long option	description
-V	-version	show version number
-h, -?	-help	show usage
-I	-interactive	launch interactive interface
-c	-cgi	add CGI headers in output
-g term	-goal=term	evaluate query term against program
-p FILE	-program=FILE	evaluate program
-i <*format*>	-in=<*format*>	input format
-o <*format*>	-out=<*format*>	output format

The command line switch -I starts the prototype with the interactive interface, otherwise, it is started in evaluation mode. The switch -c is useful when using Xcerpt programs as CGI[1] scripts that are evaluated on a Web server; it adds appropriate HTTP headers (like Content-Type:) to the output that allow browsers to render the result correctly.

As input format (switch -i), the prototype supports xml (the Xcerpt program is in XML syntax), xcerpt1 (the Xcerpt program is in the old Xcerpt syntax), and xcerpt2 (the Xcerpt program is in the new Xcerpt syntax). If no input format is specified, the default value of xcerpt2 is used. Output formats can be specified only, if the goals of the program do not contain an explicit format specification. The -o switch supports the same arguments as -i. Other switches are explained in the following Section.

Running an Xcerpt Program

The basic command line syntax for running an Xcerpt program is:

```
xcerpt (<program file>)   or   xcerpt -p (<program file>)
```

The latter syntax is provided for symmetry with the -I switch. In both cases, the file <*program file*> is loaded as an Xcerpt program and all goals in it are evaluated.

In combination with these commands, it is possible to use the switches -c, -i, and -o described above. In all cases, the output of the Xcerpt program is written either to the resources specified in the program or to standard output (i.e. the current console) if no explicit output resources are given. The syntax of the output again is either specified in the output resource, or the syntax specified by -o is used.

In addition, it is possible to evaluate a query term specified at the command line against the rules of the program. In this case, the prototype is called with

```
xcerpt -g <query term> -p <program file>
```

[1] *Common Gateway Interface*, a common standard for creating dynamic Web applications

Note that the switch `-p` is required, and that the specified query term must be in the syntax specified with `-i`. The query term is evaluated *only* against the rules of the program, not against its goals. This option is useful when developing Xcerpt programs. The output is a set of substitutions, always written to standard output, and in the syntax specified by the switch `-o`.

Xcerpt Programs as Unix Scripts

On Unix systems, it is possible to turn "text files" into executable scripts by providing in the first line a specification of the interpreter to use. In this case, it is sufficient to just call the script itself instead of specifying the complete command for the interpreter on the command line. For example, shell scripts for the standard Unix shell usually look as follows:

```
#!/bin/sh
echo "Hello World."
```

The first line specifies where to find the executable of the interpreter (in the example above `/bin/sh`), the rest is the content of the script. The whole script is passed to the standard input of the interpreter. The Xcerpt prototype supports this behaviour. An Xcerpt program can look as follows:

```
#!/usr/local/bin/xcerpt
GOAL
result { all var Book }
FROM
in {
resource { "file:bib.xml" },
bib {{ var Book }}
}
END
```

Assuming, the Xcerpt program is stored in a file with name `books`, it can be evaluated by just entering the command `books` instead of `xcerpt books` (assuming the permissions are set correctly). This is particularly useful when writing Web applications. In this case, the Web server does not need to be aware of Xcerpt and can simply treat the Xcerpt program as a CGI script.

Interactive Interface

The interactive interface can be started with the command `xcerpt -I`. It provides a *command prompt* indicated by the prefix symbols `?-`. The following commands are available in this interface:

Commands for program management:	
`:load <resource>`	load the program at the specified resource into memory
`:run`	run the loaded programs
`:clear`	remove all loaded programs from memory
Generic commands:	
`:quit`	leave the interactive interface
`:help`	show summary of commands
`:version`	show version information
`:reset`	remove all settings
`:set <key> = <value>`	set the property <key> to <value>
`:set`	show all options
Debugging commands:	
`:unify <t1> = <t2>`	unify <t1> and <t2> and return the resulting constraint store
`:parse <resource>`	print the term representation of the specified resource

An example session in this interactive interface (loading and running a program) looks as follows:

```
how may I help you?
?- :load prog.xcerpt
Loading prog.xcerpt ...
?- :run
<results>
```

Note that the interface might behave in unexpected ways due to Haskell's lazy evaluation. For example, the program is not actually loaded before the command `:run` is issued. As it is intended mainly for debugging the prototype, the interactive interface does not provide additional commands. It is, however, easy to add this functionality if desirable.

1.2 Overall Structure of the Source Code

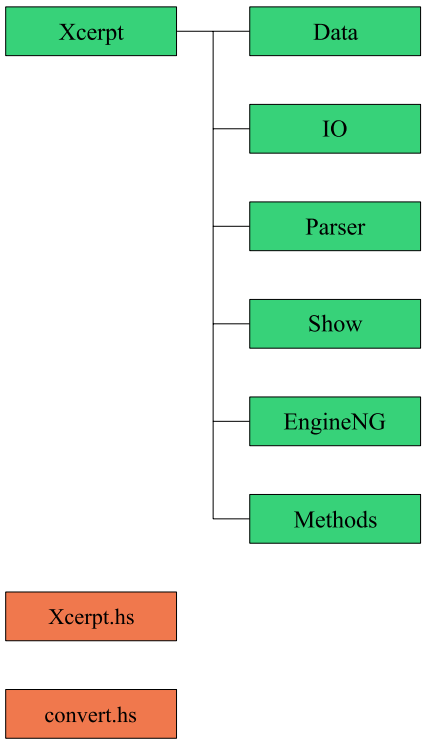

Figure 1.1: Overall module and file structure; modules in green, files in red

The source code of the runtime system is structured using Haskell's hierarchical module mechanism. The outline of the structure is shown in Figure 1.1. On the top level, there is the module `Xcerpt` containing the actual runtime system and the two files `Xcerpt.hs` and `convert.hs`, which implement the command line interface and the conversion program and both use parts of the module `Xcerpt`. The module `Xcerpt` consists of the following submodules:

Xcerpt.Data contains data structures and helper functions to operate on these structures

Xcerpt.IO contains functions for accessing local and remote resources and accessing them in Haskell

Xcerpt.Parser contains the various parser modules (currently Xcerpt version 1 and 2, XML and HTML) and provides functions for parsing strings into the data structures of `Xcerpt.Data`

Xcerpt.Show contains functions for formatting and pretty-printing the data structures of `Xcerpt.Data`

Xcerpt.EngineNG implements the core part of the runtime system the unification and the constraint-based backward chaining algorithm

Xcerpt.Methods contains the implementations of predefined functions, aggregations and comparisons that are available in Xcerpt programs

In the following, the respective modules are explained in more detail and certain aspects are highlighted to provide proficient programmers the means to modify the prototype as desired to test new features. Most of the code presented here is simplified: the prototype usually contains additional data structures or more complex function definitions that are needed for technical reasons or have been introduced for certain test cases. It is assumed that the reader is already proficient with programming in Haskell, and is familiar with tools like parser and lexer generators (like `yacc` and `lex`).

1.3 Module Xcerpt.Data: Data Structures

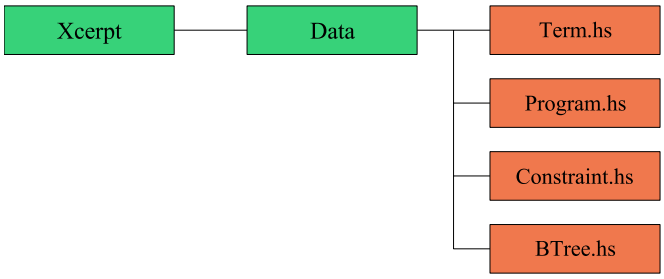

Figure 1.2: Module and File Structure of the package Xcerpt.Data; modules in green, files in red

The structure of the module `Xcerpt.Data` is shown in Figure 1.2. The module consists of four files:

Term.hs defines a unified data structure for representing data, query, and construct terms, and provides helper functions to perform various tasks on these terms (e.g. find all variables, test whether two terms are equal, ...).

Program.hs contains data structures for programs, rules, resources, and queries

Constraint.hs contains the internal data structures of the constraint solver

BTree.hs contains the definition of generic BTrees used internally in some aspects of program evaluation (see below)

`Term.hs` and `Program.hs` are the files that are most relevant to developers. Their data structures are explained in the following two sections.

1.3.1 Term.hs: Data Structures for Terms

1.3.1.1 Data Structures

Listing 1.1 defines the data structure `Term`, which is used to represent data, construct, and query terms in a unified structure. The code is simplified in that it omits some constructs to improve readability.

Listing 1.1: Data Definition of `Term`

```
data Term = Elem { label :: Term, namespace :: String,
              ordered,total :: Bool, children :: [Term] }
       | Text String
       | RegExp { pattern :: String, vars :: [Maybe String] }
       | Var String
       | String :→ Term
       | Desc Term
       | All [Term]
       | Some Int [Term]
       | Reference { identifier :: String, refers :: Maybe Term }
       | Anchor { identifier :: String, content :: Term }
```

Lines 1 and 2 define the most common form of `Term`, i.e. compound terms (e.g. `f[a,b,c]`) that consist of a *label*, a *namespace*, a subterm specification (*ordered* and *total*), and a list of subterms (*children*). A label is of type `Term`, because this allows to represent text labels, variable labels, and regular expression labels in a uniform manner; the types of the other fields are straightforward. In Haskell, field names may be used as functions for retrieving the respective field value. Assuming that a compound term is bound to a variable `t`, the following code retrieves the label of `t`:

```
label t                                                                    1
```

Lines 3 and 4 define `Terms` for representing text content and regular expressions. The definition of a regular expression consists of a regular expression pattern (in POSIX syntax without Xcerpt extensions) and a list of variables associated with the subexpressions in that pattern. Processing of Xcerpt extensions is performed during parsing.

Lines 5 and 6 define variables and variables with restrictions. A variable is always identified by its name. Variable names in the runtime system are usually different to the original variable names in the Xcerpt program, because variable renaming is performed to avoid conflicts between different rule instances.

Line 7 defines the structure of descendant terms, and lines 8 and 9 define the structure of terms of the form `all t` and `some i t`.[2].

Lines 10 and 11 define referring occurrences and defining occurrences of references. A referring occurrence (constructor `Reference`) consists of a reference name and possibly the referred term (if the reference is already dereferenced). A defining occurrence simply associates an identifier with a term.

Terms can e.g. be created in the following manner:

```
let t = Elem { label = "f", namespace = "http://www.example.com",          1
           ordered = True, total = True,
           children = [Text a, Text b, Text c] }                           3
```

Besides the definition shown above, the file `Term.hs` contains definitions for arithmetic expressions and conditions.

1.3.1.2 Helper Functions

The file `Term.hs` contains two higher order helper functions based on which most other functions are defined. Both take a function as argument and implement a generic recursive traversal over the structure of `Term`, applying the function argument to each subterm.

collectInTerm takes as arguments a term, a transformation function, a merging function, and a default value, and returns a collection of information based on the transformation and merging functions; the transformation function maps subterms to arbitrary values and the merging function merges a list of these values to a single value

recurseTerm takes as arguments a term, a transformation function (transforming one term to another), and returns a transformed term with the same structure

These generic functions are best illustrated on some examples of helper functions that are defined based on them. The following function checks whether a term contains a grouping construct. It uses the function `collectInTerm` and merges the results using `or`:

[2] both take a list of terms as arguments for future extensions

Listing 1.2: Helper function defined using `collectInTerm`

```
containsGrouping :: Term →Bool
containsGrouping (All _) = True
containsGrouping (Some _ _) = True
containsGrouping t = collectInTerm t containsGrouping or False
```

Lines 2 and 3 define that terms of the form `all t` and `some i t` contain a grouping construct. In a sense, these definitions overwrite the recursive traversal implemented by collectInTerm. Line 4 applies to all other cases and calls `collectInTerm` with `containsGrouping` as transformation function, `or` as merging function, and a default value of `False`. Assuming that the examined term is complex, `collectInTerm` applies the transformation function to all children and merges the list of results with the merging function.

Likewise, the following function uses `recurseTerm` to rename all variables in a term by adding a certain postfix (given as first argument):

Listing 1.3: Helper function defined using `recurseTerm`

```
renameVariables :: String →Term →Term
renameVariables p (Var x) = (Var x++p)
renameVariables p (x $\leadsto$ t) = (x++p) :→ renameVariables p t
renameVariables p t = recurseTerm t (renameVariables p)
```

Lines 2 and 3 add the postfix `p` to variable names, and line 3 in addition applies the function to the pattern restriction. Line 4 implements for all other terms a recursive traversal in which `(renameVariables p)` is applied to all subterms.

One of the main advantage of these two generic functions is that modifications of the data structures (e.g. adding a new kind of terms) usually only need to be reflected in the definition of these two helper functions; all functions that are based on them work without further modification. Whenever changing the data structures, it is therefore important to modify at least these two functions as well.

1.3.2 Program.hs: Data Structures for Programs

Listing 1.4: Data structures for programs

```
data Program = Prog [Rule] deriving Show
data Rule = Rule { rhead :: Term, rbody :: Query }
       | Goal { output :: [Resource], rhead :: Term, rbody :: Query }
         deriving Show
```

Consider Listing 1.4. Programs are simply represented as lists of rules. A rule is either a goal (line 3) or a standard rule (line 2). Both rules and goals consist of a rule head (a term) and a rule body (a query part – see below); in addition, goals contain a (list of) output resources.

Listing 1.5: Data structures for query parts

```
data Query = QTerm {resources :: [Resource], term :: Term }
        | QAnd { resources :: [Resource], queries :: [Query] }
        | QOr { resources :: [Resource], queries :: [Query] }
          deriving (Eq,Show)
```

A query part is either a query term, an `And`-connection of query parts, or an `Or`-connection of query parts.[3] Each query part has a list of associated resources (which might be empty), i.e. the `in` construct of Xcerpt is already resolved during parsing.

Listing 1.6: Data structures for resources

```
data Resource = XML URI
              | Xcerpt URI
              | HTML URI
              | Parsed Term
                deriving (Eq,Show)
```

Resources can be either in XML, Xcerpt, or HTML format (lines 1–3). The respective constructors are used by the parser to determine which parsing module to use. The resource is identified by a URI. Line 4 is used to represent data terms or XML/HTML documents that have already been parsed. The prototype retrieves all resources in a preprocessing step and replaces resource specifications of the first three kinds by a parsed representation. The advantage of this approach is technical: program evaluation does not need to perform I/O and thus avoids the complexity of Haskell's I/O system. Instead, it focusses on the complexity of program evaluation. While this might seem inefficient, Haskell's lazy evaluation guarantees that resources are only actually retrieved when needed. The only drawback is that it anticipates the use of variables in resource specifications.

[3] The file `Program.hs` defines some additional kinds of queries not mentioned here to improve readability.

1.4 Module Xcerpt.IO: Input/Output

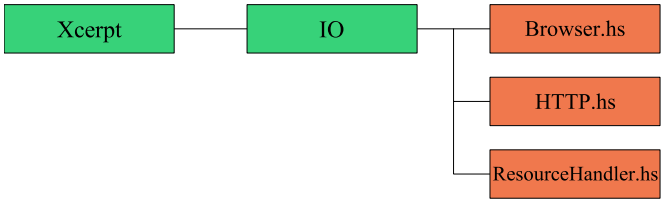

Figure 1.3: Module and File Structure of the package Xcerpt.IO; modules in green, files in red

The module `Xcerpt.IO` contains functions for performing input/output operations to local files or over the network. The module contains the following files:

ResourceHandler.hs is the main file of this module; it defines functions for retrieving a resource into a term or string

Browser.hs and HTTP.hs implement access to network resources via the HTTP protocol; they are taken from a library implemented by Warrick Gray[4] and available under the BSD license

The two important functions exported by `ResourceHandler.hs` are the following:

parseResource takes a resource specification as defined above and returns a parsed term structure of the data using the parser for the specified format

writeResource takes a term and writes it to the specified resource. The first argument is a file handle used if the specified resource is standard output (i.e. `stdout:`), in which case the output can be redirected by the system as appropriate.

[4] `http://homepages.paradise.net.nz/warrickg/haskell/http/`

1.5 Module Xcerpt.Parser: Parser

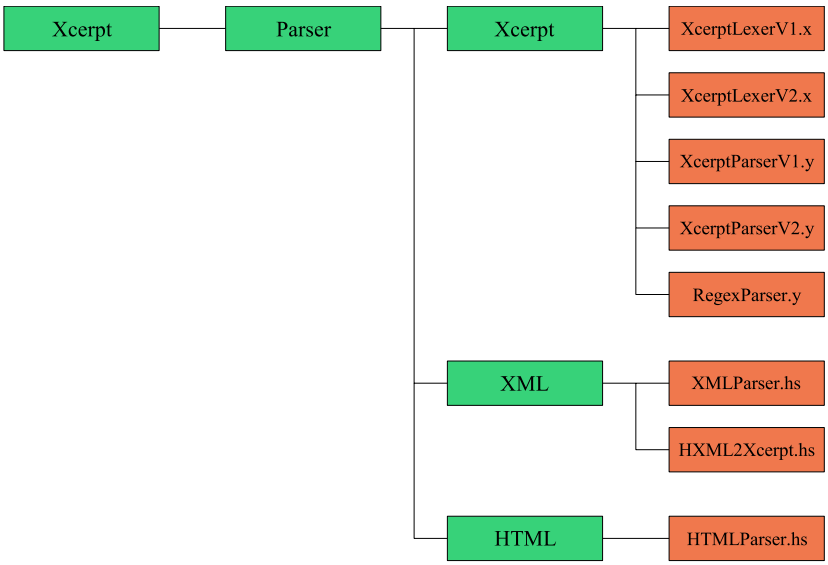

Figure 1.4: Module and File Structure of the package Xcerpt.Parser; modules in green, files in red

The Xcerpt parser currently consists of three parsing modules:

Xcerpt.Parser.Xcerpt provides functions for parsing terms and programs in Xcerpt syntax (old and new)

Xcerpt.Parser.XML provides functions for parsing terms and programs in XML syntax (based on HXML[5])

Xcerpt.Parser.HTML provides functions for parsing HTML documents into Xcerpt terms; it differs from the XML parser in that it is somewhat error-resistant and tries to also parse documents that are not well-formed XHTML

All parser modules provide the function `parseTerm` for parsing (data) terms, and the Xcerpt and XML parser in addition provide the function `parseProgram` for parsing programs (in Xcerpt and XML syntax).

1.5.1 Xcerpt.Parser.Xcerpt: Xcerpt V1 and V2 Parser

The Xcerpt parser module consists of two separate parsers: one for the old Xcerpt syntax (V1) primarily used in publications before 2004 (e.g. [19]), and one for the new Xcerpt syntax (V2) used in 2004 and later (and also in this chapter). Both parsers are implemented using the Haskell lexer generator *alex*[6] and the Haskell parser generator *happy*[7].

[5] `http://www.flightlab.com/~joe/hxml`
[6] `http://www.haskell.org/alex/`
[7] `http://www.haskell.org/happy/`

1.5.1.1 Lexer Specifications

In *alex*, tokens are defined in terms of regular expressions, similar to other lexer generators. More specific instructions for using *alex* can be found in the *alex* documentation [29]. For example, the following code defines identifiers to begin with an alphabetic character and continue with alphanumeric characters. It returns a token `TIdentifier` which stores the current position in the input file and the value of the character sequence matching the token. The first lines define character classes, and the last two lines define the token `TIdentifier`.

```
$idchar = [A-Z a-z 0-9 \- \_ \:]

tokens :-
<0> $alpha $idchar* { tok (\p s →TIdentifier p s) }
```

The files `XcerptLexerV1.x` and `XcerptLexerV2.x` contain the respective lexer definitions for the old and new Xcerpt syntax, including definitions for the various available tokens. Both files define a function `lexer` that takes as input a single string and returns as output a list of tokens.

1.5.1.2 Grammar Specifications

The parser generator *happy* uses LALR(1) grammars that consist of rules in a syntax similar to Backus-Naur Form (BNF), but extended by constructs that allow to define actions for grammar rules. Developers interested in extending or modifying the parser should consult *happy*'s documentation at [53]. For instance, the following code specifies the grammar rule for compound Xcerpt terms with partial and unordered term specification as a label (non-terminal), followed by two opening curly braces (terminals), a list of terms (non-terminal), and two closing curly braces (terminals); it returns a `Term` instance with constructor `Elem` and the field values set appropriately (occurrences of `$n` refer to the value of the n'th token of the rule). Furthermore, a list of terms (`PTermL`) is defined as either a term (non-terminal), followed by a comma and a list of terms, or a single term, or an empty list of terms; it returns a Haskell list of `Term` elements:

```
PTerm :: { Term }
PTerm : label '{' '{' PTermL '}' '}' { Elem {label=(Text $1), namespace="",
total=False, ordered=False, children=$4}}

PTermL : PTerm ',' PTermL { ($1:$3) }
| PTerm { $1:[] }
| { [] }
```

The files `XcerptParserV1.y` and `XcerptParserV2.y` contain the grammar definitions for parsing Xcerpt terms and programs. In particular, they define the functions `parseTerm` and `parseProgram`, which combine the lexer with the generated parser. Both take as input a single string and return a `Term` resp. a `Program`. In addition, the parser module contains the grammar definition `RegexParser.y`, which defines a grammar for parsing regular expressions with Xcerpt extensions. The regular expression parser is used internally inside the Xcerpt and XML parsers.

1.5.2 Xcerpt.Parser.XML: XML parser

The prototype's XML parser module uses the HXML parser for Haskell, which is very efficient and makes use of Haskell's lazy evaluation. The module consists of two files:

HXMLToXcerpt.hs provides transformation functions that convert XML data from HXML's internal data structures to the prototype's `Term` structure. In particular, these transformation functions take care of attributes and Xcerpt term constructs like term specifications or variables, and resolve namespaces.

XMLParser.hs provides transformation functions that transform a term containing appropriate constructs in the Xcerpt namespace (`http://xcerpt.org`) into a `Program`.

1.5.3 Xcerpt.Parser.HTML: HTML parser

Unfortunately, most of the HTML documents available in today's Web do not conform to the XHTML standard and are therefore not well-formed XML. To make use of existing Web pages, the Xcerpt prototype also contains an HTML parser module. This module uses the Haskell XML parser HaXML [79], which provides an error tolerant HTML parser that parses HTML documents into the same structure as XML documents. The HTML parser consists of the single file `HTMLParser.hs`, which defines a function `parseTerm` to parse HTML documents into a `Term` structure. A function `parseProgram` is not available for HTML, as Xcerpt programs cannot be represented in HTML.

1.6 Module Xcerpt.Show: Output Formatting

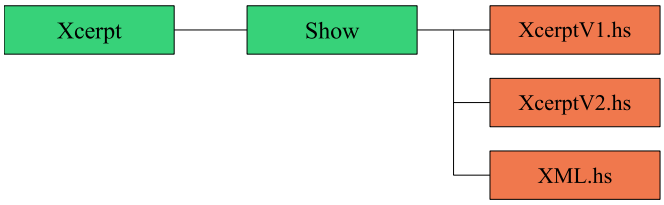

Figure 1.5: Module and File Structure of the package Xcerpt.Show; modules in green, files in red

The module `Xcerpt.Show` contains functions for pretty-printing Xcerpt data structures. The main interfaces to these functions are the classes `XcerptPrintable` and `XMLPrintable`, which define pretty-printing in Xcerpt (V1 and V2) and in XML syntax.

Listing 1.7: XcerptPrintable

```
class XcerptPrintable a where
   asXcerpt :: Int →a →String

showXcerpt :: XcerptPrintable a => a →String
showXcerpt = asXcerpt 0
```

The class `XcerptPrintable` defines a function prototype `asXcerpt` that takes the current level of nesting (`Int`) and the data structure to be printed (`a`) as arguments and returns a `String`. The function `showXcerpt` is a convenient wrapper for the default nesting level of `0`.

Listing 1.8: XMLPrintable

```
class XMLPrintable a where
   asXML :: Bool →Int →a →String

showXML :: XMLPrintable a => a →String
showXML = asXML True 0
```

Likewise, the class `XMLPrintable` defines a function prototype `asXML`. `asXML` takes as arguments a `Bool` indicating whether to add Xcerpt attributes for *ordered/unordered* and *total/partial* term specifications in the resulting XML document, an `Int` for the current level of nesting, and the data structure to be printed (`a`). Again, the function `showXML` is a convenient wrapper for default level of nesting and adding Xcerpt attributes.

Both classes are instantiated for the data structures `Term`, `Rule`, and `Program`. The module is divided into the following files:

XcerptV1.hs contains the definition and implementation of the class `XcerptPrintable` for the old Xcerpt V1 syntax (before 2004)

XcerptV2.hs contains the definition and implementation of the class `XcerptPrintable` for the new Xcerpt V2 syntax (2004 and later)

XML.hs contains the definition and implementation of the class `XMLPrintable` for the XML syntax

1.7 Module Xcerpt.EngineNG: Program Evaluation

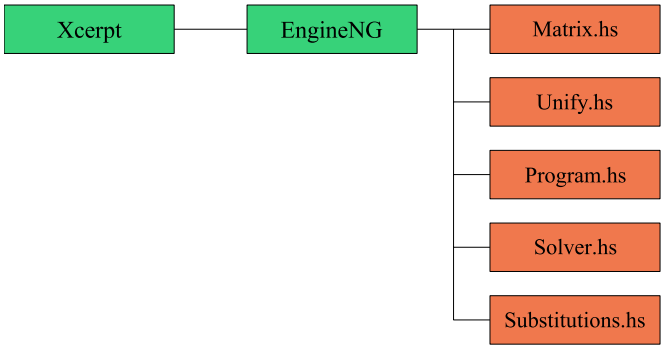

Figure 1.6: Module and File Structure of the package Xcerpt.EngineNG; modules in green, files in red

The module `Xcerpt.EngineNG` is the "heart" of the runtime system: it contains the evaluation algorithms described in Deliverable I4-D11 and consists of the following parts:

Matrix.hs contains an auxiliary data structure used by the unification algorithm called the *memoisation matrix*; using it, simulation unification can be evaluated in a rather efficient manner.

Unify.hs contains the implementation of the *simulation unification* algorithm described in Section 2.2; it uses the memoisation matrix and the constraint solver described below.

Program.hs contains the implementation of the backward chaining algorithm described in Section 2.3; it uses the unification algorithm and the constraint solver described below.

Solver.hs contains the implementation of a simple and somewhat inefficient but reliable constraint solver

Substitutions.hs implements functions for converting constraint stores into substitutions, and for applying substitutions to terms (cf. Section **??**)

The following Sections illustrate this implementation in more detail.

1.7.1 Constraint Solver

The constraint solver implemented in the file `Solver.hs` operates on a (conjunctive) list of `Constraints` and yields a list of consistent alternative conjunctions of constraints. It applies simplification rules (or "verification rules") to pairs of constraints. Each application of a verification rule yields a pair of two lists: a list of removed constraints and a list of new constraints.

```
type VerificationRule = (Constraint,Constraint) →([Constraint], [Constraint])
```

In contrast to traditional constraint solvers, the result of simplification rules in this prototype may also contain disjunctions; in the results of verification rules, these are represented by a constraint of the form `Or [...]`, and the incremental solver (in `verifyInc`) generates the disjunctive normal form represented by a list of lists of constraints (i.e. a disjunction of conjunctions of constraints).

The current implementation uses the two verification rules `consistency` and `transitivity` (both also defined in `Solver.hs`), which correspond to the respective rules in Section 2.1.4. The definition of `consistency` is given in Listing 1.9:

Listing 1.9: Consistency Rule

```
1 consistency :: VerificationRule

3 consistency (c1@(ts1@(Var v,_) :< ts2), c2@(ts1'@(Var v',_) :< ts2') )
     | (v == v') = ([c1],[solve $ andFl [(unify t2 t2'), (unify t2' t2)] ])
5    where t2 = fst ts2
          t2' = fst ts2'
7 consistency _ = ([],[])
```

The definition in line 3 catches the case where the two constraints are of the forms $var\ v \preceq t_2$ and $var\ v' \preceq t_2'$ such that $v = v'$. In this case, one of the constraints is removed (`c1`), and the two upper bounds are unified, i.e. $t_2' \preceq t_2 \wedge t_2 \preceq t_2'$ is added. Line 7 matches all other cases and neither removes nor adds constraints, indicating that `consistency` is not applicable.

The main part of the constraint solver is implemented in the function `verifyInc` (which stands for *incremental verification*). `verifyInc` takes as parameters a list of verification functions (currently only `consistency` and `transitivity`), and two lists of constraints (the current constraint store and the *increment*, i.e. the newly added constraints). The increment must always be a part of the constraint store and the constraint store without the increment is considered to be consistent; in this way, it is sufficient to only consider pairs of constraints where at least one of the constraints is part of the increment. The function `verifyInc` is implemented as follows (note the comments in the source code):

Listing 1.10: Constraint Solver

```
1 verifyInc :: [VerificationRule] →[Constraint] →[Constraint] →[[Constraint]]

3 verifyInc rules current [] = maybe [] (:[]) $ simplifyPath current

5 verifyInc rules current added = concat $ recVerify $ added'
     where run = flatPair . unzip . filter (/= ([],[])) $ map (applyRules rules)
          (pairs (id current) (added))
7
          -- recursively call verifyInc for all conjuncts in the disjunctive
9         -- normal form (see added' below); first parameter to verifyInc is
          -- the list of verification rules, second is the verified constraint
11        -- store minus the removed constraints and plus the new constraints,
          -- third is the list of new constraints (increment)
13        recVerify = map (\x →verifyInc rules (old `addList` (new x)) (new x))

15        -- the new constraints of x are the constraints of x minus the
          -- current list of constraints
17        new x = (dupelim x) `minusList` current

19        -- the remaining list of constraints is the current list of
          -- constraints minus the removed constraints
```

```
21        old = current `minusList` removed'

23        -- added' is a list of lists of constraints containing the
          -- disjunctive normal form of all additions (generated by getPaths)
25        added' = simplifyPaths $ getPaths (And $ snd run)

27        removed' = fst run

29        -- generate all pairs of the elements of two lists. since the first
          -- list always contains the second list as a tail, and the order of
31        -- the pairs is of no importance, we can drop all elements of the
          -- second list if it contains the current element.
33        pairs l1 l2 = let l1' = filter isSimConstraint l1
                            l2' = filter isSimConstraint l2
35                    in [(x,y) | x <- l1',
                                  y <- (dropIfUntil l2' x l2'), x /= y]
```

Since it uses Haskell's function combinators (`$` and `.`), the definition of the function `verifyInc` (line 5) is best read from right to left, and begins with the auxiliary definition of `run` (line 6): `run` implements a complete run over all pairs of constraints from the old constraint store (`current`) and constraints from the increment (`added`). The result is a pair consisting of a list of constraints that need to be removed, and a list of constraints that need to be added in subsequent calls of `verifyInc`. From the result of `run`, the values `added'` (line 25, containing the disjunctive normal form of the new constraints) and `removed'` (line 27, containing a list of constraints to be removed) are extracted. With these lists, the function `recVerify` (line 13) is called, which calls `verifyInc` recursively for each of the conjuncts in `added'`. The recursion terminates upon saturation, i.e. when no new constraints are added (line 3). The application of `concat` to the results of recVerify merges the results of the separate recursive calls into a single list. The result is a list of consistent conjunctions, each representing an alternative solution.

Besides `verifyInc`, the file `Solver.hs` contains a function `simplify` that can be applied to any constraint or constraint store to create a simplified representation without considering dependencies between constraints. In particular, `simplify` eliminates the constraints with boolean values of `True` or `False`. The file `Solver.hs` defines two additional convenience functions used below:

solveCS takes an arbitrary constraint (in general a constraint store), and returns a consistent constraint store in disjunctive normal form, or the boolean constraint `False`.

solveM takes a memoisation matrix containing constraints or sub-matrices (usually created in a unification), and returns a consistent constraint store in disjunctive normal form, or the boolean constraint `False`.

1.7.2 Unification

The file `Unify.hs` contains a prototypical implementation of the Simulation Unification algorithm described in Chapter I4-D11. This Section first introduces a naïve implementation, which is straightforward but has a very bad time and space behaviour. As an improvement over this approach, the so-called *memoisation matrix* (defined in the file `Matrix.hs`) is then introduced. Unification with the memoisation matrix is considerably more efficient both with respect to time and space. A further refinement of the

memoisation matrix is *matrix compactisation* (a pruning method to exclude parts that never contribute to a valid answer), with which this Section is concluded.

The implementation is described in a very simplified manner; the actual code in the prototype contains many further constructs that improve efficiency or cover some of the more complex constructs, but anticipate a clean presentation. The algorithms are described in a Haskell-like notation, with some syntactic additions that are not available in Haskell but useful for readability. In particular, it uses the Xcerpt term notation instead of the prototype's data structure.

Due to the potentially exponential size of the desired result, time and space complexity are in general exponential. However, an important measure is the number of *unification steps*, i.e. recursive calls of the `unify` function, that are performed. Each such step is computationally expensive, as it requires string comparisons of the labels and recursive calls of `unify` for (in the worst case) all possible combinations of children of the unified subterms. Thus, the number of unification steps is a measure of the number of comparisons that need to be done.

1.7.2.1 Naïve approach

When unifying two (compound) terms with matching labels, the naïve approach simply builds a disjunction of all alternative combinations of recursive unifications of the subterms and solves each separately (like the declarative description of Simulation Unification in Section 2.2). `unify` is thus a function that takes two terms as arguments and returns a `Constraint` representing the disjunction of combinations of subterm unifications and has the following signature:

```
unify :: Term →Term →Constraint
```

In the following, let `mappings` be the set of functions Π as defined in Definition 2.4 (this list can be created in Haskell in a straightforward manner). The function `unify` for two compound terms $l_1\{\{t_1, \ldots, t_n\}\}$ $l_2\{s_1, \ldots, s_m\}$ can be implemented as follows:

Listing 1.11: Naïve Implementation of `unify`

```
unify l1{{t1,...,tn}} l2{s1,...,sm} =
  if l1 /= l2 then False
  else Or [ And (zipWith unify [t1,...,tn] [sπ(1),...,sπ(n)])
                                  | π <- mappings ]
```

So, in the case of a label mismatch, the result is the atomic constraint $False$ (see rule 4 in Deliverable I4-D11). In any other cases, for each mapping π in `mappings`, a conjunctive constraint store is created by recursively applying the `unify` function to the list of children $[t_1, \ldots, t_n]$ and their mapping $[s_{\pi(1)}, \ldots, s_{\pi(n)}]$.

Example 1 *Consider a unification of the two terms* $t_1 = f\{\{var\ X, c\}\}$ *and* $t_2 = f\{a, b, c, d\}$. *Applying the naïve* `unify` *to* t_1 *and* t_2 *yields (in mathematical notation):*

$$(unify(var\ X, a) \wedge unify(c, b)) \vee (unify(var\ X, a) \wedge unify(c, c)) \vee (unify(var\ X, a) \wedge unify(c, d)) \vee$$
$$(unify(var\ X, b) \wedge unify(c, a)) \vee (unify(var\ X, b) \wedge unify(c, c)) \vee (unify(var\ X, b) \wedge unify(c, d)) \vee$$
$$(unify(var\ X, c) \wedge unify(c, a)) \vee (unify(var\ X, c) \wedge unify(c, b)) \vee (unify(var\ X, c) \wedge unify(c, d)) \vee$$
$$(unify(var\ X, d) \wedge unify(c, a)) \vee (unify(var\ X, d) \wedge unify(c, b)) \vee (unify(var\ X, d) \wedge unify(c, c))$$

or after evaluating the recursive calls of `unify`*:*

$$(var\ X \preceq a \wedge False) \vee (var\ X \preceq a \wedge True) \vee (var\ X \preceq a \wedge False) \vee$$
$$(var\ X \preceq b \wedge False) \vee (var\ X \preceq b \wedge True) \vee (var\ X \preceq b \wedge False) \vee$$
$$(var\ X \preceq c \wedge False) \vee (var\ X \preceq c \wedge False) \vee (var\ X \preceq c \wedge False) \vee$$
$$(var\ X \preceq d \wedge False) \vee (var\ X \preceq d \wedge False) \vee (var\ X \preceq d \wedge True) \vee$$

It is easy to observe that this implementation contains many redundancies (e.g. $unify(c, c)$ *is computed thrice).*

After unification, it is necessary to apply the constraint solver to the resulting constraint store in order to eliminate conjunctions that are inconsistent (either because one of the recursive unification steps fails or because two constraints exclude each other). The constraint solver in `Solver.hs` provides a function `solveCS`, which takes an arbitrary constraint store and creates a consistent constraint store in disjunctive normal form.

Complexity. As there are $\frac{m!}{(m-n)!}$ different total injective mappings from $\{t_1, \ldots, t_n\}$ to $\{s_1, \ldots, s_m\}$, the cardinality of `mappings` is $\frac{m!}{(m-n)!}$. As a consequence, the resulting disjunctive constraint store will contain $\frac{m!}{n!}$ conjunctive subformulas, and require $n \cdot \frac{m!}{(m-n)!}$ unification steps. In particular, many recursive unifications will be performed on the same pairs of subterms, leading to much redundancy.

1.7.2.2 The Memoisation Matrix

An optimisation over the naïve appriach is to remove redundant unification steps by only performing each unification of pairs of subterms once. The results of these recursive calls are stored in a matrix called the *memoisation matrix* (as it memos the results of unifications for further processing). In this manner, it is possible to reduce the number of necessary unification steps significantly; whereas the naïve approach required $n \cdot \frac{m!}{(m-n)!}$ unification steps, the memoisation matrix requires at most $n \cdot m$ unification steps at one level. Nonetheless, the desired exponential result can be created in later steps by collecting the appropriate unification results in the matrix.

The `unify` function uses the following additional data structure (defined in `Matrix.hs`) to store unification results (the actual implementation in `Matrix.hs` is much more complex, as it allows to use nested matrices and stores additional properties needed for respecting ordered and/or total term specifications):

```
data MMatrix = MMatrix [[Constraint]]
```

The matrix is initialised with all possible combinations of unifications of children from one term with children of the other term.[8] Using the terms $l_1\{\{t_1, \ldots, t_n\}\}$ and $l_2\{s_1, \ldots, s_m\}$ as above, the matrix is thus of size nŒm:

Listing 1.12: Memoisation Matrix Creation

```
initMatrix :: [Term] →[Term] →[[Constraint]]
initMatrix [] l = []
initMatrix (t:ts) [s1,...,sm] =
  ((map (unify t) [s1,...,sm]) : initMatrix ts [s1,...,sm])
```

Usage of the matrix is best illustrated on an example:

[8] Note that, using Haskell's lazy evaluation, the actual values of the cells are only computed upon use; implementations in other languages should reflect this appropriately

Example 2 *Consider a unification of the two terms $f\{\{var\ X, c\}\}$ and $f\{a, b, c, d\}$. The matrix for the children is initialised as follows:*

$t_1 \backslash t_2$	a	b	c	d
$var\ X$	$unify(var\ X, a)$	$unify(var\ X, b)$	$unify(var\ X, c)$	$unify(var\ X, d)$
c	$unify(c, a)$	$unify(c, b)$	$unify(c, c)$	$unify(c, d)$

Immediate evaluation gives the following matrix:

$t_1 \backslash t_2$	a	b	c	d
$var\ X$	$var\ X \preceq a$	$var\ X \preceq b$	$var\ X \preceq c$	$var\ X \preceq d$
c	$False$	$False$	$True$	$False$

Creating the different total mappings of subterms of the one term to subterms of the other term from this matrix is straightforward; informally, each mapping corresponds to a different "path" through the matrix such that a single cell of every row is collected. Note that this method is similar to the "Connection Method" described in [10]. The function `getPaths` serves to create all mappings:

Listing 1.13: Path Generation in Memoisation Matrix

```
getPaths :: [[Constraint]] →[[Constraint]]
getPaths [] = [[]]
getPaths [l] = map (\x →[x]) l
getPaths (x:xs) = [ (x':xs') | x' <- x, xs' <- (getPaths xs) ]
```

Using Haskell's *list comprehension*, the set of paths is expressed in a very compact manner. However, bear in mind that there are $\frac{m!}{(m-n)!}$ possible paths in the matrix. In case the term specification of the corresponding query term is ordered or total, the function `getPaths` needs to be modified appropriately to only generate monotonic or surjective mappings; this modification is straightforward and not described here.

The result of a unification is a constraint store in disjunctive normal form, the conjunctions of which each correspond to a top-down path in the memoisation matrix. In the example above, e.g. the red path represents the conjunction $var\ X \preceq a \wedge True$.

Combining the pieces introduced separately above, the function `unify` with memoisation matrix is thus implemented as follows (in simplified form):

Listing 1.14: `unify` with Memoisation Matrix

```
unify l1{{t1,...,tn}} l2{s1,...,sm} =
  if l1 /= l2 then False
  else Or . map And . getPaths $ initMatrix [t1,...,tn] [s1,...,sm]
```

During or after unification, it is necessary to resolve inconsistencies by calling the constraint solver for the resulting matrix. For this purpose, the file `Solver.hs` (described above) provides a function `solveM` that takes a (filled) memoisation matrix as input and solves each of the conjunctive paths in it. As an optimisation, the implementation in the prototype instead solves even while collecting the different paths. To this aim, the function `solveM` reverts to a different implementation for `getPaths` called `getConsistentPaths`, which uses the incremental constraint solver `verifyInc` to only generate paths that are consistent.

Complexity. The overall space and time complexity is still exponential, as the possible size of the desired result is exponential as well. However, the complexity measured in the number of unification steps in this approach is reduced to at most $n \cdot m$, where n, m are the number of nodes in t_1, t_2 respectively. Each node from t_1 is at most unified with each node from t_2, but in many practical cases less – depending on the depth and breadth of the term structure.

1.7.2.3 Matrix Compactisation

An important observation is that a large part of the fields of the matrix will evaluate to $False$ in many applications. Since a path will be translated into a conjunctive constraint store, each path containing at least one $False$ is immediately $False$ itself. It is thus desireable to not consider such paths at all.

The Xcerpt Prototype uses a *matrix compactisation* such that paths containing $False$ will not be considered. This compatisation can be implemented as follows:

Listing 1.15: Matrix Compactisation

```
compactise :: [[Constraint]] →[[Constraint]]
compactise matrix = map (filter (\x →x /= False)) matrix
```

Each row of the matrix is compactised such that it no longer contains any $False$ constraints. Thus, each path is valid if it is *and*-connected. Obviously, applying the `getPaths` function to a matrix that is compactised in such a way still returns the same (valid) paths as it would have returned without the compactisation. All missing paths are those that would have evaluated to $False$ when the *and*-connector would have been applied.

However, using the inexpensive compactisation can reduce the time and space consumption in many practical cases, as a large amount of paths usually contains at least one $False$. This can easily be seen on the example used above.

Tests conducted using the Haskell implementation as described here have shown an execution time improvement by a factor of 30 in average for the considered data. Note that this compactisation does not allow to determine whether the mapping corresponding to a path is injective and/or monotonic, so generating the correct paths for ordered unification requires to add additional information to the matrix cells. In the current prototype, this is done by adding the subterm positions for the second term.

The complete `unify` function with memoisation matrix and matrix compactisation looks as follows:

Listing 1.16: `unify` with matrix compactisation

```
unify l1{{t1,...,tn}} l2{s1,...,sm} =
  if l1 /= l2 then False
  else Or . map And . getPaths . compactise $ initMatrix [t1,...,tn] [s1,...,sm]
```

1.7.3 Backward Chaining

The backward chaining algorithm is implemented in the file `Program.hs`. The main functions exported by this file are `runProgram` (evaluate a program and write resulting terms to resources specified in program or standard output if no resource is given), `hRunProgram` (evaluate a program and write resulting terms to resources specified in program or the handle provided to this function if no resource is given), and `tRunProgram` (evaluate a program and return a list of all resulting terms, disrespecting potential resource specifications). Furthermore, this file provides the functions `evalQuery` and `evalQueryCompat` for evaluating a query part against a program instead of evaluating the goals in the program

The main data structure used by the backward chaining algorithm is a tree (structure `BTree` of module `Xcerpt.Data.BTree`) representing the current constraint store. In this tree, each leaf node represents a conjunct of the disjunctive normal form, but unlike the decomposition trees of Deliverable I4-D11, this tree does not convey the history of applications of simplification rules. The sole purpose of this tree is to provide an efficient method for building the DNF by splitting a leaf node into two or more successors if a disjunction needs to be inserted. To operate on this tree, `Program.hs` provides four internal functions `insertAtC` (to insert a constraint in a certain leaf node), `deleteAtC` (to remove a constraint in a certain leaf node), `replaceAtC` (to replace a constraint in a certain leaf node), and `replaceC` (to replace a constraint in all leaf nodes). All functions ensure that a conjunct is consistent by calling the constraint solver described above.

When evaluating a program, the algorithm loops over all conjuncts (function `runC`) in a breadth-first fashion, selects constraints that are not yet fully evaluated (function `selectC`), and applies simplification rules (function `eval`) until no more simplification rules can be applied. For this purpose, `runC` uses a data structure called `EvalContext` as helper (it mainly contains the current program and the current position in the constraint store).

The function `eval` decides, depending on the kind of constraint, how to evaluate the constraint and applies unification of query unfolding if necessary. The results are combined and the tree representing the constraint store is updated. Of particular interest is the treatment of the dependency constraint, which requires to perform an auxiliary computation before the "waiting" constraint can be evaluated. Depending on the result of this auxiliary computation, either the resulting substitutions are applied, or the constraint fails.

Query unfolding and standardisation apart is performed by the function `unfoldQuery`, which takes as an additional argument a prefix used for variable renaming. This prefix is composed depending on the current level of recursion and the position of queries in a conjunction/disjunction such that it is sufficiently unique to avoid conflicts during evaluation. Note that unfolding a query term may yield dependency constraints in case the query term is evaluated against the head of a rule containing a grouping construct.

1.8 Module Xcerpt.Methods: User-Defined Functions

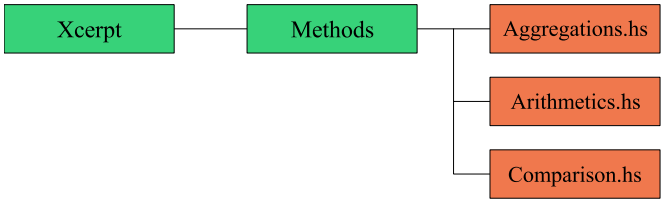

Figure 1.7: Module and File Structure of the package Xcerpt.Methods; modules in green, files in red

The module `Xcerpt.Methods` contains the definitions of functions that are available in Xcerpt either as arithmetic/string functions in construct terms (file `Arithmetics.hs`), or as aggregation functions (file `Aggregations.hs`, or as sorting specification in `order by` (file `Comparisons.hs`). So as to not add every function explicitly to the parser, all functions are stored in associated lists in which each entry consists of a pair of string and function definition.

Listing 1.17: Definition of Aggregation Functions

```
type AggregationFunction = [Term] →[Term]

aggregations :: [ (String,AggregationFunction) ]
aggregations =
   [ ("count", ((:[]) . Text . show . length ) ),
     ("sum", ((:[]) . Text . show . sum . map parseFloat) ),
     ("avg", ((:[]) . Text . show . avg . map parseFloat) ),
     ("min", ((:[]) . Text . show . min' . map parseFloat) ),
     ("max", ((:[]) . Text . show . max' . map parseFloat) ),
     ("reverse", reverse ),
     ("first", take 1 ),
     ("last", take 1 . reverse ),
     ("rest", tail ),
     ("prefix", reverse . tail . reverse )
   ]
```

Listing 1.17 shows the definition of the associated list `aggregations`, which contains the definition of the currently available aggregation functions. Helper functions (like `map'`) are omitted for space reasons. The lists in the files `Arithmetics.hs` and `Comparison.hs` are defined in a similar manner.

Extending the prototype by new user-defined functions can be achieved easily by adding new function definitions to these lists.

Chapter 2

Operational Semantics of Xcerpt 1.0

This chapter describes an algorithm for the evaluation of Xcerpt programs using a backward chaining strategy. The algorithm is defined in terms of a simple constraint solver (described in Section 2.1). Constraint solving is a method that allows a rather efficient evaluation by excluding irrelevant parts of the solution space as early as possible, and has been applied to many practical problems (cf. [33]). Constraint solving is advantageous because

- it uses declarative simplification rules that are easy to understand,
- it allows to reduce the search space by detecting inconsistencies early,
- it tries to avoid complex computations (like creating answer terms) as long as possible, and
- it allows to easily add user-defined theories specified in terms of additional simplification rules to the evaluation engine.

This constraint solver differs from traditional constraint solvers in that it needs to treat disjunctions between constraint formulas and negation, but the approach taken here is rather straightforward.

The evaluation algorithm is defined in two parts: first, an algorithm called *simulation unification* is introduced. Simulation unification is a novel kind of (non-standard) unification that allows to treat the particularities of Xcerpt terms properly and is based on the notions of ground query term simulation and answers, cf. [67]. It has first been proposed in [20] and is further refined here. Based on simulation unification, a *backward chaining* algorithm is then described that eventually determines answer terms as defined in [67]. Salient aspects of this backward chaining algorithm are the treatment of the grouping constructs `all` and `some`, and the unusually high level of branching in the proof trees that result from incomplete term specifications. While evaluation rules for programs with negation and optional subterms are given, these are not verified against the declarative semantics, as the fixpoint theory described in [67] currently does not cover negation.

This chapter is structured as follows: Section 2.1 introduces the constraint solver and data structures used in this chapter, and defines the meaning of a constraint store in form of *solution sets*. Section 2.2 describes the simplification rules that constitute simulation unification algorithm and shows the correctness of this algorithm against an abstract formalisation of most general simulation unifiers. Finally, Section 2.3 describes the rules for a backward chaining evaluation. A soundness and weak completeness result for this algorithm is also given.

2.1 A Simple Constraint Solver

The evaluation of Xcerpt programs is described in terms of a constraint solver that applies so-called *simplification rules* to a constraint store consisting of conjunctions and disjunctions of constraints. The purpose of the constraint solver is to determine variable bindings for variables occurring in query and construct terms, which ultimately yield substitutions that can be used to create the answer terms of a program. A simplification rule in this chapter has the following form:

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

where $C_1, \ldots, C_n$ ($n \geq 1$) are atomic constraints (the condition) and D is either an atomic constraint, or a conjunction or disjunction of constraints (the consequence). If a simplification rule is applied, then the conjunction $C_1 \wedge \cdots \wedge C_n$ in the constraint store is replaced by the constraint D. Note that these simplification rules are similar to the simplification rules in the language *Constraint Handling Rules* [32], albeit with a different notation.

The constraint solver is non-deterministic to a high degree in that the order in which simplification rules are applied is not significant. This approach might be advantageous, as it gives much freedom to the evaluation engine to e.g. perform optimisations (cf. Section **??**).

This constraint solver differs from common approaches in that the result of a rule may contain disjunctions, whereas usually only conjunctions are admitted. Such constraint solvers have been studied in constraint programming research, e.g. in [84]. The approach taken in this chapter is rather simplistic, as it after each application of a simplification rule creates the disjunctive normal form (DNF) of the constraint store. Simplification rules are independently applied to the different conjuncts of the DNF. This approach is rather inefficient in implementations, and various optimisations can be considered. A straightforward optimisation would be to not create the DNF after *each* simplification step, but instead only if it is "necessary", because no other simplification rules apply. However, such optimisations are not further investigated in this chapter, as the focus is on Web query languages and not on constraint programming.

Furthermore, the constraint solver needs to be able to treat negation. As both negation constructs `not` and `without` describe negation as failure, the negation behaves differently to classic negation in some cases (cf. Example 6). The treatment of negation is described in the formula simplification rules in Section 2.1.3, and in the consistency verification rules 3, 4, and 5 in Section 2.1.4 below.

2.1.1 Data Structures and Functions

2.1.1.1 Constraints

The main data structure of the evaluation algorithm is the *constraint store* which may contain several types of constraints, including other (sub-)constraint stores. For the purpose of this chapter, constraints are defined by the following grammar (defined in a variant of *Extended Backus-Naur Form*):

```
<constraint> := <conjunction> | <disjunction>
| 'True' | 'False'
| '(' <constraint> ')'
| <sim-constraint>
```

```
| <dep-constraint>
| <query-constraint> .
<conjunction> := <constraint> ('∧' <constraint>)+ .
<disjunction> := <constraint> ('∨' <constraint>)+ .
<negation> := '¬' <constraint> .
<sim-constraint> := <query-term> '≤u' <construct-term> .
<dep-constraint> := '(' <constraint> '|' <constraint> ')' .
<query-constraint> := '⟨' <query-term> '⟩''{' <data-term-list>? '}' .
<dbterm-list> := <data-term> (',' <data-term>)* .
```

It is easy to observe that a constraint store usually consists of arbitrary conjunctions, disjunctions, and negations of constraints. As usual, conjunctions always take precedence over disjunctions unless explicitly specified by parentheses. A brief description of the other kinds of constraints is given below:

Truth Values. The truth values "True" and "False" have their expected meaning in a constraint store. Simplification of the constraint store can eliminate them in all cases except when they are the only remaining constraint.

Simulation Constraint. A simulation constraint – written $t_1 \preceq_u t_2$ for some construct, data, or query term t_1 and some construct or data term t_2 – is a binary constraint which requires that variables are only bound to data terms such that there is a ground query term simulation between the ground instances of t_1 and t_2. The term t_1 is called the left hand side of the simulation constraint and t_2 is called the right hand side of the simulation constraint in subsequent sections. So as to distinguish the simulation constraint from the ground query term simulation, but nonetheless emphasise the relationship between the two, the symbol $\preceq_u$ is used (with u for "to be unified"). Note that the right hand side of a simulation constraint is always necessarily a construct or data term, because the simplification rules in the simulation unification and backward chaining algorithms never put a query term to the right hand side.

Most simulation constraints can be further reduced by applying the simulation unification algorithm on them until at least one of the sides consists merely of a variable. If a simulation constraint is of the form $X \preceq_u t$ where X is a variable, t is also called an *upper bound* of X. Likewise, if a simulation constraint is of the form $t \preceq_u X$, t is called an *lower bound* of X.

Query Constraint. A query constraint is a constraint consisting of a valid Xcerpt query (i.e. either a query term, an and/or-connection of queries, a negated query, or an input resource specification containing a query). Query constraints are used to represent queries that are not yet evaluated and are unfolded during the evaluation (if necessary). For some query Q, the query constraint is denoted by $\langle Q \rangle$.

A query constraint may optionally have a set of associated data terms which results from resolving and parsing an external resource (elimination of the `in` construct). If a query constraint $\langle Q \rangle$ is associated with the data terms $\{t_1, \ldots, t_n\}$, this is denoted by $\langle Q \rangle_{\{t_1,\ldots,t_n\}}$.

Dependency Constraint. A meta-constraint stating a dependency between two constraints. If C and D are constraints, the dependency constraint $(C \mid D)$ requires that C may only be evaluated if the evaluation of D did not fail (otherwise, the complete constraint fails). Thus D usually needs to be completely evaluated before C can be processed. The substitutions resulting from the evaluation of D are applied to C if they exist (i.e. under the condition that D is neither *False* nor *True*).

The justification for the dependency constraint are the requirements of the grouping constructs `all` and `some`, which require to consider all alternative solutions for the query part of a rule. If `all` or `some`

appears in the head of a rule which is evaluated, the unification of a query with the head cannot be completed before the rule is fully evaluated.

2.1.1.2 Functions

substitutions(CS): The ultimate step of the algorithm, after no more rules are applicable or necessary, is always to generate a set of substitutions from the constraint store. In this step, CS is put in DNF, all constraints of the form $X \preceq_u t$ (where X is a variable and t is a construct term[1]) are replaced by $X = t$ and for each conjunct of CS a separate substitution is generated from these replacements. Note that

- $substitutions(True)$ is the set of all all-grounding substitutions
- $substitutions(False) = \{\}$, i.e. there exists no substitution.

Thus, neither a result of $True$ nor a result of $False$ are desirable for a query containing variables. Fortunately, the evaluation algorithm never yields $True$ in case a variable occurs in a query, and only yields $False$ if the evaluation fails.

apply(Σ,t): Applying a set of substitutions Σ to a term is implemented recursively over the term structure. The implementation of this function can be derived from Definitions in [67] a straightforward manner.

retrieve(R): Given a resource description R, the function $retrieve(R)$ returns a set of those terms that are represented by this resource provided that the data can in some way be parsed into Xcerpt's term representation. A resource description may for example contain a URI for identifying the resource and a format specification to indicate which parser to use. The current prototype provides support for XML, HTML and Xcerpt syntax, but different formats are more or less straightforward to implement (e.g. Lisp S-expressions, RDF statements or relational databases).

restrict(V,C): restricts the constraint store C to only such (non-negated) simulation constraints where the lower bound is a variable occurring in V. This function is used for evaluating query negation below.

deref(id): Dereferences the term reference identified by id and returns the subterm associated with the identifier id.

vars(Q): Returns the set of all variables occurring non-negated in a query Q.

2.1.2 Solution Set of a Constraint Store

As the evaluation algorithm aims at determining an (all-grounding) substitution set for certain variables, each constraint store conceptually represents a (all-grounding) substitution set in which each substitution provides assignments for all conceivable variable names. This set is called the *solution set* of the constraint store, and represents the possible answers that the evaluation of the constraint store yields. Depending on the constraint store, this solution set is restricted to substitutions fulfilling certain conditions. For example, the constraint $X \preceq_u f\{a\}$ requires that all substitutions in the solution set provide an assignment for the

[1] due to the way rules are evaluated, the right hand side of a simulation constraint is always a construct term

variable X that is compatible (i.e. *simulates*) with $f\{a\}$. Likewise, the constraint $f\{\{\}\} \preceq_u X$ requires that the solution set only contains substitutions that provide an assignment t for X such that $f\{\{\}\} \preceq t$.

In the following, we will consider only the solution set of a fully solved constraint store. Such a constraint store contains only simulation constraints where one side of the inequation is a variable, of conjunctions or disjunctions of constraints, and of the boolean constraints $True$ and $False$. This notion of solution sets will be used in the formalisation of simulation unifiers later in this chapter. Recall that all-grounding substitutions are substitutions that map every possible variable to a data term.

Definition 2.1 (Solution Set of a Constraint Store) *Let CS be a completely solved constraint store, i.e. consisting only of simulation constraints where one side is a variable, conjunctions, disjunctions, and the boolean constraints $True$ and $False$. The solution set $\Omega(CS)$ is a grounding substitution set recursively defined as follows:*

- $\Omega(True)$ *is the set of all all-grounding substitutions (cf. [67])*
- $\Omega(False) = \{\}$, *i.e. the empty set*
- $\Omega(X \preceq_u t)$ *is the set of all all-grounding substitutions σ such that $\sigma(X) \cong \sigma(t)$*
- $\Omega(t \preceq_u X)$ *is the set of all all-grounding substitutions σ such that $\sigma(t) \preceq \sigma(X)$*
- $\Omega(C_1 \wedge C_2) = \Omega(C_1) \cap \Omega(C_2)$
- $\Omega(C_1 \vee C_2) = \Omega(C_1) \cup \Omega(C_2)$
- $\Omega(\neg C) = \Omega(True) \setminus \Omega(C)$

The rationale behind using sets of all-grounding substitutions is that a constraint store in general merely restricts the possible answers. Further constraints might add new variables that would have to be considered. Using infinite substitutions also simplifies working with the solution set, because it suffices to use simple set operations instead of introducing a new "substitution theory". For example, merging of two all-grounding substitution sets merely requires the intersection of both.

Note that the solution set of a constraint store CS is in general always infinite, because each substitution contains assignments for an infinite number of variables. However, restricting this set to only finitely many variables V (i.e. those variables occurring in CS), yields a finite set in case every such variable occurs in each conjunct of the disjunctive normal form of CS on the right side of a simulation constraint.

The following result is important because it relates the abstract notion of solution set to the actually computed substitutions. It follows trivially from the definition of solution sets and the definition of the function $substitutions(\cdot)$. Recall that $\Sigma_{|V}$ is the substitution set Σ restricted to the variables in V.

Corollary 1 *Let $CS = C_1 \vee \cdots \vee C_n$ be a constraint store in disjunctive normal form, and V the set of variables occurring in CS. If in every conjunct C_i, each variable $X \in V$ occurs in a simulation constraint of the form $X \preceq_u t$ where t is a data term, then $substitutions(CS) = \Omega(CS)_{|V}$.*

Note that as Xcerpt programs are range restricted, this corollary holds for every full evaluation of an Xcerpt program.

2.1.3 Constraint Simplification

The usual simplification rules for formulas apply, for example:

- $False \wedge C$ reduces to $False$ for any constraint C, $False \vee C$ reduces to C for any constraint C
- $True \wedge C$ reduces to C for any constraint C, $True \vee C$ reduces to $True$ for any constraint C
- $\neg(C \wedge D)$ simplifies to $\neg C \vee \neg D$, $\neg(C \vee D)$ simplifies to $\neg C \wedge \neg D$
- $\neg\neg\neg C$ simplifies to $\neg C$
- $\neg False = True$ and $\neg True = False$

Note, however, that constraints of the form $\neg\neg C$ (where C is not of the form $\neg C'$) may not be simplified to C, because the range restrictedness disallows variable bindings also for variables that are negated twice or more times.

2.1.4 Consistency Verification Rules

Before a variable can be bound to a term, it is necessary that the constraints for this variable are *consistent*. There are two kinds of consistency verification rules, *consistency* and *transitivity*, divided into four rules to distinguish the cases with and without negation. The fifth rule described here reduces certain kinds of negated simulation constraints.

All consistency verification rules are considered to be part of the constraint solver and are needed both for the simulation unification and the backward chaining algorithm. It is assumed that they are always applied if possible and that the constraint store can always be treated as consistent.

2.1.4.1 Rule 1: Consistency

The *consistency* rule guarantees that upper bounds for a variable are consistent. This verification rule implements the solution set definition of $\Omega(C \wedge D) = \Omega(C) \cap \Omega(D)$ and ensures that a conjunct does not induce two assignments for a variable that are not simulation equivalent.

$$\frac{X \preceq_u t_1 \quad X \preceq_u t_2}{X \preceq_u t_1 \wedge t_1 \preceq_u t_2 \wedge t_2 \preceq_u t_1}$$

Note that both t_1 and t_2 are necessarily construct or data terms. Thus, the constraint $\preceq_u$ is applicable, which requires a construct or data term on the right hand side.

Example 3 (Consistency Rule)

1. *consider the two simulation constraints $X \preceq_u f\{var\ Y\}$ and $X \preceq_u f\{a\}$; applying the consistency rule yields $X \preceq_u f\{var\ Y\} \wedge a \preceq_u Y \wedge Y \preceq_u a$ (after mutual unification), which limits the bindings for Y to a.*
2. *consider the two simulation constraints $X \preceq_u f\{a\}$ and $X \preceq_u f\{b\}$; applying the consistency rule determines that they are inconsistent, because $f\{a\}$ and $f\{b\}$ do not simulate.*

2.1.4.2 Rule 2: Transitivity

The *transitivity* rule replaces variable occurrences of a variable X in the upper bounds of a variable by the upper bound of X. This rule is justified by the simulation pre-order defined in [67] and is needed to ultimately create ground terms as bindings for all variables. In the following, the notation $t[t'/X]$ denotes "replace all occurrences of X in t by t'".

$$\frac{\begin{array}{l} t_1 \preceq_u t_1' \text{ such that } t_1' \text{ contains the variable } X \\ X \preceq_u t_2 \end{array}}{X \preceq_u t_2 \wedge t_1 \preceq_u t_1'[t_2/X]}$$

Note that the first constraint is consumed by this rule. This might appear somewhat unusual, as further applications of the transitivity rule might yield new constraints. However, if some constraint of the form $X \preceq_u t_2'$ is added, it needs to be compatible with the constraint $X \preceq_u t_2$ (which is still in the conjunction) and would thus not yield differing information.

Example 4 (Transitivity Rule)
1. *consider the simulation constraints $X \preceq_u Y$ and $Y \preceq_u a$; applying the transitivity rule yields the additional constraint $X \preceq_u a$ and removes $X \preceq_u Y$.*
2. *consider the simulation constraints $X \preceq_u f\{var\ Y\}$ and $Y \preceq_u a$; applying the transitivity rule yields the additional constraint $X \preceq_u f\{a\}$ and removes $X \preceq_u f\{var\ Y\}$.*

It would be possible to define a similar transitivity rule for the lower bounds in a simulation constraints. This is, however, not necessary, as the lower bounds do not yield variable bindings and thus need not be ground.

2.1.5 Constraint Negation

Negated constraints represent exclusion of certain variable bindings, and may result from the evaluation of the constructs `without` (subterm negation), `optional` (optional subterms), and `not` (query negation). For example, the constraint $\neg(X \preceq f\{a, b\})$ disallows bindings for X that are simulation equivalent with $f\{a, b\}$. Note that, although these constructs implement negation as failure, constraint negation is the ordinary negation of classical logic. The usual transformation rules apply, namely $\neg(C \wedge D) = \neg C \vee \neg D$, $\neg(C \vee D) = \neg C \wedge \neg D$, $\neg True = False$, and $\neg False = True$. Note, however, that $\neg\neg C \neq C$, because C is not allowed to define variable bindings.

The following three additional consistency verification rules are used in the constraint solver to treat constraint negation. All three rules assume that the negation appears immediately in front of an atomic constraint. This assumption is safe when the constraint store is in disjunctive normal form. The rules continue the numbering scheme of the previous consistency verification rules. Therefore, the first rule has number 3.

2.1.5.1 Rule 3: Consistency with Negation

To detect inconsistencies between a non-negated and a negated simulation constraints, the consistency rule needs to be modified to yield inconsistency in case a non-negated constraint for a variable is consistent with a negated constraint for the same variable. The following rule means that if a simulation constraint provides an upper bound for a variable (which represents a candidate binding for the variable), then there must not be a negated simulation constraint that excludes this upper bound:

$$\frac{X \preceq_u t_1 \quad \neg(X \preceq_u t_2)}{X \preceq_u t_1 \wedge \neg(t_1 \preceq_u t_2 \wedge t_2 \preceq_u t_1)}$$

Example 5 (Consistency Rule with Negation) *Consider the constraint store*

$$X \preceq_u f\{a, b\} \wedge \neg(X \preceq_u f\{b, a\}) \wedge \neg(X \preceq_u g\{a\})$$

Applying the consistency rule with negation yields

$$X \preceq_u f\{a, b\} \wedge \neg(f\{a, b\} \preceq_u f\{b, a\} \wedge f\{b, a\} \preceq_u f\{a, b\}) \wedge \neg(X \preceq_u g\{a\})$$

the DNF of which is

$$X \preceq_u f\{a, b\} \wedge \neg(f\{a, b\} \preceq_u f\{b, a\}) \wedge \neg(X \preceq_u g\{a\}) \vee$$
$$X \preceq_u f\{a, b\} \wedge \neg(f\{b, a\} \preceq_u f\{a, b\}) \wedge \neg(X \preceq_u g\{a\})$$

and after further decomposition steps

$$X \preceq_u f\{a, b\} \wedge \neg(True) \wedge \neg(X \preceq_u g\{a\}) \vee$$
$$X \preceq_u f\{a, b\} \wedge \neg(True) \wedge \neg(X \preceq_u g\{a\})$$

which ultimately yields False, i.e. no valid bindings.

Note that although subterm and query negation can never yield variable bindings themselves, there might be variables that only appear in negated simulation constraints but nowhere else in a non-negated simulation constraint, e.g. as the result of decomposition with `without` or `optional`. These are treated by Rule 5 below.

2.1.5.2 Rule 4: Transitivity with Negation

Like the consistency rule, the transitivity rule needs to be adapted to cover negation properly. The following rule specifies that if there is a negated simulation constraint where the upper bound t_1' contains a variable, and this variable is bounded in a non-negated simulation constraint, then substituting the upper bound for the variable in the first constraint must not yield a simulation.

$$\frac{\neg(t_1 \preceq_u t_1') \text{ such that } t_1' \text{ contains the variable } X \quad X \preceq_u t_2}{\neg(t_1 \preceq_u t_1') \wedge X \preceq_u t_2 \wedge \neg(t_1 \preceq_u t_1'[t_2/X])}$$

Likewise, if there is a non-negated simulation constraint where the upper bound contains a variable occurring in a negated simulation constraint, then substituting the upper bound for the variable in the first constraint must not yield a simulation.

$$\frac{t_1 \preceq_u t_1' \text{ such that } t_1' \text{ contains the variable } X \quad \neg(X \preceq_u t_2)}{t_1 \preceq_u t_1' \wedge \neg(X \preceq_u t_2) \wedge \neg(t_1 \preceq_u t_1'[t_2/X])}$$

Note that unlike rule 2, transitivity with negation may not remove any of the original constraints, because information would be lost.

2.1.5.3 Rule 5: Negation as Failure

The last rule is necessary for cases where a variable only appears in a negated simulation constraint, but nowhere else in a non-negated simulation constraint of the constraint store. Due to the range restrictedness of Xcerpt rules, such constraints can never be produced directly in the treatment of `not` or `without` (range restrictedness enforces that each variable occurring in a negated part also appears elsewhere in a non-negated part). They may, however, be the consequence of applications of rules 3 and 4, and might be produced when decomposing a query term containing the construct `optional` (see Section 2.2.2 below).

Such constraints are reduced to $False$. The rationale behind this is that, in case the variable does not occur elsewhere outside a negation, the simulation constraint inside the negation represents a solution for a negated query or subterm, and therefore the negated constraint must fail. In case the variable does also appear elsewhere outside a negation rules 3 and 4 are applicable (which again might yield negated simulation constraints).

$$\frac{\neg(X \preceq_u t) \text{ such that } X \text{ does not appear in a non-negated simulation constraint}}{False}$$

Constraints of the form $\neg True$ and $\neg False$ are treated by the formula simplification described above. An interesting application of this rule involves double negation:

Example 6 (Negation as Failure Rule) *Consider the simulation constraint $\neg\neg(X \preceq_u t)$ such that X does not occur elsewhere in a non-negated simulation constraint. Applying Rule 5 to this constraint yields $\neg False = True$ (and not $X \preceq_u t$ as one might expect). The rationale for this is that the negation used is negation as failure and not classical negation, and variables within a simulation constraint that are negated twice do not define variable bindings.*

2.1.6 Program Evaluation

Program evaluation starts at the program goals, and tries to determine answer terms by evaluating the query parts for each goal in a backward chaining fashion. Given a program P, the general scheme of program evaluation is as follows (the backward chaining algorithm itself is described in Section 2.3 below):

Algorithm 2.1
procedure *main():*
    foreach *goal $t \leftarrow Q \in \mathcal{P}$* ***do:***
        let *Subst* **:=** *solve($\langle Q \rangle_\varnothing$)*
        print *apply(t,Subst)*

Of course, printing the result in the scheme above has to respect a possible output resource associated with the head of a goal. The backward chaining algorithm itself is called with the function $solve(C)$ (where C is a constraint) which returns a list of substitutions that result from solving the constraint given as parameter. The general scheme of the function $solve$ is as follows (cf. the function `substitutions(·)` above):

Algorithm 2.2
function *solve(Constraint C):*
    while *a rule can be applied to C* ***do:***
        select some constraint D in C and some rule R applicable to D
        let *D'* **:=** *apply rule R to D*

replace D by D' in C
put C in disjunctive normal form and verify consistency
return *substitutions(C)*

Note that "rule" in the algorithm above denotes a simplification rule of the constraint solver and not an Xcerpt rule. Rules from all three parts may be interleaved and the decision on the selection of rule applications is deliberately left open (i.e. the algorithm described here is non-deterministic), as long as the selection is "fair" (i.e. each possible rule is applied within finitely many steps). This non-determinism allows for interesting considerations about selection strategies that have not been investigated much in logic programming.

2.2 Simulation Unification

Simulation Unification, as previously described in [20], is an algorithm that, given two terms t_1 and t_2, determines variable substitutions such that the ground instances of t_1 and t_2 simulate. Like standard unification (cf. [63]), simulation unification is *symmetric* in the sense that it can determine (partial) bindings for variables in both terms. Unlike standard unification, it is however *asymmetric* in the sense that it does not make the two terms equal, but instead ensures a ground query term simulation, which is directed and asymmetric. The outcome of Simulation Unification is a set of substitutions called *simulation unifier*.

Simulation Unification consists mainly of decomposition rules that operate recursively and in parallel on the two unified terms (Section 2.2.2). When all terms are completely decomposed, the result is a constraint store containing conjunctions and disjunctions of simulation constraints where the left or the right side is a variable. These yield variable bindings by replacing simulation constraints of the form $X \preceq_u t$ by $X = t$. The consistency verification rules described above ensure that all simulation constraints are consistent and can be interleaved at any point.

2.2.1 Simulation Unifiers

In Classical Logic, a unifier is a substitution for two terms t_1 and t_2 that, applied to t_1 and t_2, makes the two terms identical. The *simulation unifiers* introduced here follow this basic scheme, with two extensions: instead of equality, simulation unifiers are based on the (asymmetric) simulation relation of [67] and instead of a single substitution, substitution sets as in [67] are considered. Both extensions are necessary, as they recognise the special Xcerpt constructs all and $some$ and incomplete term specifications.

Informally, a *simulation unifier* for a query term t^q and a construct term t^c is a set of substitutions Σ, such that each ground instance $t^{q'}$ of t^q in Σ simulates into a ground instance $t^{c'}$ of t^c in Σ. This restriction is too weak for fully describing the semantics of the evaluation algorithm. For example, consider a substitution set $\Sigma = \{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto b, Y \mapsto a\}$, a query term $t^q = f\{var\ X\}$ and a construct term $t^c = f\{var\ Y\}$. With the informal description above, Σ would be a simulation unifier of t^q in t^c, but this is not reasonable. We therefore also require that the substitution $\sigma \in \Sigma$ that yields $t^{q'}$ also is "used" by $t^{c'}$. This can be expressed by grouping the substitutions according to the free variables in t^c.

Definition 2.2 (Simulation Unifier) *Let t^q be a query term, let t^c be a construct term with the set of free variables $FV(t^c)$, and let Σ be an all-grounding substitution set. Σ is called a* simulation unifier *of t^q in t^c, if for each $[\![\sigma]\!] \in \Sigma/_{\simeq_{FV(t^c)}}$ holds that*

$$\forall t^{q'} \in [\![\sigma]\!](t^q) \quad t^{q'} \preceq [\![\sigma]\!](t^c)$$

Recall from [67] that all substitutions in an all-grounding substitution set assign data terms to each variable. Intuitively, it is sufficient to only consider grounding substitutions for t^q and t^c. However, all-grounding substitution sets simplify the formalisation of most general simulation unifiers below.

Example 7 (Simulation Unifiers) *1. Let $t^q = f\{\{var\ X, b\}\}$ and let $t^c = f\{a, var\ Y, c\}$. A simulation unifier of t^q in t^c is the (all-grounding) substitution set*

$$\Sigma_1 = \big\{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto c, Y \mapsto b\}\big\}$$

2. Let $t^q = f\{\{var\ X\}\}$ and let $t^c = f\{all\ var\ Y\}$. A simulation unifier of t^q in t^c is the (all-grounding) substitution set

$$\Sigma_2 = \big\{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto a, Y \mapsto a\}\big\}$$

Assignments for variables not occurring in the terms t^q and t^c are not given in the substitutions above.

Simulation unifiers are required to be *grounding* substitution sets, because otherwise the simulation relation cannot be established. Also, only grounding substitution sets can be applied to construct terms containing grouping constructs, because a grouping is not possible otherwise. This restriction is less significant than it might appear: as rules in Xcerpt are range restricted, the evaluation algorithm always determines bindings for the variables in t^c, so that it is always possible to extend the solutions determined by the simulation unification algorithm to a grounding substitution set by merging with these bindings.

Usually, there are infinitely many unifiers for a query term and a construct term. Traditional logic programming therefore considers the most general unifier (mgu), i.e. the unifier that subsumes all other unifiers. Since simulation unifiers are always grounding substitution sets, such a definition is not possible for simulation unifiers. Instead, we define the *most general simulation unifier* (mgsu) as the smallest superset of all other simulation unifiers. Note that the notion *most general simulation unifier* is – although different in presentation – indeed similar to the traditional notion of most general unifiers, because a most general simulation unifier subsumes all other simulation unifiers.

Definition 2.3 (Most General Simulation Unifier) *Let t^q be a query term and let t^c be a construct term without grouping constructs such that there exists at least one simulation unifier of t^q in t^c. The* most general simulation unifier *(mgsu) of t^q in t^c is defined as the union of all simulation unifiers of t^q in t^c.*

In Section 2.2.3, we shall see that the simulation unification algorithm described here computes the most general simulation unifier. Note that the most general simulation unifier is indeed always a simulation unifier if t^c does not contain grouping constructs. This is easy to see because the union of two simulation unifiers simply adds ground instances of t^q and t^c where for every ground instance $t^{q'}$ of t^q there exists a ground instance $t^{c'}$ of t^c such that $t^{q'} \preceq t^{c'}$. This does in general not hold for construct terms with grouping, but as grouping is not treated inside the unification algorithm, the definition above suffices for the purpose of formalising the results of this algorithm.

2.2.2 Decomposition Rules

Decomposition rules take a single simulation constraint and try to recursively decompose the two terms in parallel until no further rules are applicable. Each decomposition step yields one or more subsequent constraints, often even a large disjunction containing alternatives. This reflects the many different alternative ground query term simulations that need to be considered in case of partial term specifications.

All decomposition rules are first given without examples, because the examples tend to be very extensive, and mutually depend on other decomposition rules.

2.2.2.1 Preliminaries

In the following, let l (with or without indices) denote a label, and let t^1 denote query terms and t^2 construct terms (both with or without indices). Furthermore, let $\perp$ be a special term (not occurring as subterm in any actual term) with the property that for all $t \neq \perp$ holds that $t \preceq_u \perp = False$, i.e. no term unifies with $\perp$. In the following sections, it is furthermore assumed that t^2 contains neither grouping constructs, functions, aggregations, nor optional subterms. In practice, this restriction is insignificant, because construct terms containing one of these constructs are always made ground before computing the simulation unification (see *Dependency Constraint* below).

Definition 2.4 *Given two terms $t^1 = l\{t^1_1, \ldots, t^1_n\}$ and $t^2 = l\{t^2_1, \ldots, t^2_m\}$, the following sets of functions $\Pi_X : \langle t^1_1, \ldots, t^1_n\rangle \to \langle t^2_1, \ldots, t^2_m\rangle$ are defined:*

- *$SubT^+ \subseteq \langle t^1_1, \ldots, t^1_n\rangle$ is the sequence of all non-negated subterms of t^1 and $SubT^- \subseteq \langle t^1_1, \ldots, t^1_n\rangle$ is the sequence of all negated subterms of t^1*
- *$SubT^! \subseteq \langle t^1_1, \ldots, t^1_n\rangle$ is the sequence of all non-optional subterms of t^1 and $SubT^? \subseteq \langle t^1_1, \ldots, t^1_n\rangle$ is the sequence of all optional subterms of t^1*
- *Π is the set of partial, index injective functions π from $\langle t^1_1, \ldots, t^1_n\rangle$ to $\langle t^2_1, \ldots, t^2_m\rangle$ that are total on $SubT^+$ and on $SubT^!$, each completed by $t \mapsto \perp$ for all t on which π is not defined*
- *Π_{mon} is the set Π restricted to all index monotonic functions*
- *Π_{bij} is the set Π restricted to all index bijective functions*
- *Π_{pp} is the set of all position* preserving *functions*
- *Π_{pr} is the set of all position* respecting *functions*
- *$\Pi_{m-pr} = \Pi_{mon} \cap \Pi_{pr}$, $\Pi_{b-pr} = \Pi_{bij} \cap \Pi_{pr}$, $\Pi_{b-pp} = \Pi_{bij} \cap \Pi_{pp}$, and $\Pi_{m-b} = \Pi_{bij} \cap \Pi_{mon}$*

To simplify the rules below, all *partial* mappings in Π are assumed to be completed by mapping all values on which the mappings are undefined to the special term $\perp$. In this manner, every mapping in Π can be considered to be total in case the distinction is not necessary, whereas in the cases where partial mappings are considered (`optional` and `without`), the distinction is made explicitly.

Example 8 *Consider the terms $t^1 = f[[a, \text{without } b]]$ and $t^2 = f[a, b, c]$. The set of index monotonic mappings of the set of subterms of t^1 into the set of subterms of t^2 (Π_{mon}) is as follows (`without` b abbreviated as $\neg b$):*

$$\{a \mapsto a, \neg b \mapsto \perp\} \quad \{a \mapsto b, \neg b \mapsto \perp\} \quad \{a \mapsto c, \neg b \mapsto \perp\}$$
$$\{a \mapsto a, \neg b \mapsto b\} \quad \{a \mapsto b, \neg b \mapsto c\}$$
$$\{a \mapsto a, \neg b \mapsto c\}$$

Note that all these mappings can be generated in a rather straightforward manner by creating a table with the terms $t^1_1 \cdots t^1_n$ arranged top-down and the terms $t^2_1 \cdots t^2_m$ arranged left-right and then determining paths from the first line to the n^{th} line that fulfil certain properties. This technique is called the *memoisation matrix*.

2.2.2.2 Root Elimination

Root elimination rules compare the roots of the two terms and distribute the unification to the subterms.

Brace Incompatibility The first set of rules treat incompatibility between braces and thus all of these rules reduce the simulation constraint to *False*. For instance, an ordered simulation into an unordered term is not reasonable, as the order cannot be guaranteed.
Decomposition Rule ***decomp.1***:

$$\frac{l[t_1^1, \ldots, t_n^1] \preceq_u l\{t_1^2, \ldots, t_m^2\}}{False} \qquad \frac{l[[t_1^1, \ldots, t_n^1]] \preceq_u l\{t_1^2, \ldots, t_m^2\}}{False}$$

Left Term without Subterms This set of rules consider all such cases where the left term does not contain subterms. These cases have to be treated separately from the general decomposition rules below, since using the latter would yield the wrong result in such cases. For instance, an empty *or* is equvialent to *False* but the result should always be *True* in case the left term is only a partial specification. In the following, let $m \geq 0$ and $k \geq 1$:
Decomposition Rule ***decomp.2***:

$$\frac{l\{\{\ \}\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{True} \qquad \frac{l\{\{\ \}\} \preceq_u l[t_1^2, \ldots, t_m^2]}{True} \qquad \frac{l[[\ ]] \preceq_u l[t_1^2, \ldots, t_m^2]}{True}$$

$$\frac{l\{\ \} \preceq_u l\{t_1^2, \ldots, t_k^2\}}{False} \qquad \frac{l\{\ \} \preceq_u l[t_1^2, \ldots, t_k^2]}{False} \qquad \frac{l[\ ] \preceq_u l[t_1^2, \ldots, t_k^2]}{False}$$

$$\frac{l\{\ \} \preceq_u l\{\ \}}{True} \qquad \frac{l\{\ \} \preceq_u l[\ ]}{True} \qquad \frac{l[\ ] \preceq_u l[\ ]}{True}$$

As specified by these rules, a term without subterms but a partial specification (double braces) matches with any term which has the same label. If the term specification is total, it matches only with such terms that also do not have subterms.

Decomposition without `all`, `some`, `without`, and `optional` The general decomposition rules eliminate the two root nodes in parallel and distributes the unification to the various combinations of subterms that result from ordered/unordered specification and from total/partial term specifications. If there exists no such combination, then the result is an empty *or*, which is equivalent to *False*. These term specifications are represented by the different sets of mappings Π, Π_{bij}, Π_{mon}, Π_{pr}, and Π_{pp}. In the following, let $n, m \geq 1$.
Decomposition Rule ***decomp.3***:

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{\bigvee_{\pi \in \Pi_{pp}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^1)} \qquad \frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{pr}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^1)}$$

$$\frac{l\{t_1^1, \ldots, t_n^1\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{\bigvee_{\pi \in \Pi_{bij} \cap \Pi_{pp}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^1)} \qquad \frac{l\{t_1^1, \ldots, t_n^1\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{bij} \cap \Pi_{pr}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^1)}$$

$$\frac{l[[t_1^1, \ldots, t_n^1]] \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{mon} \cap \Pi_{pr}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^1)} \qquad \frac{l[t_1^1, \ldots, t_n^1] \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{mon} \cap \Pi_{bij}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^1)}$$

For instance, if the left term has a partial, unordered specification for the subterms, the simulation unification has to consider as alternatives all combinations of subterms of the left term with subterms of the right term, provided that each child on the left gets a matching partner on the right.

Label Mismatch In case of a label mismatch, the unification fails. In the following, let $l_1 \neq l_2$.
*Decomposition Rule **decomp.4***:

$$\frac{l_1\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l_2\{t_1^2, \ldots, t_m^2\}}{False} \qquad \frac{l_1\{t_1^1, \ldots, t_n^1\} \preceq_u l_2\{t_1^2, \ldots, t_m^2\}}{False}$$

$$\frac{l_1\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l_2[t_1^2, \ldots, t_m^2]}{False} \qquad \frac{l_1\{t_1^1, \ldots, t_n^1\} \preceq_u l_2[t_1^2, \ldots, t_m^2]}{False}$$

$$\frac{l_1[[t_1^1, \ldots, t_n^1]] \preceq_u l_2[t_1^2, \ldots, t_m^2]}{False} \qquad \frac{l_1[t_1^1, \ldots, t_n^1] \preceq_u l_2[t_1^2, \ldots, t_m^2]}{False}$$

2.2.2.3 $\rightsquigarrow$ Elimination

Pattern restrictions of the form $X \rightsquigarrow t^1 \preceq_u t^2$ are decomposed by adding t_2 as upper bound for the variable X (as usual), adding the pattern restriction as lower bound for X (to ensure that there exists no upper bound that is incompatible with the pattern restriction), and immediately trying to unify t_1 and t_2. The latter step is not strictly necessary, as it would also be performed by consistency rule 2 (transitivity). However, immediate evaluation is advantageous as it excludes incompatible upper bounds immediately.
*Decomposition Rule **var***:

$$\frac{X \rightsquigarrow t^1 \preceq_u t^2}{t^1 \preceq_u t^2 \wedge t^1 \preceq_u X \wedge X \preceq_u t^2}$$

2.2.2.4 Descendant Elimination

The descendant construct in terms of the form $desc\ t$ is decomposed by first trying to unify t with the other term, and then trying to unify $desc\ t$ with each of the subterms of the other term in turn. In this manner, unifying subterms at all depths can be determined. Let $m \geq 0$.
*Decomposition Rule **desc***:

$$\frac{\text{desc } t^1 \preceq_u l\{t_1^2, \ldots, t_m^2\}}{t^1 \preceq_u l\{t_1^2, \ldots, t_m^2\} \vee \bigvee_{1 \leq i \leq m} desc\ t^1 \preceq_u t_i^2} \qquad \frac{\text{desc } t^1 \preceq_u l[t_1^2, \ldots, t_m^2]}{t^1 \preceq_u l[t_1^2, \ldots, t_m^2] \vee \bigvee_{1 \leq i \leq m} desc\ t^1 \preceq_u t_i^2}$$

2.2.2.5 Decomposition with `without`

The declarative specification of `without` in the ground query term simulation of [67] requires that a partial function (of the set of non-negated subterms into the set of subterms of the second term) is not completable to a (partial or total) function such that one of the negated subterm is mapped to a subterm in which it simulates. Since the term on the right hand side of a simulation constraint is always a data or construct term, it is sufficient to consider the case where the right term does not contain negated subterms. For a simulation constraint $t^1 \preceq_u t^2$, the decomposition rules for the case without negated subterms is intuitively described as follows:

- A mapping π is first restricted to the non-negated subterms of t^1, i.e. the subterms of the left term that are not of the form `without` t, on which the decomposition is performed in the same way as for decomposition without `without`. Note that there might be several different mappings that are identical with π for all the non-negated subterms and only differ on the negated subterms.

- It is then necessary to verify whether there exists a mapping π' that maps the non-negated subterms of t^1 to the same subterms of t^2 as π (in particular, π' might be π itself), and permits to map at least one negated subterm `without` s^1 of t^1 to a subterm s^2 of t^2 such that $s^1 \preceq s^2$. In this case, the mapping restricted to the positive subterms of t^1 is considered to be invalid, because it is completable to a mapping that allows to map a negated subterm of t^1 to a matching non-negated subterm of t^2. Thus, *all* mappings that map the positive subterms of t^1 to the same subterms of t^2 have to be ruled out.

It is important to note that the set of mappings Π is defined (in the Preliminaries above) as the set of all *partial* functions that are *total* on the set of positive subformulas. Recall furthermore, that the mappings in Π are completed by mapping all undefined values to $\bot$.

In the following, let $SubT^+ \subseteq \langle t_1^1, \ldots, t_n^1 \rangle$ be the sequence of all subterms not of the form `without` t, and let $SubT^- \subseteq \langle t_1^1, \ldots, t_n^1 \rangle$ be the sequence of all subterms of the form `without` t. Also, two functions π and π' are considered to be equal on the positive part, denoted $\pi(SubT^+) = \pi'(SubT^+)$, if for all $t \in SubT^+$ holds that $\pi(t) = \pi'(t)$. Furthermore, let $p(\cdot)$ be a function that removes the `without` construct in front of a negated subterm, i.e. $p(\text{without } t) = t$.
Decomposition Rule ***without***:

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{\bigvee_{\pi \in \Pi_{pp}} \left(\bigwedge_{t^+ \in SubT^+} t^+ \preceq_u \pi(t^+) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{pp} \text{ with } \pi(SubT^+) = \pi'(SubT^+)} \bigvee_{t^- \in SubT^-} p(t^-) \preceq_u \pi'(t^-) \right) \right)}$$

$$\frac{l[[t_1^1, \ldots, t_n^1]] \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{\text{m-pr}}} \left(\bigwedge_{t^+ \in SubT^+} t^+ \preceq_u \pi(t^+) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{\text{m-pr}} \text{ with } \pi(SubT^+) = \pi'(SubT^+)} \bigvee_{t^- \in SubT^-} p(t^-) \preceq_u \pi'(t^-) \right) \right)}$$

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{pr}} \left(\bigwedge_{t^+ \in SubT^+} t^+ \preceq_u \pi(t^+) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{pr} \text{ with } \pi(SubT^+) = \pi'(SubT^+)} \bigvee_{t^- \in SubT^-} p(t^-) \preceq_u \pi'(t^-) \right) \right)}$$

Note that decomposition with `without` is currently not covered in the completeness and correctness proofs of Section 2.2.3.

2.2.2.6 Decomposition with `optional` in the query term

Intuitively, decomposition with `optional` in the query term should "enable" the maximal number of optional subterms such that they can participate in the simulation. In the following, this is expressed as follows:

- for all required subterms (i.e. not of the form `optional` t), the treatment is as before (since all negated subterms are required, they must be treated here as well, but this is omitted in the rules below to enhance readability)
- for all optional subterms, a certain number is "enabled" by adding appropriate simulation constraints, and all others are "disabled" by adding appropriate negated simulation constraints

In the following, these requirements are expressed as follows: given a partial mapping $\pi \in \Pi$ (by definition π must be total on the set of non-optional subterms, but may be partial on the set of optional subterms), it is first verified whether π yields a simulation by unifying all terms on which π is defined with their mapping (in the same manner as before). In the second part of the formula, it is then necessary to ensure that π is also the *maximal* mapping with this property, i.e. π is not completable to a mapping π' such that this would also yield a simulation. This is ensured by adding a negated disjunction testing for all mappings that are identical with π on the subterms for which π is defined, but differ on the other subterms, whether

there exists an additional subterm that would unify with the subterm it is mapped to in π'. If yes, π is not maximal and completable to π'. If no, π is maximal.

For a given mapping π, let $SubT_\pi \subseteq SubT$ be the sequence on which π is defined and not mapped to $\perp$, i.e. for all $t \in SubT_\pi$ holds that $\pi(t) \neq \perp$, and let $\overline{SubT_\pi} = SubT \setminus SubT_\pi$. Also, two functions π and π' are considered to be equal on a set of subterms $X \subseteq SubT$, denoted $\pi(X) = \pi'(X)$, if for all $t \in X$ holds that $\pi(t) = \pi'(t)$. Furthermore, let $p(\cdot)$ be a function that removes the `optional` construct in front of an optional subterm, i.e. $p(\text{optional } t) = t$.

Decomposition Rule ***optional***:

$$\frac{l\{t_1^1, \ldots, t_n^1\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{\bigvee_{\pi \in \Pi_{b-pp}} \left(\bigwedge_{t \in SubT_\pi} t \preceq_u \pi(t) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{b-pp} \text{ with } \pi(SubT_\pi) = \pi'(SubT_\pi)} \bigvee_{t' \in \overline{SubT_\pi}} p(t') \preceq_u \pi'(t') \right) \right)}$$

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{\bigvee_{\pi \in \Pi_{pp}} \left(\bigwedge_{t \in SubT_\pi} t \preceq_u \pi(t) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{pp} \text{ with } \pi(SubT_\pi) = \pi'(SubT_\pi)} \bigvee_{t' \in \overline{SubT_\pi}} p(t') \preceq_u \pi'(t') \right) \right)}$$

$$\frac{l[t_1^1, \ldots, t_n^1] \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{m-b}} \left(\bigwedge_{t \in SubT_\pi} t \preceq_u \pi(t) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{m-b} \text{ with } \pi(SubT_\pi) = \pi'(SubT_\pi)} \bigvee_{t' \in \overline{SubT_\pi}} p(t') \preceq_u \pi'(t') \right) \right)}$$

$$\frac{l[[t_1^1, \ldots, t_n^1]] \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{m-pr}} \left(\bigwedge_{t \in SubT_\pi} t \preceq_u \pi(t) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{m-pr} \text{ with } \pi(SubT_\pi) = \pi'(SubT_\pi)} \bigvee_{t' \in \overline{SubT_\pi}} p(t') \preceq_u \pi'(t') \right) \right)}$$

$$\frac{l\{t_1^1, \ldots, t_n^1\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{b-pr}} \left(\bigwedge_{t \in SubT_\pi} t \preceq_u \pi(t) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{b-pr} \text{ with } \pi(SubT_\pi) = \pi'(SubT_\pi)} \bigvee_{t' \in \overline{SubT_\pi}} p(t') \preceq_u \pi'(t') \right) \right)}$$

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{pr}} \left(\bigwedge_{t \in SubT_\pi} t \preceq_u \pi(t) \wedge \neg \left(\bigvee_{\pi' \in \Pi_{pr} \text{ with } \pi(SubT_\pi) = \pi'(SubT_\pi)} \bigvee_{t' \in \overline{SubT_\pi}} p(t') \preceq_u \pi'(t') \right) \right)}$$

Note the close similarity to the decomposition rules for terms containing `without`. Intuitively, this similarity means that decomposition with `optional` corresponds to creating all different alternatives where zero or more `optional` subterms are "turned on" by omitting the `optional` and the others are "turned off" by replacing `optional` by `without`, and evaluating all resulting terms as alternatives. Consider for example the term

$$f\{\{var\ X \to\ a, \text{optional } var\ Y \to\ b, \text{optional } var\ Z \to\ c\}\}$$

The substitution resulting from the evaluation of this query term is equivalent to the union of the results of the four terms

$$f\{\{var\ X \to\ a, var\ Y \to\ b, var\ Z \to\ c\}\}$$
$$f\{\{var\ X \to\ a, var\ Y \to\ b, \text{without } var\ Z \to\ c\}\}$$
$$f\{\{var\ X \to\ a, \text{without } var\ Y \to\ b, var\ Z \to\ c\}\}$$
$$f\{\{var\ X \to\ a, \text{without } var\ Y \to\ b, \text{without } var\ Z \to\ c\}\}$$

Note that this representation might be surprising on a first glance, because the intuitive understanding of `optional` would be to simply leave out the optional subterms instead of replacing them by negated

subterms, as in:

$$f\{\{var\ X \to\ a, var\ Y \to\ b, var\ Z \to\ c\}\}$$
$$f\{\{var\ X \to\ a, var\ Y \to\ b\}\}$$
$$f\{\{var\ X \to\ a, var\ Z \to\ c\}\}$$
$$f\{\{var\ X \to\ a\}\}$$

However, this term representation does not reflect that an optional subterm is *required* to match, if it is *possible* to match. Consider for example a unification with the term $f\{a, c\}$. The correct solution would be the substitution set

$$\Sigma = \left\{\{X \mapsto a, Z \mapsto c\}\right\}$$

whereas the evaluation of the second set of terms would yield

$$\Sigma = \left\{\{X \mapsto a, Z \mapsto c\}, \{X \mapsto a\}\right\}$$

Note that decomposition with `optional` is currently not covered in the completeness and correctness proofs of Section 2.2.3.

2.2.2.7 Incomplete Decomposition with grouping constructs, functions, aggregations, and optional subterms in construct terms

A unification with a term containing grouping constructs, functions, or aggregations is in general incomplete because a complete decomposition requires to handle meta-constraints over the constraint store itself, which is very inconvenient. Consider for instance a unification $f\{a, b, c\} \preceq_u f[all\ X]$. To provide the full information stated in this constraint, it would be necessary to add a meta-constraint stating that there must be exactly three alternative bindings for X, and of those, one must be a, another b and the third c. Evaluation of a query containing X would thus become very complex.

Although a complete decomposition is preferable, it is (fortunately) not necessary for evaluating Xcerpt programs, as grouping constructs always depend on the bindings of the variables in the query part of a rule. Rules containing grouping constructs are treated by the *dependency constraint* (cf. Section 2.3.1), which performs an auxiliary computation for solving the query part of a rule and then substitutes the results in the rule head. Thus, in this case it is sufficient to treat the unification of a query term with a data term, which does not contain grouping constructs (and obviously also no variables).

However, it is still desirable to unify a term containing grouping constructs as far as possible in order to exclude irrelevant evaluations of query parts in the dependency constraint as early as possible. For example, the terms $f\{a, b\}$ and $g\{all\ var\ X\}$ will never yield terms that unify, regardless of the bindings for X. Likewise, the terms $f\{g\{a\}, g\{b\}\}$ and $f\{all\ h\{var\ X\}\}$ will never yield terms that unify, because neither $g\{a\}$ nor $g\{b\}$ can be successfully unified with any of the ground instances of $h\{var\ X\}$.

Therefore, the algorithm described here takes a different approach, in which a unification with *all* only yields a *necessary* set of constraints, not a *sufficient* set. The algorithm is thus *incomplete* (or "partial") in this respect.

The following decomposition rule is used, where the return value is either simply *True* or *False*, with the informal meaning "there might be a result" or "a result is precluded".
Decomposition Rule ***grouping***:

$$\frac{t^1\ \preceq_u\ \text{all}\ t^2}{(t^1\ \preceq_u\ t^2)\ \neq\ False}$$

In the case where the constraint is reduced to *True*, it is possible that there is a result, but it is also possible that there is none, depending on the further evaluation of the variables in t^2.

2.2.2.8 Term References: Memoing of Previous Computations

Resolving References. References occurring in either term of a simulation constraint are dereferenced in a straightforward manner using the $deref(\cdot)$ function described above:
*Decomposition Rule **deref***:

$$\frac{\uparrow id \preceq_u \ t^2}{t^1 \preceq_u \ t^2} t^1 = deref(id) \qquad \frac{t^1 \preceq_u \ \uparrow id}{t^1 \preceq_u \ t^2} t^2 = deref(id)$$

Memoing. Dereferencing alone is not sufficient for treating references, because the simulation unification would not terminate in case both terms contain cyclic references. The technique used by the algorithm to avoid this problem is *memoing* (also known as *tabling*). In general, memoing is used to avoid redundant computations by storing the result of all previous computations in memory (e.g. in a table). If a computation has already been performed previously, it is not necessary to repeat it as the result can simply be retrieved from memory. This technique is among others used in certain implementations of Prolog [81, 24].

Consider for example the following (naïve) implementation of the Fibonacci numbers in Haskell:

```
fib :: Int →Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Without memoing, this implementation performs many redundant computations.[2] For example, for the computation of $fib(n)$ it is necessary to compute $fib(n-1)$ and $fib(n-2)$, and for the computation of $fib(n-1)$ it is necessary to compute $fib(n-2)$ and $fib(n-3)$. Thus, $fib(n-2)$ needs to be computed twice. With memoing, the second computation could instead refer to the previous computation.

In Xcerpt, memoing for unification with references can be implemented by keeping for each conjunct in the disjunctive normal form a history of all previous applications of simplification rules (without their results) that were used for the creation of the conjunct. In every decomposition step it is then first verified whether the considered constraints have already been evaluated in a previous application of this simplification rule. If yes, the constraint reduces to $True$; if no, the computation is continued as usual.

In the following rule, let $\mathcal{H}$ be a set of constraints that have been considered in previous applications of simplification rules in the current conjunct of the disjunctive normal form (history). Furthermore, t^1 is considered to be not of the form $desc\ t$.
*Decomposition Rule **memoing***:

$$\frac{desc\ t^1 \preceq_u \ t^2 \text{ such that } desc\ t^1 \preceq_u \ t^2 \in \mathcal{H}}{False} \qquad \frac{t^1 \preceq_u \ t^2 \text{ such that } t^1 \preceq_u \ t^2 \in \mathcal{H}}{True}$$

It might be somewhat surprising that the constraint is reduced to $True/False$ instead of inserting the result of a previous computation. The rationale behind this is that the result of the previous computation is already part of the current conjunct in the disjunctive normal form. $True$ and $False$ are the neutral elements of conjunction and disjunction, and thus terminate the unification while keeping results of previous computations.

2.2.3 Soundness and Completeness

The following theorem shows soundness and completeness for the simulation unification algorithm applied to a simulation constraint of the form $t^q \preceq_u t^c$. t^q is assumed to not contain subterm negation or optional

[2] Note that Haskell's lazy evaluation performs a technique similar to memoing

subterms. Also, as rules with grouping constructs are always evaluated in an auxiliary computation using the dependency constraint, it is assumed that t^c does not contain grouping constructs. Furthermore, t^c is assumed not to contain functions, aggregations or optional subterms.

Theorem 2.5 (Soundness and Completeness of Simulation Unification) *Let t^q be a query term without subterm negation and optional subterms and let t^c be a construct term without grouping constructs, functions/aggregations, and optional subterms. A substitution set Σ is a most general simulation unifier of t^q and t^c if and only if the simulation unification of $t^q \preceq_u t^c$ terminates with a constraint store CS such that $\Sigma = \Omega(CS)$.*

2.3 Backward Chaining

The backward chaining algorithm presented here is inspired by the SLD resolution calculus used in logic programming [50]. However, traditional approaches like SLD resolution do not account well for Xcerpt constructs like partial term specification or grouping constructs. Both kinds of constructs seriously influence the resolution calculus:

High Branching Rate. In traditional logic programming, there are two elements of nondeterminism that lead to branching in the proof tree: selection of the predicate to unfold in the evaluation of a rule body, and the selection of the program rule used for further chaining. Xcerpt's usage of partial patterns adds a third element: When using partial patterns, there is in general no single way to match two terms. Instead, all possible alternative matchings have to be considered, which leads to a significantly higher branching rate.

Grouping Constructs `all` *and* `some`. Unlike Prolog's *setof* and *bagof* predicates, the grouping constructs `all` and `some` are an integral part of the language. It is hence desirable to support such higher order constructs in the proof calculus itself rather than treating them as external predicates.

In the following, a backward chaining algorithm based on constraint solving is introduced. It makes use of the simple constraint solver of Section 2.1 and the simulation unification algorithm of Section 2.2. In this algorithm, it is assumed that Xcerpt programs are range restricted, stratified, and separated apart. Evaluation always begins with a single, folded query constraint, i.e. a single constraint of the form $\langle Q \rangle$ for some goal $t^c \leftarrow_g Q$, and terminates when the constraint store either fails or is sufficiently solved to produce the answer term for the goal.[3] "Sufficiently" currently means that the constraint store is solved completely, but it might be desirable to investigate optimisations based on the construct term t^c of the goal that solve only relevant parts of the constraint store.

Instead of using backtracking to evaluate rule chaining, the backward chaining algorithm for Xcerpt uses disjunctions in the constraint store to represent alternatives. In this manner, it is possible to use other selection strategies than depth-first search for the selection of paths to evaluate. This is desirable as the `all` construct requires to find all solutions to a query anyway.

Note that the algorithm does not necessarily terminate for any input, as programs may contain recursive rules that produce infinite chains. As it is desirable to have this expressive power in Xcerpt, it is the duty of programmers to ensure that programs terminate. Non-termination might also be desirable, e.g. to produce continuous streams of data (together with the `all` construct), but such applications have not yet been investigated in detail.

The following Sections first introduce the dependency constraint as a means to treating the grouping constructs `all` and `some`, functions, and aggregations by performing an auxiliary computation. Afterwards, simplification rules for unfolding folded queries are discussed, which also implement the main part of the

[3] Recall that the result of a goal is always either failure or a single data term.

algorithm Different approaches to backward chaining in Xcerpt have been considered in the course of this chapter [21, 69]. The approach presented here is a further refinement of the "all at once" approach presented in [21].

2.3.1 Dependency Constraint

The dependency constraint is of the form $(t_1 \preceq_u t_2 \mid D)$ for a simulation constraint $t_1 \preceq_u t_2$ and some constraint D (usually a folded query) and expresses a temporal and functional dependency between $t_1 \preceq_u t_2$ and D. A dependency constraint of the form above requires to completely evaluate the constraint D in an auxiliary computation (also considering other constraints with which the dependency constraint is in conjunction) before $t_1 \preceq_u t_2$, and applies the substitution resulting from the evaluation of D to t_2 (application to t_1 is not necessary as the terms t_1 and t_2 stem from different rules and are thus variable disjoint). If the evaluation of D fails, then the dependency constraint also fails without evaluating $t_1 \preceq_u t_2$. The following simplification rule formalises this treatment:

$$\frac{(\ t_1 \preceq_u t_2 \mid D\ )}{\bigvee_{t_2' \in \Sigma(t_2)} t_1 \preceq_u t_2'} \quad \Sigma = subst(solve(D))$$

Note that if Σ is empty (i.e. there is no solution for D), the set $\Sigma(t_2)$ is empty and thus the result of the evaluation is the empty disjunction, which simplifies to $False$. In case the evaluation of D yields simply $True$, the resulting substitution set Σ is not empty, but contains the empty substitution (identity).

The dependency constraint is necessary because the (incomplete) simulation unification with a construct term containing the grouping constructs `all` or `some`, or functions and aggregations, usually does not sufficiently characterise the possible bindings of the variables in the two terms.

In order to detect inconsistencies early (and avoid unnecessary recursion), it is reasonable to perform a partial unification between the query term and the construct term and add that result to D in order to exclude such cases for which no answer can exist. Consider for instance the simulation constraint $f\{\{g\{var\ X\}\}\} \preceq_u f\{all\ h\{var\ Y\}\}$). A partial unification could determine that for all Y must hold that $g\{\cdot\} \preceq_u Y$, but not $g\{var\ X\} \preceq_u Y$ as this would possibly yield inconsistent restrictions for the variable X. The following refinement of the rule above uses the incomplete decomposition of `all` and `some` to add such information:

$$\frac{(\ t_1 \preceq_u t_2 \mid D\ )}{\bigvee_{t_2' \in \Sigma(t_2)} t_1 \preceq_u t_2'} \quad \Sigma = subst(solve(D \wedge t^1 \preceq_u t^2))$$

2.3.2 Query Unfolding

The rules for query unfolding take a folded query constraint of the form $\langle Q \rangle$ and evaluate it by "unfolding" it. For and/or connected queries, this simply means to distribute the evaluation to the subqueries and connect the corresponding folded query constraints with the respective connectives. For query terms (i.e. atomic queries), this means either to query the terms at the associated resource, or to query the construct parts of program rules. In both cases, the algorithm reverts to simulation unification for determining the solution. In case a query term queries the construct parts of program rules, it is furthermore necessary to evaluate the respective query parts of the rules and to take care of grouping constructs that possibly occur in the construct part of rules. The following query unfolding rules are used:

And/Or-Connection The connectives *and* and *or* are simply mapped to their counterparts in the constraint store. The rules for *and* and *or* are therefore straightforward:

$$\frac{\langle \texttt{and}\{Q_1,\ldots,Q_n\}\rangle_{\mathcal{R}}}{\langle Q_1\rangle_{\mathcal{R}} \wedge \cdots \wedge \langle Q_n\rangle_{\mathcal{R}}} \qquad \frac{\langle \texttt{or}\{Q_1,\ldots,Q_n\}\rangle_{\mathcal{R}}}{\langle Q_1\rangle_{\mathcal{R}} \vee \cdots \vee \langle Q_n\rangle_{\mathcal{R}}}$$

Note that the resource specification $\mathcal{R}$ is distributed recursively, and that in particular, $\mathcal{R}$ may be empty (i.e. $\mathcal{R} = \varnothing$).

Query Negation Xcerpt query negation is negation as failure (NaF), and evaluated in an auxiliary computation very much like the dependency constraint. The result of this auxiliary computation is a constraint formula C specifying which variable bindings are disallowed for the variables occurring in Q. It is thus first restricted to constraints containing variables that occur in Q and then added negated to the original constraint store. The consistency verification rules 3–5 of the constraint solver ensure that variables cannot be bound to values disallowed by C.

$$\frac{\langle \texttt{not}\ Q\rangle_{\mathcal{R}}}{\neg C} \quad V = vars(Q), C = restrict(V, solve(\langle Q\rangle_{\mathcal{R}}))$$

Resource Specification In the case where the query is the specification of an input resource, this resource needs to be retrieved. The function $retrieve(RSpec)$ takes a resource specification of any form (e.g. an URI together with a format specification of "xml" such that it can be parsed correctly) and returns a set of data terms corresponding to this resource. Note that it is also possible that a resource contains more than one term, e.g. when the resource is another Xcerpt program.

$$\frac{\langle in\{RSpec, Q\}\rangle_{\mathcal{R}'}}{\langle Q\rangle_{\mathcal{R}}} \quad \mathcal{R} = retrieve(RSpec)$$

Note that the old resource specification $\mathcal{R}'$ is shadowed by the new resource specification $\mathcal{R} = retrieve(RSpec)$

Query Term Two simplification rules process query terms. The first rule considers query terms with associated resources. In this case, the query term is unfolded to a *disjunction* of simulation constraints, one constraint for each resource. The intuitive meaning is "query any of the given resources".

$$\frac{\langle t^q\rangle_{\{t_1,\ldots,t_n\}}}{t^q \preceq_u t_1 \vee \cdots \vee t^q \preceq_u t_n}$$

The second query term unfolding works on such query terms that have *no* resource associated. In such a case, the query term is evaluated against all rules in the program. For each rule containing grouping constructs, functions, or aggregations, a dependency constraint is added which evaluates the unification between the query term and the head of the rule only, if the body of the rule can be evaluated successfully and the result can be applied to the rule head. For each rule not containing a grouping construct, the folded query is replaced by a simulation constraint between the query term and the construct term of the rule together with the (folded) query part of the rule. Each rule evaluation is an alternative, hence the result is a disjunction of constraints.

In the following, let $\mathcal{P}_{grouping} \subseteq P$ be the set of program rules $t^c \leftarrow Q$ such that t^c contains grouping constructs, functions, aggregations, or optional subterms, and let $\mathcal{P}_{nongrouping} \subseteq P$ bet the set of program rules $t^c \leftarrow Q$ such that t^c does not contain grouping constructs, functions, aggregations, or optional subterms.

Note that goals are not considered in either case, as they do not participate in chaining. Furthermore, $n \geq 0$ and $m \geq 0$.

$$\frac{\langle t^q \rangle_\varnothing}{\bigvee_{t^c \leftarrow Q \in \mathcal{P}_{grouping}} (t^q \preceq_u t^c \mid \langle Q \rangle_\varnothing) \quad \vee \quad \bigvee_{t^c \leftarrow Q \in \mathcal{P}_{nongrouping}} t^q \preceq_u t^c \wedge \langle Q \rangle_\varnothing \quad \vee \quad \bigvee_{t^d \in P} t^q \preceq_u t^d}$$

2.3.3 Soundness and Completeness

In this section, it is shown that the backward chaining algorithm is sound with respect to the fixpoint semantics described in [67], and that it is complete in all cases where the algorithm terminates. This completeness result is weak, but appears to be inherent to backward chaining. As rules with grouping constructs in the rule head require the body to be maximally satisfied, the proofs for soundness and completeness are tightly interweaved. We therefore first show the following Lemma, which is at the core of both soundness and (weak) completeness. Recall that $\Omega(CS)$ denotes the solution set of a constraint store CS.

Lemma 2.6 *Let P be a negation-free, grouping stratified Xcerpt program without goals, let M_P be the fixpoint of P, and let Q be a negation-free query (composed of one or more query terms). If the evaluation of $\langle Q \rangle$ terminates with a constraint store CS, then $\Sigma = \Omega(CS)$ is a maximal substitution set with $M_P \vDash \Sigma(Q)$.*

This Lemma contains almost all necessary "ingredients" for both soundness and completeness: it states that the solution set of the resulting constraint store is a maximal (i.e. "complete") substitution set for the satisfaction (i.e. "soundness") of the query part of a goal.

Recall for the remainder of this section that goals differ from rules in that the ground instances of the goal heads cannot be queried by query terms. This difference is not reflected in the declarative semantics described in [67], but can be achieved by ensuring that no query term simulates into a ground instance of a goal head, e.g. by wrapping goal heads as subterms of a term with a label not used elsewhere in the program.

2.3.3.1 Soundness

Theorem 2.7 (Soundness of the Backward Chaining Algorithm) *Let P be a negation-free, grouping stratified Xcerpt program, and let $G = t^c \leftarrow_g Q$ be a goal in P. If the evaluation of Q in P terminates with a constraint store $\mathcal{CS}$ inducing a grounding substitution set $\Sigma = substitutions(\mathcal{CS})$, then $\Sigma(t^c)$ is a subset of the fixpoint M_P of P.*

Proof. Let P be a negation-free, grouping stratified Xcerpt program, and let $G = t^c \leftarrow_g Q$ be a goal in P. Assume that $P' \subseteq P$ is P without the goals. According to Lemma 2.6, evaluation of $\langle Q \rangle$ in P' terminates with a constraint store $CS = D_1 \vee \cdots \vee D_n$ in disjunctive normal form such that the substitution set $\Psi = \Omega(CS)$ is a maximal substitution set with $M_{P'} \vDash \Psi(Q)$.

As the results of goals do not participate in rule chaining, adding the goals to P' does not influence the other rules in P' and only adds new data terms to $M_{P'}$. Thus, also for M_P holds that $M_P \vDash \Psi(Q)$, and Ψ is maximal. $\Psi(t^c) \subseteq M_P$ then follows from the definition of T_P. Furthermore, because P is range restricted, it holds that every variable X in t^c appears in every conjunct D_i in a simulation constraint of the form $X \preceq_u t$. Hence, with Corollary 1 follows that $substitutions(CS) = \Omega(CS)_{|V}$, where V is the set of variables occurring in t^c. Thus, $substitutions(CS)$ yields the same ground instances of t^c as $\Psi = \Omega(CS)$. The backward chaining algorithm is thus sound. .

2.3.3.2 Completeness

In general, backward chaining is incomplete with respect to the fixpoint semantics described in [67]. This is easy to see on a small example. Consider the program

$$f\{a\} \leftarrow f\{a\}$$
$$f\{a\}$$

The fixpoint for this program obviously is simply $\{f\{a\}\}$. However, evaluation of e.g. $f\{var\ X\}$ does not terminate in the backward chaining evaluation, because the rule in the program above is applicable infinitely often. This problem is not particular to Xcerpt: other logic programming languages like Prolog terminate neither with such programs.

To solve this, SLD resolution [50] uses a *fairness* clause that states that every clause (i.e. rule or data term) must be used eventually, which ensures that SLD resolution determines an answer after finitely many steps, if an answer exists. Unfortunately, this fairness clause is not applicable in Xcerpt, because the grouping constructs require to retrieve *all* solutions to a query, whereas fairness only guarantees to find *one* solution after finitely many steps. Consider for example the program

$$g\{all\ var\ X\} \leftarrow f\{var\ X\}$$
$$f\{a\} \leftarrow f\{a\}$$
$$f\{a\}$$

This program is grouping stratifiable and the fixpoint of this program is obviously $\{f\{a\}, g\{a\}\}$. Construction of the result $g\{a\}$ however requires to retrieve all solutions to $f\{var\ X\}$; a single solution does not suffice because it violates the maximality requirement in the semantics of the `all` construct.

Hence, we restrict the statement of completeness to negation-free, grouping stratified Xcerpt programs *for which the evaluation algorithm terminates*. This result is obviously somewhat unsatisfactory, because any non-terminating program would be complete under this assumption. We therefore also give criteria and suggest enhancements that ensure that programs terminate (in case the fixpoint is finite).

Theorem 2.8 (Weak Completeness of the Backward Chaining Algorithm) *Let P be a negation-free, grouping stratified Xcerpt program, with a stratification $P = P_1 \uplus \cdots \uplus P_m$ ($m \geq 1$), and let $G = t^c \leftarrow_g Q$ be a goal in P such that the evaluation of Q terminates. Assume that P has a fixpoint $M_P = T_P^\omega(P)$. If the evaluation of Q in P terminates with a constraint store CS, then CS induces a* maximal *substitution set Σ with $\Sigma(t^c) \subseteq M_P$ (i.e. there exist no other ground instances of t^c in M_P).*

Proof. By Theorem 2.7, evaluation of Q in P yields a constraint store CS inducing a substitution set Σ with $\Sigma(t^c) \subseteq M_P$. Hence, we only have to show that Σ is also maximal wrt. t^c, i.e. there exists no Σ' with $\Sigma_{|V} \subseteq \Sigma'_{|V}$ for the set of variables V occurring in t^c.

From Lemma 2.6, we know that the evaluation of $\langle Q \rangle$ in P terminates with a constraint store CS such that $\Psi = \Omega(CS)$ is a maximal substitution with $M_P \vDash \Psi(Q)$, and thus $\Psi(t^c) \subseteq M_P$. Furthermore, Ψ is maximal wrt. to Q. As by definition of goals, no ground instances of t^c besides those produced by the goal may exist[4], Ψ is thus also maximal wrt. $\Psi(t^c) \subseteq M_P$. Also, we have already seen in the proof of Theorem 2.7 that $\Sigma = substitutions(CS) = \Omega(CS)_{|V}$ where Σ yields the same ground instances of t^c as $\Omega(CS)$. Thus, Σ is also maximal wrt. $\Sigma(t^c) \subseteq M_P$. .

[4] otherwise, disambiguation is possible because results of goals do not participate in rule chaining

2.3.3.3 Criteria for Termination

No Recursion. Disallowing recursion is an obvious way to ensure termination. This restriction appears very strict on a first glance. However, due to the powerful grouping constructs `all` and `some`, this restricted class still allows many useful programs that would require recursion in traditional logic programming. For example, the program computing the sum of rows and columns in an HTML table described in [67] didn't use recursion despite the rather complex task. Likewise, many of the other examples of [67] do not require recursion while still being useful programs.

Of course, as has been argued before, there are many applications that still require recursion. It is therefore important to study refinements of this restriction that disallow only certain kinds of recursion. A useful candidate are programs where only the ground instances of rules are non-recursive (so-called *locally hierarchical programs* [61]).

Retrieving only Some Solutions. In many cases, it is actually not necessary to retrieve all solutions of the constraint store, e.g. when the rules that depend on the recursion do not contain grouping constructs. Also, a user might be satisfied with results that can be delivered in a certain time span. For both cases, the change to the evaluation algorithm would only be minor: instead of iterating as long as a rule can be applied to the constraint store, the function `solve(·)` (Section 2.1.6) would need to terminate as soon as one of the conjuncts of the constraint store is completely solved. Also, a fair rule application strategy would be necessary (e.g. breadth-first search or some other complete search strategy).

Tabling. Tabling [24] is a technique (used e.g. in XSB Prolog) where redundant and non-terminating rule applications are avoided by caching the results of previous applications, and is known to terminate more often than the SLD resolution used in standard Prolog [74]. In particular, it avoids the problem described above.

Chapter 3

Object-oriented API for Xcerpt 1.0 and 2.0

3.1 Introduction

3.1.1 Motivation

Xcerpt The language Xcerpt [67] is a declarative, rule-based query language for graph-structured data. Xcerpt aims to be simple to use for users and nevertheless to be powerful enough to build complex query programs.

In contrast to other XML [78] query languages, Xcerpt provides means to reason with Semantic Web data similar to those of other rule-based or logic programming languages (e.g. Prolog) as well as with conventional Web data.

Altough there are a wide range of standard query languages such as the W3C recommendations XQuery [77] and XSLT [75], Xcerpt differs from them in that it is better suited for many tasks than other languages. Due to Xcerpt's deductive, rule-based nature, this is particularly true for Semantic Web applications, where reasoning is of major importance.

APIs An Application Programming Interface (API) is a language interface for the communication between a software application and an operating system or any other underlying software system. An API supports easy access to software applications from other applications, no matter which programming language the other applications are written in. Using APIs do not require deep knowledge of the underlying software system. The programmer knows what the underlying software system does and what it is useful for, but usually he does not know how it is used effectively. An API's main purpose is the ease of use of underlying systems and an API is central for *developing layered* software, where the *lower layer* implementations can be exchanged without affecting the upper layers.

APIs provide interfaces, classes and methods, which perform common tasks needed in the underlying system. An API must be designed with greater care than usual interfaces as they are expected to be used by different applications and, once released, hard to change. Most often an API provides possibilities to implement more efficient applications than if the programmer himself takes care of the communication with the underlying system.

APIs usually evolve as the underlying software system evolves. Using an API helps encapsulating the

underlying software system. If the underlying software system changes, the programmer using an API does not have to rewrite the programs. The programmer just has to ensure that he uses an up to date implementation of the APIs.

Xcerpt API Standard APIs are defined for most popular query language, such as *XQJ* for XQuery or *JDBC* for SQL. These APIs are based on abstract specifications, and concrete implementations are provided for a wide range of programming languages. Providing APIs helps these languages to increase their popularity. It is obvious why a well specified API facilitates this process: the usage of the query language is clearly defined, the programmer does not have to deal with particularities of the underlying query language engine and he can easily exchange different implementations of the API.

So far, Xcerpt has mostly been used as a stand-alone tool for data extraction and analysis. At time the only possibility to execute Xcerpt programs is via a console program, which is not the best starting position for a query language.

With an Xcerpt API all software applications using Xcerpt queries can be designed independently from a concrete Xcerpt engine, because the usage is clearly defined by the interfaces of the Xcerpt API. An Xcerpt API may help to increase the popularity of Xcerpt, as it gets the attention of a wide range of users.

3.1.2 Contribution

The aim of this chapter is to provide an object oriented Xcerpt API. First, an abstract specification of the Xcerpt API is given, and some requirements for the interfaces and methods are established, which have to be adhered to by all concrete implementations of the API. A general architecture of the Xcerpt API is designed followed by a definition of various usage possibilities (of the Xcerpt API).

A concrete implementation of the Xcerpt API in the programming language Java is given for the Xcerpt prototype engine, which is written in Haskell. The usage of this concrete API implementation is demonstrated by use cases, and tests are conducted for the correctness of several parts and for the performance of the API.

3.1.3 Outline of this Chapter

This chapter contains seven sections.

For the purpose of specifiying an Xcerpt API other similar query APIs are analyzed in Section two, which gives an overview of such related work. This overview contains a set of APIs for popular query languages. Learned lessons from these APIs are pointed out and motivate to take over some features for the specification of the Xcerpt API.

Based on the lessons learned an Xcerpt API specification is described in Section three, which is the main part of this chapter. It explains the requirements and the architecture of the API, which are to be followed by each concrete implementation of the API. It also represents an elaborate API specification by describing its main interfaces and methods of the Xcerpt API. At last, various usage possibilities of the Xcerpt API are pointed out by code examples.

As the aim of this chapter does not only consist of the specification of an Xcerpt API, a concrete implementation of the specified Xcerpt API is in the focus of Section four. The implementation is written in the programming language Java, and uses the Xcerpt Haskell prototype engine. It is structured after the corresponding architecture that is specified by the Xcerpt API.

As proof-of-concept of the API specification and implementation, Section five shows some use cases for the Xcerpt API. The focus of the use cases lies on an ad hoc implemented Web application, which serves several purposes, such as demonstrating usage of the Xcerpt API, providing a Web service (supporting

machine-to-machine interactions) and a Web interface. The Web application works independently, no matter which concrete Xcerpt API implementation is used.

Section six is dedicated to testing the Xcerpt API implementation. It is classified into conformance and performance tests. With the conformance tests the correctness of specific parts of the API implementation shall be shown and the purpose of the performance tests is to clarify that the performance overhead occurring in the Xcerpt API implementation is acceptable.

Finally, section seven summarizes this chapter and gives perspectives for further work on the Xcerpt API.

3.1.4 Query Languages

With query languages one can retrieve information from a collection of data. Query languages can be generally classified into two groups: database query languages and information retrieval *IR* [82] query languages. The first group includes very popular query languages, such as SQL, XQuery, or XSLT for example. Since XML is used increasingly in the web and database world, there are more and more query languages that deal with XML data. The second group, IR query languages, deal with weighting and ranking, "relevance-orientation", semantic relativism and logic-based probabilism. An example of an IR query language is *Google*'s query interface based on keywords and phrases or the Common Query Language (CQL), a formal language for representing queries to Information Retrieval systems such as web indexes, bibliographic catalogs and museum collection information. We first look at some of these languages to establish their basic capabilities. For that, we compare XQuery and Xcerpt, since we are going to base some of the design decisions for the Xcerpt API on XQJ (XQuery API for Java). Section 3.1.5 introduces XQuery, the XML Query Language developed at W3C, which is by now a W3C recommendation. Finally, Section 3.1.6 introduces Xcerpt, a novel declarative and rule-based query and transformation language for the (semantic) web.[1]

3.1.5 XQuery

The XML Query Language, XQuery, is a query language for extracting and transforming information from XML documents and is a W3C recommendation since January 23, 2007. XQuery makes use of the XML selection language XPath for addressing specific parts of XML documents in a navigational manner. The next section describes the basics of XPath followed by an introduction to XQuery expressions is given. Finally, the XQuery Data Model (XDM) is explained, which is the common data model for both, XPath and XQuery.

XPath The XML Path Language, XPath [76], is a W3C recommended selection language for XML, and is mainly used in host query languages like XQuery or XSLT. XPath expressions are used to address nodes in XML trees. The central construct in XPath is the *path expression*, which consists of several so called *location steps*. In a path expression, location steps are separated by '/' and consist themself of the following parts:

1. an *axis specifier*, specifying the *direction* of the step's navigation
2. a *node test*, specifying a condition that must be fulfilled by nodes to which the step is navigating
3. zero or more *predicates*, which further filter the set of nodes to which the step is navigating

[1] Some examples in this section are examples used from [67].

The next table shows all axis specifiers usable in location steps. Afterwards the node tests are shown and described.

Axis specifiers	attribute	contains the attributes of the context node
	child	contains the children of the context node
	descendant	contains the descendants of the context node
	descendant-or-self	contains the context node and the descendants of the context node
	following	contains all nodes that are descendants of the root of the tree in which the context node is found, are not descendants of the context node, and occur after the context node in document order
	following-sibling	contains the context node's following siblings, those children of the context node's parent that occur after the context node in document order
	self	contains just the context node itself
	parent	returns the parent of the context node, or an empty sequence if the context node has no parent
	ancestor	is defined as the transitive closure of the parent axis; it contains the ancestors of the context node (the parent, the parent of the parent, and so on)
	ancestor-or-self	contains the context node and the ancestors of the context node; thus, the ancestor-or-self axis will always include the root node
	preceding	contains all nodes that are descendants of the root of the tree in which the context node is found, are not ancestors of the context node, and occur before the context node in document order
	preceding-sibling	contains the context node's preceding siblings, those children of the context node's parent that occur before the context node in document order

Node tests	attribute()	matches any attribute node
	comment()	matches any comment node
	$\langle label \rangle$	matches any element node with name equal to label
	node()	matches any node
	$*$	matches any element node, dependend on the axis specifier

Besides the axis specifier and the node test (which are separated by '::'), a location step may contain predicates, which are enclosed by square brackets. The predicate contains conditions, which must be fullfilled by the nodes selected by the location step. Several possibilities exist to specify conditions in predicates. For example, they may be value comparisons, node existence tests or boolean conjunctions and disjunctions.

A location step is evaluated on a sequence of nodes, which is the result of the application of a previous location step. For each node in that sequence, the location step is executed separately. In the application of a location step on such a node, that node is called context node. Starting from the context node, the location step navigates to all nodes reachable with the specified axis specifier and node test, which results in a new sequence of nodes. If the location step contains a predicate, the resulting sequence is further filtered.

The execution of a path expression's last location step yields the result for the path expression. The result is always a (possibly empty) sequence. This result sequence may not only contain nodes, but can also contain primitive data. A more detailed description of XPath's data model is given in Section 3.1.5.

XQuery Expressions This section introduces several kinds of XQuery expressions, which can be used to select data, iterate over data and to construct new data. All examples shown in this section consider the input XML document shown in Listing 3.1.

Listing 3.1: Input XML document for examples

```
<?xml version="1.0"?>
<faculty fID="informatics">
      <courses term="summer">
            <course>
                  <title>Analysis and Design of Algorithms</title>
                  <lecturer>
                        <name>Prof. Dr. Abc</name>
                        <phone>123456</phone>
                  </lecturer>
            </course>
            <course>
                  <title>Databases</title>
                  <lecturer>
                        <name>Dr. Xyz</name>
                        <phone>456241</phone>
                  </lecturer>
                  <assistant>
                        <name>Assist Ant</name>
                        <phone>548976</phone>
                  </assistant>
            </course>
      </courses>
      <courses term="winter">
            <course>
                  <title>Introduction to Robotics</title>
                  <lecturer>
                        <name>Prof. Dr. Abc</name>
                        <phone>123456</phone>
                  </lecturer>
            </course>
      </courses>
</faculty>
```

The selection of XML data is done with XPath expressions. A simple XQuery expression selecting specific nodes in an XML document consists of just the appropriate XPath expression. Hence, every XPath expression is also an XQuery expression (but not vice versa).

For constructing new data, *constructor* expressions are used in XQuery. With constructors one can construct attribute, comment, document, element, text and processing instruction nodes. A simple example for a constructor expression is `<someElement>text content</someElement>`. The constructor expression in this example looks exactly like the data it constructs.

XQuery supports embedding further XQuery expressions within constructor expressions. These subexpressions must be enclosed within curly braces. The occurrence of such subexpression within a constructor expression is replaced by the result of the evaluation of the subexpression.

The *FLWOR* (pronounced "flower") expression has a central role in XQuery. It is analog to the SELECT **FROM** WHERE clause in SQL. The word FLWOR is an acronym made up of the initial letters of the words: *F*or, *L*et, *W*here, *O*rder by and *R*eturn. A FLWOR expression is used for iteration over sequences and to bind intermediate results to variables.

For the next example consider that we want to extract and list the lecture titles from the XML document shown in Listing 3.1. The XQuery expression for this rearrangement is shown in Listing 3.2 and the achieved result of this transformation can be seen in Listing 3.3.

Listing 3.2: XQuery expression to extract and list lecture titles

```
<summerlectures>
{
   for $c in /descendant::courses[attribute::term='summer']/child::course
   return
      <lecture>
      {
         string($c/child::title)
      }
      </lecture>
}
</summerlectures>
```

Listing 3.3: Resulting list of lecture titles for the summer term

```
<summerlectures>
   <lecture>Analysis and Design of Algorithms</lecture>
   <lecture>Databases</lecture>
</summerlectures>
```

In Listing 3.2, each course that is offered in a summer term is written to the output within an outermost element <*summerlectures*>. The element node <*course*> is renamed by <*lecture*> and its only child is the text node() of the former <*title*> element. With the *for* construct the variable *$c* is bound to a course element, that is offered in summer in each iteration step. The *return* part constructs an outer element (for each variable binding) and its content. A *return* clause may also contain a nested XQuery expression like in the former example, or it just returns literals.

WHERE is optional (as well as **LET** and **ORDER BY**). The *where* clause can be replaced by predicates or if-clauses (or vice versa) in the *for* clause. In the example above it may be:

Listing 3.4: Replacing predicates by using where clause

```
for $c in /descendant::courses/child::course
where $c/parent::courses/attribute::term = 'summer'
   return ...
```

ORDER BY orders the variable bindings from the *FOR* part by the given ordering. *ORDER BY* is followed by an XPath expression that selects the nodes by which the ordering is done. For instance the courses may be ordered alphabetically by their titles:

```
for $c in /descendant::course
order by $c/child::title
```

XQuery/XPath Data Model: XDM Before an XQuery program can be processed, the input has to be represented by an XDM (XQuery/XPath Data Model) instance. An XDM instance can be obtained from an input XML document in several ways. One way is to parse the input document with an XML parser, that generates an *Information Set*. This information set is then validated against relevant schemas. So a *Post-Schema Validation Infoset* (PSVI) is created. Then with this abstract information structure an XDM instance is constructed. In XQuery (and XPath) every XML document is represented by an ordered tree. Document order means for example that the *root node* is the first node of the document. Another constraint for the document order, among other three, is, that every node occurs before its children and descendants. The data model contains seven kinds of nodes: document, element, attribute, text, namespace, processing instruction, and comment. There are *Accessor*s defined for each node type. These are not functions for the user, but for the data model to gather information for each item. For example the document node has a children accessor that returns the sequence of children of this document node. Because of completeness all accessors are defined for each node. So sometimes an empty sequence is returned. Every value in the data model is an ordered sequence with zero or more items. There is no difference between a singleton (a sequence containing only one item) or the item itself. Items are tuples of values and types, they can be either a node or an atomic value. A sequence cannot be nested. The data model represents not only the input and output of an document, but also intermediate values during processing at runtime. The data model specifies 23 predefined types for the items. But there is the possibility of user or implementation defined types also.

3.1.6 Xcerpt

This section introduces Xcerpt, a rule based query and transformation language for the Web. Xcerpt arises from the PhD chapter of Sebastian Schaffert at the Ludwig-Maximilians University of Munich. By now Xcerpt is part of the EU FP6 Network of Excellence REWERSE and is actively supported. In contrast to the path-based approach of query languages like XSLT or XQuery, Xcerpt is pattern based. Another special aspect of Xcerpt is, that query and construction parts are separated strictly. The following parts provide an insight into the language Xcerpt with examples, that illustrate the potentials and usage of Xcerpt.

Term Specifications In Xcerpt three term specifications are provided, *data term*s, *query term*s and *construct term*s, which all have the same syntax.

An Xcerpt data term represents semistructured data and is the input of an Xcerpt program. The input can be given in the Xcerpt specific syntax or in any semistructured data format (e.g. XML or RDF). In Xcerpt, term specifications can be ordered or unordered. This is indicated by square brackets or curly braces. Listing 3.5 shows the same input document as in Listing 3.1. But in contrast the input in Listing 3.5 is written in Xcerpt's data term syntax.

Listing 3.5: Input in Xcerpt's data term syntax

```
faculty [
      attributes { fID { "informatics" } },                                    2
      courses [
            attributes { term { "summer" } },                                  4
            course [
                  title [ "Analysis and Design of Algorithms" ],               6
                  lecturer [
                        name [ "Prof. Dr. Abc" ],                              8
                        phone [ "123456" ]
```

```
                ]
          ]
          course [
                title [ "Databases" ],
                lecturer [
                      name [ "Dr. Xyz" ],
                      phone [ "456241" ]
                ]
                assistant [
                      name [ "Assist Ant" ],
                      phone [ "548976" ]
                ]
          ]
     ]
     courses [
          attributes {term {"winter"}},
          course [
                title [ "Introduction to Robotics" ],
                lecturer [
                      name [ "Prof. Dr. Abc" ],
                      phone [ "123456" ]
                ]
          ]
     ]
]
```

As one can see in the example above, two different brackets are used in the input document. They indicate whether a data term consists of ordered or unordered set of subterms. Square brackets denote an ordered sequence of subterms in a term, that is the subterms' order is significant[2]. So in this example the course element's children are always ordered, that is the title element occurs *before* the lecturer element. Curly braces however denote an unordered set of subterms, that is the order of the subterms is insignificant. In Xcerpt attributes have no special treatment like in XML, they are grouped in an *attributes* labeled subterm, that is the first subterm in the set of subterms regardless of the set is ordered or unordered. For compatibility with XML, *attributes* in Xcerpt has always an unordered term specification.

This paragraph deals with the second type of terms, the query term. A query term is a (possibly incomplete) pattern, that queries data terms. The incompleteness of such a pattern can be in depth or in breadth respective to the graph structure induced by the given data term. An incompleteness in depth is expressed by a descendant construct, i.e. *descendant d* matches with all d elements at arbitrary depth. An incompleteness in breadth is expressed by partial term specifications. A partial term specification, expressed by double curly braces (double square brackets, resp.) matches with the data term with possibly other elements (with no other elements, resp.) beside those elements matched by the query term. A total term specification however is expressed with single brackets as explained in the previous paragraph. Query terms may also contain variables for selecting data. They are expressed by using the keyword *var* followed by an identifier. In Xcerpt there are four types of variables:

[2] An order is needed for accessing the data by position

Label variables	occur in place of a label in a query term
Namespace variables	occur in place of namespace prefixes in a query term
Term variables without restriction	can be bound to any subterms
Term variables with restriction	are bound only to subterms, that match with the pattern they are restricted to

The following examples take a closer look to the query terms of Xcerpt.

Listing 3.6: Query term in Xcerpt

```
courses {{
      var C →course {{
            title [ var T ]
      }}
}}
```

This query matches courses with at least one course element that has in turn at least one title element. This example shows two different notions of variables. The first one (*var C*) is a variable with restriction, it binds to course elements with at least one title element. The second one (*var T*) is a variable without any restrictions. This query is partial, that means that a data term with any additional subterm to courses matches with this query term also.

Listing 3.7: Query term using *without*

```
courses {{
      course {{
            title { var T ],
            without lecturer [ "Prof. Dr. Abc" ]
      }}
}}
```

The query term in the upper example retrieves all titles of courses which are not lectured by *Prof. Dr. Abc*. For such cases of negation, Xcerpt provides the *without* construct. It is only applicable on subterms and makes sense to be used in partial term context, for an ordered sequence of subterms is not required for a negation.

Rules Rules are essential elements of Xcerpt programs, which consist of at least one rule. They are similar to functions in other programming languages or views in relational databases. With rules one can retrieve different representations of source data from the Web. A rule is of form as shown in Listing 3.8

Listing 3.8: Rule in Xcerpt

```
CONSTRUCT
   <construct part>
FROM
   <query part>
END
```

A rule relates construct terms with query terms. The `<construct part>` constructs the result of a rule and the `<query part>` contains a query term, that can be an *and* or *or* connection of many query terms. All variables, that occur in the `<construct part>` must also occur in the `<query part>`. The input of a rule is always a data term (more than one resources are connected with an *and* or *or* operator). The resource specification may refer to external resources or to the output of other rules, for the result of a

rule is also a data term. This process of querying the result of other rules is called *rule chaining*. Rule chaining enables to have complex and recursive programs and has the advantage to break down such complex queries. Xcerpt provides also the possibility to group the constructed terms. This is done by the keywords **`all`** and **`some`**. Listing 3.9 demonstrates boolean connectives in queries and Listing 3.10 shows how explicit grouping is done with the *group by* construct.

Listing 3.9: Boolean connectives in queries

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

Listing 3.10: Explicit grouping with “group by”

```
table [
      all tr [
      all td [
            var Value ] group by { var Row }
      ] group by { var Col }
]
```

A special rule is the *GOAL* rule. Every Xcerpt program has to have at least one GOAL. Unlike construct-query rules, GOALs have an output specified. If no output resource is specified, the standard output is implicitely assumed to be the output resource. Example 3.11 shows a GOAL with explicit output resource specification and Example 3.12 shows a GOAL which writes the result to the standard output.

Listing 3.11: Goal writing its result to the specified file

```
GOAL
      out {
            resource [ "file:prices.html", "html" ],
            html [
                  head [ title [ "Price Comparison" ] ],
```

```
                body [ var Content ]
            ]
      }
FROM
      var Content →table {{ }}
END
```

Listing 3.12: Goal writing its result to the standard output implicitly

```
GOAL
      html [
            head [ title [ "Price Comparison" ] ],
            body [ var Content ]
      ]

FROM
      var Content →table {{ }}
END
```

3.2 Related APIs

For the purpose of specifying an Xcerpt API, other popular APIs for query languages are analyzed first. This section gives an overview about such APIs. With this overview lessons are to be learned about what features make up a well defined API and which ones can be taken over into the Xcerpt API.

3.2.1 JDBC

The Java Database Connectivity (JDBC) [72, 62, 30, 14] is an API for the Java programming language for accessing (relational) databases, and is part of Java SE since version 1.1. With JDBC programmers can use SQL statements for querying and updating data in databases. It is desirable that these statements are sent to databases in a platform independent manner. As there are many different commercial and non-commercial Database Management Systems (DBMS), there must be different implementations of the interfaces specified in the JDBC API, which are called JDBC Drivers. So, the same Java program can be used with several DBMS just by switching the JDBC Driver, the Java code remains unchanged. The JDBC Driver, usually provided by the specific DBMS, converts statements in the Java program to a DBMS specific protocol.

The following sections show what JDBC consists of, and how the API can be used in a Java program. Section 3.2.1.1 shows how JDBC Drivers are initialized and establish a connection to the DBMS. In Section 3.2.1.2, different kinds of statements are introduced and their execution is explained. Section 3.2.1.3 finally shows how results of the statements are returned to the Java program and how these results can be processed.

3.2.1.1 Initializing and Connection

As there are many different JDBC Drivers, there must be ways for selecting specific ones. For better generality the selection should occur dynamically at program runtime. This way the programmer can choose to switch to other DBMS at any time without changes to the program code. Furthermore there

may be the need to use more than one DBMS at the same time. The selection of drivers is managed by the JDBC Driver Manager.

One of the essential JDBC classes is the `Connection` class. This class is in fact an interface and hence cannot be instantiated directly. JDBC Drivers are responsible for providing implementations of this interface. The Java programmer can create a `Connection` instance by using one of the JDBC Driver Manager's factory methods. The Driver Manager will return an object of a specific `Connection` implementation, depending on the chosen JDBC Driver.

Listing 3.13 shows how a JDBC connection is established. First the JDBC Driver (here a MySQL driver) is loaded by calling `Class.forName(''com.mysql.jdbc.Driver'')`. After loading the driver a connection is established using the `DriverManager`'s `getConnection()` method. Its arguments in this example are the URL of the database and a user-password pair. The programmer is responsible for catching any exceptions which may be thrown by the `getConnection()` method when errors occur. When work with the Connection object is finished, all resources must be closed using its `close()` method.

Listing 3.13: Establishing a JDBC Connection

```
try {
   Class.forName("com.mysql.jdbc.Driver");
   Connection con = DriverManager.getConnection(
         "jdbc:mysql:///dbtest",
         "user", "password");
} catch (SQLException e) {
   handle(e);
} finally {
   con.close();
}
```

Once the connection to a specific DBMS is established, the `Connection` instance can be used for querying information about the DBMS. For example the programmer can ask for database capabilities or supported SQL versions. The `Connection` instance is also used for creating statements and sending them to the DBMS for execution. `Statement` is another essential class of the JDBC API. Like Connection, it is an interface, and factory methods are used for instantiation of `Statement` objects. The creation of a `Statement` is done via the `createStatement()` method of a `Connection` instance, as is shown in Listing 3.14

Listing 3.14: Creation of a JDBC Statement

```
Statement stmt = con.createStatement(
      ResultSet.TYPE_SCROLL_INSENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
```

The `Statement` object is not yet linked with a specific SQL query or update. In fact, the same statement object can be used with different SQL queries. The existence of the `Statement` interface in JDBC is justified by the fact, that there are subinterfaces of `Statement`(`PreparedStatement`, `CallableStatement`), with different execution behavior. This is shown in Section 3.2.1.2. The `createStatement()` method in the previous listing accepts two optional arguments indicating the behavior of results of the statement execution. This is shown in more detail in Section 3.2.1.3.

3.2.1.2 Executing Statements

Statements in JDBC can be classified in different ways. Query statements return row result sets and update statements return the number of rows, which have been updated by the execution of the statement. JDBC furthermore distinguishes between regular statements, prepared statements, and callable statements. Listing 3.15 shows the execution of a regular JDBC statement.

Listing 3.15: Executing a JDBC Statement

```
try {
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT Name, Age FROM PERSONS");
   processResults(rs);
} finally {
   stmt.close();
   rs.close();
}
```

The `Statement` object can be used for executing one or more SQL queries (or updates). Processing of results in the result set is shown in Section 3.2.1.3. This example executes the query `SELECT Name, Age FROM PERSONS`.

Some DBMS support precompilation of SQL statements. This makes sense whenever SQL statements, with possibly different parameters, are executed multiple times. The SQL statement is then sent to the database only once, and can be precompiled. For subsequent executions of the statement, the user needs to provide the new execution arguments only. This is done in JDBC as shown in Listing 3.16.

Listing 3.16: Precompiled (prepared) statements in JDBC

```
PreparedStatement pstmt = con.prepareStatement(
      "UPDATE PERSONS SET Age = ? WHERE Name = ?");

pstmt.setInt(1, 30);
pstmt.setString(2, "John");
pstmt.execute();

pstmt.setInt(1, 40);
pstmt.setString(2, "Jack");
pstmt.execute();
```

In this example a prepared statement is created instead of a regular statement. In contrast to regular statements, prepared statements are directly linked to SQL queries (or updates). Here a parameterized SQL update is used. Whenever the statement is to be executed, parameters for the statement are set using one of several set methods. The first argument in the set method is the number of the parameter in the prepared statement. In this example the same prepared statement is used twice, setting John's age to 30 and Jack's age to 40.

JDBC also allows the execution of procedures, which are stored in the DBMS. Stored procedures represent logical units, they are sets of queries which belong together and perform a specific task. Stored procedures are executed using `CallableStatement` instances. The usage of this class is very similar to the previous examples, and are not further discussed here.

3.2.1.3 Answer/Result API

The last one of the important JDBC interfaces is `ResultSet`. A result set is usually generated by executing a query statement, and represents the result table of the database query. It defines many methods for accessing rows and columns of the result. Very noticeable is JDBC's capability of retrieving result rows iteratively. In JDBC this capability is expressed by the so called *cursor*. The cursor of a `ResultSet` object is a pointer to the current row in the result table, and is initially positioned before the first row. JDBC's cursor capabilities allow moving forward and/or moving in any direction relative and absolute to the current cursor position. It is also possible to update the data directly in the result set, which then is reflected to the database. Support of these capabilities is DBMS dependent, the programmer must query the database capabilities via the `Connection` instance.

In Listing 3.17 a regular statement is created with the `TYPE_FORWARD_ONLY` and `CONCUR_READ_ ONLY` options. The cursor of the result set hence can only move forward, and the result set cannot be used for updating data in the database. That means that the user can iterate through the result set only once and only from the first result to the last result. The result set's `next()` method moves the cursor forward one step and returns whether there are more results. The `getInt()` and `getString()` methods of the `ResultSet` object are used for retrieving column data inside the current row.

Listing 3.17: Processing JDBC Query Results

```
Statement stmt = con.createStatement(
      ResultSet.TYPE_FORWARD_ONLY,                                    2
      ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT Name, Age FROM PERSONS");    4
while (rs.next()) {
      String name = rs.getString("Name");                             6
      int age = rs.getInt("Age");
      processOnePerson(name, age);                                    8
}
```

The result set has various methods for retrieving the data inside a column of the current result. The column data can be retrieved either by using the column's index or its name. There are also different methods for directly converting the column's data types to Java specific types. In the previous example the data is retrieved as int and String, but there are methods for other datatypes also.

Next, the creation of updatable and scrollable results is shown in Listing 3.18. The cursor is positioned on the second result in the result set, which then is updated with a new name. With the call to `rs.updateRow()` the changes are applied to the database.

Listing 3.18: Advanced result processing in JDBC

```
Statement stmt = con.createStatement(
      ResultSet.TYPE_SCROLL_INSENSITIVE,                              2
      ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT Name, Age FROM PERSONS");    4

rs.absolute(2);                                                       6
rs.updateString("Name", "John");
rs.updateRow();                                                       8
```

The programmer is responsible for closing result sets after using them. This is done with the `close()` method of the result set instance. In the case that exceptions are thrown during result processing, the programmer must take care of catching these exceptions.

3.2.2 XQJ

This section describes XQJ (XQuery API for Java), an application programming interface for the Java programming language, designed for executing queries written in the XQuery language. XQJ is still in development under the Java Community Process [45]. As SQL, XQuery is a query language, but in contrast, it is not limited to querying databases. XQuery is designed to query both, XML databases and XML documents, a property that is carried over to the XQJ API. The result of an XQJ query is always a sequence of result items. These items may have different item types, such as *boolean*, *string* or *node*. If the result type is node, it can be accessed using one of different XML representation APIs, such as DOM (see Section 3.2.3), SAX or StAX. The structure of the XQJ API is very similar to that of the JDBC API (described in Section 3.2.1). A typical XQJ application consists of following parts:

1. connecting to an XML datasource and a specific XQuery engine
2. creating an expression (or prepared expression) from the connection
3. possibly binding values for external variables used in an expression (or prepared expression)
4. executing the expression (or prepared expression)
5. processing the result
6. optionally repeating steps 2-6 as needed, and
7. closing the connection, expressions and results and thereby freeing all resources

The following sections describe the most important parts from above. Section 3.2.2.1 shows how a connection to a specific datasource and XQuery engine is established. In Section 3.2.2.2, creation and execution of XQJ *Expression*s is explained. Section 3.2.2.3 finally describes how results of the query are returned to the Java program and how these results can be processed.

3.2.2.1 Initializing and Connection

As in JDBC, the XQJ API provides means of connecting to a datasource, and queries cannot be executed without having established a connection. An XQJ datasource is an abstraction over any data, for which an XQJ query may make sense. For example, this can be relational data in databases, XML documents in the WEB or XML documents on the local file system. The support for different datasource kinds is vendor dependent. Different XQJ engine vendors may support connections to different datasources. This is similar to the JDBC Driver concept, where different Driver implementations provide different capabilities, but goes a bit further in the spirit of XQuery's ability to query not just databases but also other XML storages.

In the XQJ API, the `XQDatasource` interface is the representation of an XQJ datasource. Obtaining an `XQDatasource` object may happen through different means, which are vendor dependent. Once obtained, it can be used for establishing a connection and retrieving an `XQConnection` object, which is the counterpart of the JDBC `Connection` object (see Section 3.2.1.1). XQJ query expressions and result sequences only exist in the context of an XQJ connection. A connection has the following properties, which affect expression and results obtained in the context of the connection:

Scrollability indicates whether the sequences can be read in a forward only manner (one can iterate through it only once and only from the first item to the last), or if the produced sequences are scrollable.

Updatability indicates whether the returned result sequences are updatable or not [3]

Holdability indicates whether the result sequence can be held beyond the transaction in which the expression was executed

Listing 3.19 shows how a connection with a datasource can be established. Assume that the `datasource` variable is a reference to an `XQDataSource` instance. The scrollability and updatability properties of the connection can be set using appropriate setter methods. Calls to any methods in the XQJ API may yield an `XQException`, which has to be catched. Finally, the connection has to be closed, to free all related ressources. This is done in the finally block, to ensure that the ressources are freed also in the case of an exception.

Listing 3.19: Establishing an XQJ XQConnection

```
try {
   XQConnection connection = datasource.createConnection();
   connection.setScrollability(XQConstants.SCROLLTYPE_SCROLLABLE);
   connection.setUpdatability(XQConstants.RESULTTYPE_READ_ONLY);

   //do something with connection
} catch (XQException e) {
   handle(e);
} finally {
   connection.close();
}
```

3.2.2.2 Creating and Executing Expressions

Query expressions are created using an `XQConnection` object. The expressions always exist in the context of a connection. If the connection is closed, all related expressions implicitely are closed too. Like JDBC, XQJ distinguishes between ordinary expressions and prepared expressions. In the XQJ API these are represented by the interfaces `XQExpression` and `XQPreparedExpression`. Furthermore, XQJ also provides means for defining external variables for the queries.

Listing 3.20: Creating XQJ Expressions

```
XQExpression expression = connection.createExpression();
XQResultSequence results = expression.executeQuery("for $i in (1,6,3) return $i");
//process results
```

Listing 3.20 shows the creation and execution of a simple query, which results in the sequence containing the numbers 1, 6 and 3. Note, that the execution of an expression always results in a sequence of result items, but the type of the particular items can be different. The XQResultSequence can be used for processing the result items, which is shown in Section 3.2.2.3. The next listing demonstrates the use of prepared expressions and external variables.

[3] At time of writing, there is no support for updatable result sequences in the XQJ API and this property is always set to `READ_ONLY`

Listing 3.21: Creating XQJ expressions

```
String query = "declare variable $j as xs:int external; "
          + "for $i in (1,6,3) return $i+$j";
XQPreparedExpression preparedExpression = connection.prepareExpression(query);

preparedExpression.bindInt(new QName("j"), 5);
XQResultSequence results = preparedExpression.executeQuery();
//process results

preparedExpression.bindInt(new QName("j"), 10);
results = preparedExpression.executeQuery();
//process results
```

In this example, the query declares an external variable `j` as type `xs:int`, and then produces the result sequence `(1+j,6+j,3+j)`. The method invocation `connection.prepareExpression(query)` creates a prepared expression, which then is executed twice, binding the external variable `j` to different values. The binding is done using the `bindInt(QName, int)` method.

The advantage of using precompiled expressions is that the programmer can use the same expression and supply it with different values each time he executes it. With this approach the execution time can be reduced and the probability of mistakes in several expression that are *string manipulated* is also reduced.

3.2.2.3 Answer/Result API

The result of executing an XQuery expression is always a sequence of result items (may be an empty sequence also). These items may be of different types, such as `int`, `boolean`, `string` or `node`. XQJ provides an *item accessor* facility for accessing result items of different types. In the case of the *node* type, the item can be accessed using several XML processing APIs, such as DOM, SAX or StAX. Result sequences in XQJ can be *scrollable* or *forward only* and *updatable* or *read only*, depending on the properties of the connection, in which context the result sequence is created. Like expressions, results are dependent on their context connection: if the connection (or the expression) is closed, the result will be closed also. Assume the XQuery expression `for $i in doc("contacts.xml")/descendant::phone/child::text()return $i`. This query results in the sequence of all phonenumbers in the `contacts.xml` document. In Listing 3.22, the method `getQuery()` returns this XQuery expression. Using an `XQExpression` object, this query is executed. The result sequence then is used for iterating all phonenumbers. Because the result is a sequence of integers, the `getInt()` method can be used to retrieve the result items.

Listing 3.22: Processing an XQJ Result Sequence

```
String query = getQuery();
XQResultSequence result = expression.executeQuery(query);

while (result.next()) {
   int phoneNr = result.getInt();

   //do something with phone number
}

result.close();
```

For the next example, assume that an XQuery expression results in a new XML document. The result sequence hence contains one item only, and the type of that item is *node*. Listing 3.23 shows how this item can be retrieved using DOM as XML representation API.

Listing 3.23: Processing an XQJ result item with result type *node*

```
XQResultSequence result = expression.executeQuery(query);

result.next();
Node domNode = result.getNode();

result.close();
```

At the beginning, the cursor of the result sequence is positioned *before* the first result item. Hence invoking `result.next()` advances the cursor to the first result (which is also the only result). Then, `result.getNode()` retrieves the result, represented as a DOM `Node`.[4] Other possibilities range from writing the result to a stream to forwarding the result to a SAX content handler or using the StAX API.

3.2.3 DOM

The Document Object Model (DOM) [42] is a W3C standard for representing XML documents as a tree structure. DOM defines an API for object oriented programming languages to access, navigate, and update XML document trees. It is not designed towards a specific programming language, but rather can be used in every language for which an appropriate DOM implementation exists. The most common use case for the DOM API is the dynamic access and update of XML documents representing webpages.

The DOM API includes a hierarchy of classes for representing the XML document tree. These classes contain methods for navigating (forward and also backward) in the document tree and accessing or updating stored data. Navigation and update is the DOM API's primary use, and navigation to arbitrary nodes is very frequent. Hence the whole document tree has to be stored in memory when DOM is used.

The class hierarchy, representing the document tree, consists of several classes for each node type (see Figure 3.1). Node types are for example elements, attributes and text. The following sections describe some of these classes, showing their usage with examples. In the examples the DOM tree shown in Figure 3.2 is assumed to be represented by a `Document` instance. It is the representation of the XML document shown in Listing 3.24.

Listing 3.24: XML document represented by the DOM tree shown in Figure 3.2

```
<members>
   <member>
      <memberid>100</memberid>
      <name>John</name>
   </member>
   <member>
      <memberid>101</memberid>
      <name>Jack</name>
   </member>
</members>
```

[4] In fact, methods for accessing result items are not specified in the `XQResultSequence` interface, but rather are specified in one of its superinterfaces, the `XQItemAccessor`

3.2.3.1 Interface Document

The root of a document tree is always an instance of the `Document` interface. It does not represent the first element tag in an XML document, but rather represents a root node, whose child is the first element node. `Document` instance can hence be seen as the representation of the whole XML document. `Document` object provides methods for accessing any nodes in the tree, e.g. by id or by tag name. `Document` also provides factory methods for creating new nodes of any type. The creation of new nodes is needed, when the document tree shall be updated.

In Listing 3.25 the variable `doc` references an `Document` instance representing the DOM tree shown in Figure 3.2. Assume we want to change how `memberid`s are stored in the XML Document. They shall be stored as attributes of the `member` element, instead of child elements. Since there are no attributes in the document yet, we first have to create them using the `createAttribute(String name)` method of the `Document` instance. This is done once for every `member` in the document. Using the `getElementsByTagName(String name)` method, we get a `NodeList` containing all elements in the document with the label `member`. The remainder of our goal is postponed to a later listing, as we do not yet know how to use `Element` and `Attr` objects.

Listing 3.25: Navigating to and creating new nodes

```
NodeList members = doc.getElementsByTagName("member");

for (int i = 0; i < members.getLength(); i++) {
   Node member = members.item(i);
   Attr attribute = doc.createAttribute("memberid");

   //remainder omitted
}
```

3.2.3.2 Interface Node

This interface represents the primary datatype in DOM. All node types are direct or indirect subinterfaces of this base interface (also `Document` is a subinterface of `Node`). The `Node` interface defines methods for navigating to nodes in a relation to a given node. For example methods exist for getting all children and the parent of a node. Since all node types are subtypes of node, this is not unproblematic, as some node types (like text or attribute) do not have children. Trying to get the children of such nodes hence yields in a runtime exception. `Node` also defines update methods, which allow setting the value of the node and appending new children to a node. As before, these methods may throw an exception, if the nodetype does not support them.

Returning to our goal of the previous example, we still have to add the new attributes to the `member` nodes. We furthermore have to remove the child elements labeled `memberid`. The `Node` interface does not provide us with means of completing our goal, but we can get one step closer. In the following listing, assume that the variable `member` is defined as shown in Listing 3.25.

Listing 3.26: Removing of nodes in the document tree

```
Node child = member.getFirstChild();
member.remove(child);
String memberID = child.getTextContent();
```

Assume, that we have an XML schema, which requires the `memberid` to be the first child of each `member` node. We can then navigate to the `memberid`, using the `getFirstChild()` method, and invoke `remove(Node)` for removing this child node. By removing a child node, that node does not get deleted from the memory. As long, as there are references to the node, one can access the node's data and all its substructure. In our example the `child` variable still references the child node, although it is removed from its parent. The `memberid` node however is not the text content representing the member's ids. It is the element wrapping the text content. We hence use the method `getTextContent()` for retrieving the memberid values and reference them with the variable `memberID`.

3.2.3.3 Interfaces Element, Attr and Text

Some of the more specific node types are `Element`, `Text` and `Attr`. They represent elements, text data and attributes in the XML document. Elements may contain child nodes and may contain attributes. The `Element` interface defines methods for getting and setting the element tag's name, its children and its attributes. The other two interfaces, `Attr` and `Text`, represent node types, which cannot have any children. Attribute nodes in DOM are not considered children of elements, but rather are properties of element nodes. Hence one cannot navigate to a parent node starting from an attribute node, the use of the specific methods will yield an exception.

Our example of replacing elements with attributes is almost finished. We attain our goal by adding the newly created attribute nodes to the `member` nodes. This cannot be done with the `Node` interface, as it does not provide the required methods. Fortunately, our `member` nodes are not only of type `Node`. In fact, they are of type `Element` (a subtype of `Node`). Listing 3.27 shows how we cast the `member` nodes to `Element` and set their attributes. Variables `member`, `attribute` and `memberID` are defined as previously.

Listing 3.27: Setting attributes for element nodes

```
attribute.setValue(memberID);
Element memberElement = (Element)member;
memberElement.setAttributeNode(attribute);
```

The string `memberID` references the text content representing the member's ids. This string is used to set the value of the newly created attributes, using the `setValue(String)` method. After casting the `member` nodes to `Element`, the `setAttributeNode(Attr)` is used for setting the element node's memberid attribute.

3.2.4 More XML processing APIs

This section introduces some more XML related APIs, which can be used for processing XML results of Xcerpt. SAX and StAX are *stream based* processing APIs for XML, they are introduced in Section 3.2.4.1 and Section 3.2.4.2.

3.2.4.1 SAX

The *Simple API for XML* (SAX) [54] was originally designed for the Java programming language, but in the meantime is adopted by a wide range of object oriented programming languages, and is now a *de facto* standard. Using SAX allows users to process XML documents in a *stream based* manner. Unlike DOM, there is no representation of the whole XML document tree kept in memory. The SAX API is designed as an *event driven* model. Users of the SAX API must provide callback methods, which are invoked by SAX when its parser traverses the XML document. SAX is a so called *push* API, which means that the

client program is driven by the parser. An uncomfortable point at *push* APIs is, that the data is delivered continuously to the client, regardless whether the user is ready for that data or not.

The most important interface in the SAX API is the `ContentHandler`. It specifies several callback methods, such as `startDocument()`, `startElement(namespace,localName,qname,attrs)`, etc. These methods must be implemented by the user and are called by SAX upon traversing of appropriate nodes in the XML document. For example, when the SAX parser traverses a new element opening tag, it invokes the user implemented method `startElement` on the arguments *namespace*, *local name*, *qualified name* and *attributes*. The user provided code is executed and on termination the process is returned to the SAX parser, which then continues its traversal of the XML document.

3.2.4.2 StAX

The Streaming API for XML (StAX) [26] is a new generation API for XML processing in Java. Like SAX, StAX allows users processing XML documents in a stream based manner. In contrast to SAX, it is designed on a pull parsing modell. Pull parsing APIs are more comfortable and familiar in the usage than push parsing APIs. The user controls the application flow and asks (pulls) the parser for new information items. Using pull parser APIs often yields a more natural code, than using push parsing APIs.

StAX defines two distinct API sets, called *iterator API* and *cursor API*. The only difference between these two APIs is their complexity. The cursor API of StAX is a low level API, which can be implemented very efficiently. For simple applications it is easier to use this low level API. The iterator API, on the other hand, is a more complex high level API. Using the iterator API allows writing more advanced applications, with the cost of more complexity. These two models are mainly realized by the two interfaces `XMLStreamReader` for the cursor model and `XMLEventReader` for the iterator model.[5]

Both, the cursor and the iterator API are very similar in usage. Both *reading* interfaces provide means for querying whether there are more information items (for example a `hasNext()` method). They also provide methods for *pulling* the next item, first advancing the cursor with a `next()` method, and then getting the current item with appropriate getter methods. Noticeable is the fact, that the user has the possibility to skip several information items, for example by using the method `nextTag()` instead of `next()`.[6]

3.2.5 Lessons Learned for the Xcerpt API

This section describes the features of the previously introduced APIs, that are to be taken over into the Xcerpt API.

Engine transparency One important aspect of the APIs is engine transparency (cf. 3.2.1.1, 3.2.2.1). This means that the API is more general and flexible in the manner that the API user can switch between different engines at runtime without changing the meaning of his programs. An engine transparency also ensures the extensibility of an API.

Iterable results This feature is important for both efficiency and flexibility of a query API. Efficiency is gained by the "on demand" creation of results and flexibility is ensured if the results are *scrollable*.

Various result representations Providing the API user with the flexibility of choosing between various result representations (cf. 3.2.2.3) allowes the user to write efficient programs. The API user can choose the result representation, which fits best into his application.

[5] For writing XML the StAX API also provides the two interfaces `XMLStreamWriter` and `XMLEventWriter`

[6] A freedom users of push parsing APIs do not have

3.3 Xcerpt API

Based on the lessons learned from the other APIs for query languages, this section describes the design of an API (application programming interface) for the query language Xcerpt. The purpose of the API is to ease the use of Xcerpt in widespread object oriented programming languages such as Java. The API is carefully designed not to be tailored towards one specific language and platform but to be language and platform independent. This chapter distinguishes between the abstract API specification (which is described in this section) and the implementation of the API in a specific programming language and for a specific Xcerpt engine (as described in Section 3.4).

This section is structured as follows. Section 3.3.1 describes requirements defined for the abstract API definition and all implementations of the API. Section 3.3.2 discusses architecture (i.e. components and modules) and the principles followed by the Xcerpt API. The abstract API definition consists mostly of interfaces that must be implemented by all implementations of the API (similar to the abstract definition of DOM). Section 3.3.3 specifies all provided interfaces of the Xcerpt API. Finally, different usage modi of the Xcerpt API are described in Section 3.3.4.

3.3.1 Requirements

The Xcerpt API is designed according to the following four general requirements:

Simplicity: The Xcerpt API shall be intuitive and easy to use.

Extensibility: The API shall be easily extensible.

Flexibility: The API shall allow for flexible usage.

Efficiency: The API shall be efficient in both time and space.

Although the satisfaction of these requirements mostly falls into the API implementation's field of responsibility, the abstract API specification must be carefully designed not to hinder API implementations. Hence these general requirements are described in more detail in the next four sections.

Of course, it is not possible to fulfill all of these requirements perfectly and to the same level. For example, the requirements *Simplicity* and *Flexibility* are competing goals. The more flexible a system is, the more complex is the usage of it. Hence the best tradeoff between the requirements must be found.

3.3.1.1 Simplicity

The Xcerpt API shall be intuitive and easy to use, which can be ensured in several ways. The Xcerpt API is designed similar to existing popular APIs for querying languages, because many programmers are familiar with those APIs. This ensures that the API is learned quickly by the majority of programmers.

A premise for a simple API is that it is small. The Xcerpt API provides only a handful of interfaces, capable of dealing with the execution of Xcerpt programs and queries, and handling of results. Thus, the API end users see immediately, which interfaces they have to use in order to achieve their goal.

The Xcerpt API further does not require end users to write "boilerplate" code. Boilerplate code is a problem in many APIs at time. They require the end users to write code, which does not deal with application logic. A characteristic of boilerplate code is that it is repeated in every program using a specific API, no matter what goals that program tries to achieve. The Xcerpt API does not force the programmer to use long code sequences. The API is designed in such a way that most tasks can be performed with a handful of statements.

Another problem of APIs at time is the lack of information about how objects are created for specific interfaces or abstract classes. Often abstract APIs define interfaces and abstract classes only, and do not provide access to concrete implementations. The API end users have to find a way of creating objects for these interfaces, which is often very complicated and badly documented. In the Xcerpt API, the creation of objects for the core interfaces is possible in a very simple manner. The creation of objects is handled centrally by the `XcerptFactory` class.

3.3.1.2 Extensibility

A well designed API has the property that the programs using the API do not have to be changed if the API is extended. The Xcerpt API is easily extensible in this manner. This is ensured for example by modularization in the API design. Section 3.3.3 shows three different modules of the API, each of which deal with different aspects of Xcerpt query execution. The Xcerpt API is furthermore designed with foresight to concrete implementations of the API for specific Xcerpt engines, which may involve several problems. The abstract API architecture is designed with engine specific difficulties in mind, providing enough space for concrete implementations to overcome their obstacles.

3.3.1.3 Flexibility

The Xcerpt API provides flexibility in several ways. First, the API allows the user to execute both, Xcerpt programs and Xcerpt queries. While the execution of Xcerpt programs is feasible in static and large applications, the execution of simple queries is designed more towards small dynamic or interactive applications. The Xcerpt API not only allows obtaining constructed results (for an executed Xcerpt program), but it also provides means for obtaining the substitution sets found for a specific goal inside an Xcerpt program or for a specific standalone Xcerpt query. Different usage modi are provided by the Xcerpt API, ranging from querying input data to querying the results of the execution of an Xcerpt program.

Another flexibility provided to the end user by the Xcerpt API is the freedom to choose where the input data for the Xcerpt programs and queries come from. Besides the regular ways provided by Xcerpt (i.e. defining a regular input resource specification in the Xcerpt program), the API also provides the possibility to define input data in a more API wise manner by using `setInputData` methods. Thus the Xcerpt API can be used, even if the application using the API does not have access to the local file system or to the web.

Flexibility in the navigation through the result set is another important point. Hence the *ResultSequence* interface is designed to allow easy and flexible navigation through the results. The API end user can decide to iterate through every result in the sequence, or he can choose to examine one specific result only.

Desirable is furthermore a flexible representation of results. The results of an Xcerpt program (or query) execution can be represented in different output formats like Xcerpt character stream, XML character stream, DOM nodes or SAX events. For this purpose the Xcerpt API provides several result transformators.

3.3.1.4 Efficiency

Efficiency is mainly in the responsibility of the concrete API implementation. Though the abstract API specification is carefully designed not to hinder an efficient implementation.

The API provides means for accessing the results of an Xcerpt program execution in an iterative manner. By doing this, the API design allows concrete API implementations to construct results on demand. When an Xcerpt program is executed through the API, the API implementation may choose not to obtain all results of the execution (e.g. by configuring the underlying engine, or by executing only parts

of the Xcerpt program). Whenever the API end user accesses one particular result in the result sequence, that result can be obtained lazily. Hence efficiency is gained for both, time and space.

A further aspect of efficiency is the precompilation of queries. Once compiled, the Xcerpt programs and queries can be executed several times, and each execution of the program or query can be associated with different input data. This saves much time, if the API end user is going to execute an Xcerpt program or query very often.

The possibility to choose between different result representations further contributes to API efficiency. If one particular result is large, one would not choose a DOM representation of that result, because this would yield high memory consumption. Instead a stream based representation is sometimes desirable in such cases. A stream based representation of results is provided in the Xcerpt API by a SAX result transformer, which transforms the results into an event stream.

3.3.2 Architecture and Principles

The core principle, the Xcerpt API is based on, is engine transparency. The API is designed carefully to be independent of both language, platform and engine. The user of the Xcerpt API should not know about the various implementations of the API for different Xcerpt engines. A user provided program using the Xcerpt API must run equally on all API implementations, regardless of engine, language or platform.

In order to facilitate this principle in API implementations for less powerful engines (cf. Section 3.4, API Implementation for the Xcerpt Prototype Engine), the API must provide means for overcoming obstacles placed by the engine, language or platform.

In order to help overcoming obstacles in underlying engines, the Xcerpt API specifies three execution phases. The first phase and the third phase are meant to help compensating lacks in the various underlying engines. The execution of every Xcerpt program within the Xcerpt API runs through these three phases. As is shown in Figure 3.3, the phases are called *preprocessing phase*, *engine execution phase* and *postprocessing phase*.

3.3.2.1 Preprocessing

The *preprocessing phase* has the function of preparing all API end user provided input, such as Xcerpt programs, queries and input data, such that the underlying Xcerpt engine can handle them. If the underlying engine has any shortcomings, the API implementation has to do a workaround in the preprocessing phase.

The preprocessing phase furthermore has the function of gathering all input data and produce local copies of them. This way a clear snapshot semantic is defined for the execution of queries, which can be fulfilled by any Xcerpt engine.

3.3.2.2 Engine Execution

After the preparation of the Xcerpt programs or queries and the input data in the preprocessing phase, the Xcerpt API executes the Xcerpt programs or queries in the underlying Xcerpt engine in the *engine execution phase*. This phase takes care of the communication with the underlying Xcerpt engine in the background, without the recognition of the API end user. Problems occurring during the execution of the underlying Xcerpt engine are carried over to the API. If the execution of the underlying Xcerpt engine yields an error message, the execution phase of the API terminates with an exception containing that error message. If the execution of underlying Xcerpt engine does not terminate, the execution phase of the API does not terminate also.

3.3.2.3 Postprocessing

The *postprocessing phase* serves the preparation of the results, which are obtained after the execution phase. The results are brought in an appropriate form and `ResultSequence` and `SubstitutionSet` objects are created and linked with the results. In this form, the results can be accessed by the end user in an API wise manner, as is explained in Section 3.3.3.

3.3.3 API Specification

This section describes the specification of the Xcerpt API, which mostly consists of interfaces that are to be implemented by a specific Xcerpt API implementation. The end user of the API usually does not have to provide own implementations for these interfaces.

Each interface and its instance methods are described in the following. In the instance method descriptions the word "this" refers to the instantiated object whose method is invoked.

The Xcerpt API can be divided into the following three modules, which are described in the following three sections.

Core Contains the interfaces an Xcerpt API end user has to deal with at most.

Node Representation Contains standard types for the representation of both results and input data.

Transformations Provides interfaces for the transformation of the standard type node representation into representation of various APIs.

3.3.3.1 Core

This module contains the most important interfaces of the Xcerpt API. It provides means for creating Xcerpt programs and queries and provides means for handling the results of program and query executions.

Interface XcerptFactory Represents a factory for the creation of Xcerpt programs, queries and diverse result transformers. The Xcerpt API end user mostly uses an instance of this interface for creating instances of all other interfaces defined in the Xcerpt API. An instance of an XcerptFactory can be obtained by the class method `createFactory(className:String)`.

The following methods are provided by `XcerptFactory`.

createFactory(className:String):XcerptFactory This class method (static method in Java) creates and returns an instance of XcerptFactory for a specific API implementation specified by its argument string. For example `createFactory("org.xcerpt.haskimpl.XcerptFactoryImpl")` returns an XcerptFactory instance for the API implementation that uses the Xcerpt prototype written in Haskell.

createProgram(program:Reader):XcerptProgram Creates and returns an `XcerptProgram` object for the Xcerpt program specified by the argument `program`. The argument *program* is a Reader providing a character stream of the Xcerpt program.

createProgram(program:Reader, vars:String[]):XcerptProgram Creates and returns an `XcerptProgram` object for the Xcerpt program specified by the argument *program*. The argument *program* is a Reader providing a character stream of the Xcerpt program. The argument *vars* specifies the variable names for which a substitution set is to be created when the returned `XcerptProgram` instance is executed.

createQuery(query:Reader, vars:String[]):XcerptQuery Creates and returns an `XcerptQuery` object for the Xcerpt query specified by the argument *query*. The argument *query* is a Reader providing a character stream of the Xcerpt query. The argument *vars* specifies the variable names for which a substitution set is to be created when the returned XcerptQuery instance is executed.

createQuery(query:String, vars:String[]):XcerptQuery Creates and returns an `XcerptQuery` object for the Xcerpt query specified by the string argument *query*. The argument *vars* specifies the variable names for which a substitution set is to be created when the returned XcerptQuery instance is executed.

createQuery(query:Reader, program:Reader, vars:String[]):XcerptQuery Creates and returns an `XcerptQuery` object for the Xcerpt query specified by the argument *query*. The query queries against the Xcerpt program specified by the argument *program*. The arguments *query* and *program* are Reader providing character streams for the Xcerpt query and program. The argument *vars* specifies the variable names for which a substitution set is to be created when the returned XcerptQuery instance is executed.

domTransformer():XcerptToDom Creates an `XcerptToDom` transformer instance that can be used to transform objects of the standard node representation (interface Node) into objects of the DOM API.

saxTransformer(ch:ContentHandler):XcerptToSAX Creates an `XcerptToSAX` transformer instance that can be used to transform objects of the standard node representation (interface Node) into appropriate events of the SAX API.

xcerptStreamTransformer(w:Writer):XcerptToXcerptStream Creates an `XcerptToXcerptStream` transformer instance that can be used to serialize objects of the standard node representation (interface Node) into an Xcerpt character stream.

xmlStreamTransformer(w:Writer):XcerptToXMLStream Creates an `XcerptToXMLStream` transformer instance that can be used to serialize objects of the standard node representation (interface Node) into an XML character stream.

Interface XcerptProgram Represents an Xcerpt program that contains at least one rule. A rule may either be a construct-query-rule (see Listing 3.28) or a goal (see Listing 3.29).

Listing 3.28: Construct-Query-Rule in Xcerpt

```
CONSTRUCT
  <Xcerpt construct term>
FROM
  <Xcerpt query>
END
```

Listing 3.29: Goal in Xcerpt

```
GOAL
  out {
    resource [ <output resource specification> ],
    <Xcerpt construct term>
  }
FROM
```

```
 <Xcerpt query>
END
```

and yields a result and substitution set.

This interface provides methods for achieving the results and substitution sets of this Xcerpt program (if the program contains goals). Results are created by the construct terms of each goal and can be obtained in a `ResultSequence` object returned by the `getResultSequence()` method. The substitution sets are created for each goal separately and contain substitutions for variables appearing in the query of the goal. Substitution sets are returned by the `getSubstitutionSet(goalNr:int)` method. The result sequence and substitution sets can not be obtained before the program is executed via the `execute()` method.

An `XcerptProgram` instance is created by an `XcerptFactory` instance for a specific API implementation.

The following methods are provided by the `XcerptProgram` interface.

execute():void Executes the Xcerpt program represented by this `XcerptProgram` instance in the underlying Xcerpt engine. This method does not terminate if the underlying Xcerpt engine does not terminate.

Invoking this method creates a result sequence and substitution sets for the goals appearing in the Xcerpt program. They can be obtained by various methods of the `XcerptProgram` interface.

getSubstitutionSet(goalNr:int):SubstitutionSet Returns the substitution set for the specified goal, containing substitutions for variables appearing in the query of the goal. The argument *goalNr* indicates for which goal of this `XcerptProgram` the substitution set is returned. If the specified goal does not yield a result, this method returns an empty substitution set.

This method must not be invoked before this XcerptProgram is executed via the `execute()` method.

getResultSequence():ResultSequence Returns the result sequence for this `XcerptProgram` containing one result element for each goal in this `XcerptProgram`, which yields a result. The returned result sequence does not contain "empty" elements for goals that do not yield a result.

This method must not be invoked before this XcerptProgram is executed via the `execute` method.

getSubstitutionSet():SubstitutionSet A convenience method returning a substitution set for the first goal in this `XcerptProgram`. Invoking `getSubstitutionSet()` has the same effect as invoking `getSubstitutionSet(1)`.

setInputData(data:Reader, id:String):void Sets the input data for some of the queries appearing in this program. Any query having an input resource specification of the form `"apiin:id"` will use this input data (where `id` corresponds to the argument *id* of this method).

The input data is specified by the argument *data*, which is a Reader providing a character stream of the data. The format of the data can be any format accepted by Xcerpt. The programmer must be careful to specify the correct format in the input resource specification (e.g `resource { "apiin:1", "xml" }` or `resource { "apiin:1", "xcerpt" }`).

If there is no apiin input resource specification with the given id in this program, then calling this method does not have any effect. Setting the input data for the same id multiple times will overwrite the data for that id.

setInputData(data:String, id:String):void Sets the input data for some of the queries appearing in this program. Any query having an input resource specification of the form `"apiin:id"` will use this data (where `id` corresponds to the argument *id* of this method).

The input data is specified by the string argument *data*. The format of the input data can be any format accepted by Xcerpt. The programmer must be careful to specify the correct format in the input resource specification (e.g `resource { "apiin:1", "xml" }` or `resource { "apiin:1", "xcerpt" }`).

If there is no apiin input resource specification with the given id in this program, then calling this method does not have any effect. Setting the input data for the same id multiple times will overwrite the data for that id.

setInputData(data:Node, id:String):void Sets the input data for some of the queries appearing in this program. Any query having an input resource specification of the form `"apiin:id"` will use this data (where `id` corresponds to the argument *id* of this method).

The input data is specified by the argument *data*. The programmer must be careful to specify the correct format in the input resource specification, i.e. "xcerpt" must be used as the input data format.

If there is no apiin input resource specification with the given id in this program, then calling this method does not have any effect. Setting the input data for the same id multiple times will overwrite the data for that id.

Interface XcerptQuery Represents an Xcerpt query term, that can be executed on the underlying Xcerpt engine. Since an Xcerpt query term itself is not an executable Xcerpt program, the Xcerpt API takes charge of bringing the query in an appropriate form that can be executed by the underlying Xcerpt engine.

A query can be executed via the `execute()` method, what yields in the creation of a substitution set for the variables appearing in the query, which can be obtained with the `getSubstitutionSet()` method.

The query either queries input data represented in its input resource specification (e.g. `in { resource { ... }, query }`), or it queries against a provided Xcerpt program (in which case the query must not have an input resource specification).

An `XcerptQuery` instance is created by an `XcerptFactory` instance for a specific API implementation. The following methods are provided by the `XcerptQuery` interface.

execute():void Executes this Xcerpt query and creates a substition set for the variables in the query, which can be obtained with the `getSubstitutionSet()` method.

This method does not terminate, if the underlying Xcerpt engine does not terminate.

setInputData(data:Reader, id:String):void Sets the input data for this query or one of its subqueries (queries inside boolean conjunctions or disjunctions). Any query having an input resource specification of the form `"apiin:id"` will use this input data (where `id` corresponds to the argument *id* of this method).

The input data is specified by the argument *data*, which is a Reader providing a character stream of the data. The format of the data can be any format accepted by Xcerpt. The programmer must be careful to specify the correct format in the input resource specification (e.g `resource { "apiin:1", "xml" }` or `resource { "apiin:1", "xcerpt" }`).

If there is no apiin input resource specification with the given id in this query, then calling this method does not have any effect. Setting the input data for the same id multiple times will overwrite the data for that id.

setInputData(data:String, id:String):void Sets the input data for this query or one of its subqueries (queries inside boolean conjunctions or disjunctions). Any query having an input resource specification of the form `"apiin:id"` will use this input data (where `id` corresponds to the argument *id* of this method).

The input data is specified by the string argument *data*. The format of the data can be any format accepted by Xcerpt. The programmer must be careful to specify the correct format in the input resource specification (e.g `resource { "apiin:1", "xml" }` or `resource { "apiin:1", "xcerpt" }`).

If there is no apiin input resource specification with the given id in this query, then calling this method does not have any effect. Setting the input data for the same id multiple times will overwrite the data for that id.

setInputData(data:Node, id:String):void Sets the input data for this query or one of its subqueries (queries inside boolean conjunctions or disjunctions). Any query having an input resource specification of the form `"apiin:id"` will use this input data (where `id` corresponds to the argument *id* of this method).

The input data is specified by the argument *data*. The programmer must be careful to specify the correct format in the input resource specification, i.e. "xcerpt" must be used as the input data format.

If there is no apiin input resource specification with the given id in this query, then calling this method does not have any effect. Setting the input data for the same id multiple times will overwrite the data for that id.

getSubstitutionSet():SubstitutionSet Returns the substitution set containing substitutions for variables in this query. This method must not be invoked before the `execute()` method is invoked.

Interface ResultSequence Represents the results of an Xcerpt program, which are created by the construct terms of the goals of the Xcerpt program. A `ResultSequence` instance is created, when an `XcerptProgram` instance is executed via the `execute()` method and is returned by the `getResultSequence()` method of the `XcerptProgram` interface.

The result sequence may be empty if the Xcerpt program does not contain any goals, or if its goals do not yield any results. An element in a result sequence represents the results of one goal.

A result sequence works with a so-called *cursor*, which is a pointer on the result elements in the sequence. Most of the methods in this interface provide means for moving this cursor forward or backward. If possible (with the underlying Xcerpt engine), an implementation of this interface must create the results in a lazy manner (or "on demand"), creating the results not before they are accessed by the end user. Initially the cursor is positioned *before* the first result element in the sequence, i.e. the cursor must be moved on a result element by the end user, before he can obtain them.

The following methods are provided by `ResultSequence`.

absolute(pos:int):boolean Moves the cursor to the absolute position in the sequence, specified by the argument *pos*. This method returns true, if there is a result element at the new position.

relative(pos:int):boolean Moves the cursor to a position relative to the current cursor position. Returns true, if there is a result element at the new position.

afterLast():void Moves the cursor to the position after the last item in the sequence. This method can be used to initiate a reverse iteration over the result elements.

beforeFirst():void Moves the cursor to the position before the first item in the sequence (initially the cursor is positioned on this position).

first():void Moves the cursor to the first position in the sequence, pointing at the first result element (if there is any).

getPosition():int Returns the current cursor position.

isAfterLast():boolean Checks whether the cursor is positioned after the last item of the sequence. Returns true, if it is, false otherwise.

isBeforeFirst():boolean Checks whether the cursor is positioned before the first item of the sequence (initially the cursor is positioned on this position). Returns true, if it is, false otherwise.

isFirst():boolean Checks whether the cursor is positioned on the first item of the sequence. Returns true, if it is, false otherwise.

isLast():boolean Checks whether the cursor is positioned on the last item of the sequence. Returns true, if it is, false otherwise.

isOnItem():boolean Checks whether the cursor is positioned on an result element. Returns true, if it is, false otherwise.

last():void Moves the cursor to the last item of the sequence.

next():boolean Moves the cursor to the next position in the sequence. Returns true, if there is an item at the new position, and false otherwise.

previous():boolean Moves the cursor to the previous position in the sequence. Returns true, if there is an item at the new position, and false otherwise.

count():int Returns the number of the items in this `ResultSequence` object.

getResult():Node Returns the result item the cursor is currently pointing at. Throws an exception, if the cursor is currently not pointing at a result item.

close():void Closes this `ResultSequence` object and releases all associated resources.

isClosed():boolean Checks whether this `ResultSequence` object is closed. Returns true, if it is, false otherwise.

Interface SubstitutionSet Represents a set of variable substitutions for a goal in an Xcerpt Program. Per goal there may be more than one substitutions for a variable, which are collected in a SubstitutionSet. Furthermore, an Xcerpt program may contain more than one goal, and hence there may be more than one `SubstitutionSet` instances created by the execution of an Xcerpt program. The substitution sets created by the execution of an Xcerpt program are obtained via the `getSubstitutionSet(goalNr:int)` method of the `XcerptProgram` interface.

The following methods are provided by SubstitutionSet.

print(w:Writer):void Serializes this substitution set to the specified character stream writer argument *w*.

getSet():Set⟨Substitution⟩ Returns an immutable set of the substitutions of this `SubstitutionSet` object. The returned set is a set representation provided by the host language. It can be used to iterate over the substitutions in this set.

Interface Substitution A `Substitution` stores bindings for variables.

The following methods are provided by `Substitution`.

get(varName:String):Node Returns the binding for the variable specified by the argument `varName`. Returns `null`, if there is no binding for the specified variable.

getMappings():Map⟨String, Node⟩ Returns an immutable map of the bindings in this substitution. The returned map is provided by the host language and can be used to iterate over the bindings in this map.

variableNamesIterator():Iterator⟨String⟩ Returns an iterator over the keys of the map of this Substitution.

isEmpty():boolean Returns true, if this substitution does not define any variable bindings, and false otherwise.

3.3.3.2 Node Representation

The Xcerpt API includes a hierarchy of classes for representing documents represented in Xcerpt syntax. This standard node representation is used throughout the Xcerpt API, both for representing input data and results. Figure 3.5 shows a class diagram containing each node type used for representing Xcerpt data. The various node types are described in more detail in the following.

Interface Node The `Node` interface is the basic data type of the Xcerpt data model, and is extended by various other types. A Node instance represents a single node in the document. All types extending the `Node` type inherit its methods for navigating through a document.

isAttributeNode():boolean Checks whether the `Node` object is an `AttributeNode`. Returns true, if it is, false otherwise.

isDocumentNode():boolean Checks whether the `Node` object is an `DocumentNode`. Returns true, if it is, false otherwise.

isElementNode():boolean Checks whether the `Node` object is an `ElementNode`. Returns true, if it is, false otherwise.

isContentNode():boolean Checks whether the `Node` object is a `ContentNode`. Returns true, if it is, false otherwise.

isCommentNode():boolean Checks whether the `Node` object is a `CommentNode`. Returns true, if it is, false otherwise.

isProcessingInstructionNode():boolean Checks whether the `Node` object is a `ProcessingInstructionNode`. Returns true, if it is, false otherwise.

getSuccessorCount():int Returns the number of successors of the `Node` object.

getSuccessors():List⟨Node⟩ Returns an immutable view of all successors of the `Node` object.

getOwnerDocument():DocumentNode Returns the owner document of the `Node` object. The owner document is the document this node was created with.

Interface NamedNode The `NamedNode` interface extends the `Node` interface, and provides additional methods. All node types that have a label, a prefix and a namespace uri (e.g. `ElementNodes` and `AttributeNodes`) extend this interface.

The following methods are provided by `NamedNode`.

getLabel():String Returns the label of this named node.

getNamespaceUri():String Returns the namespace URI of this named node. That is: If a namespace is directly set for this node, then that namespace is returned. If a prefix is defined for this node, then the namespace declaration for that prefix is retrieved in the owner document.

getNamespacePrefix():String Returns the namespace prefix of this `NamedNode` object, if it has any.

setLabel(String label):void Sets the label of this `NamedNode`.

setNamespaceUri(uri:String):void Sets the namespace URI of this `NamedNode`. This method removes any previously set prefix and namespace.

setNamespacePrefix(prefix:String):void Sets the namespace prefix of this named node. This method removes any namespace previously set for this node. Throws an `IllegalArgumentException`, if there is no namespace declaration for the specified prefix in the owner document.

equalNames(other:NamedNode):boolean A named node object equals another named node object, if they both have the same namespace and label. The namespace prefix is not important for equivalence.

Interface DocumentNode The `DocumentNode` interface represents an Xcerpt document object and extends the `Node` interface. `DocumentNode` provides methods for creating `ElementNodes`, `AttributeNodes` and `ContentNodes` (text content). The created nodes belong automatically to this document node. Since Xcerpt documents do not support nested namespace declarations (as in XML), the `DocumentNode` interface provides a method for globally setting the namespace declaration for the whole document.

The following methods are provided by `DocumentNode`.

getNamespaceDeclaration(prefix:String):String Returns the namespace declaration for the specified prefix. All nodes in this document that have the same prefix p also have the same namespace as determined by this method (Xcerpt does not support nested namespace declarations as in XML). This method returns `null` if no namespace is declared for the specified prefix.

setNamespaceDeclaration(prefix:String, namespace:String):void Sets the namespace declaration for the specified prefix and namespace. All nodes in this document that have the same prefix as the specified argument will have the same namespace as specified by this method (Xcerpt does not support nested namespace declarations as in XML). If the document has already a mapping for the specified prefix the old value is replaced by the specified namespace. Null arguments are not allowed.

getDeclaredNamespacePrefixes():Set⟨String⟩ Returns the namespace prefixes declared for this document.

createElementNode(label:String):ElementNode Creates and returns a new element node. The owner document of that element node will be set to this document.

createAttributeNode(label:String, value:String):AttributeNode Creates and returns a new attribute node. The owner document of that attribute node will be set to this document.

createContentNode(content:String):ContentNode Creates and returns a new content node. The owner document of that content node will be set to this document.

getRootElement():ElementNode Returns the root element node of this document instance. If this document does not have any root element set, returns null.

getOwnerDocument():DocumentNode This method throws an `IllegalStateException` because there is no owner document defined for `DocumentNode`s.

getSuccessorCount():int Since a document has only one successor, called root element, this method returns always 1.

getSuccessors():List⟨Node⟩

getBaseURI():URI

Interface ElementNode The `ElementNode` interface represents data terms in an Xcerpt document (corresponding to element nodes in XML), and extends the `Node` interface. Element nodes may have attributes associated with them, and provides methods for retrieving attributes or adding or removing them.

The following methods are provided by ElementNode.

getAttributes():List⟨AttributeNode⟩ Returns a list of attributes that belong to this `ElementNode`.

setOrdered(ordered:boolean):void Sets whether this element node is ordered or unordered, as specified by the argument *ordered*.

getAttributeCount():int Returns the number of attributes in this `ElementNode`.

addAttribute(attribute:AttributeNode):void Adds the specified attribute to this `ElementNode`. The owner element of the specified attribute will be set to this element node. If the specified attribute already has an owner element, an `IllegalStateException` is thrown. If an attribute with equal label and namespace already exists in this element node's attributes list, then the existing attribute is overwritten. This method throws an `IllegalStateException`, if the argument attribute does have an owner document different to this element node's owner document.

addAttributes(attributes:List⟨AttributeNode⟩):void Adds the specified attributes to this `ElementNode`. The owner element of the specified attributes will be set to this element node. If the specified attributes already have an owner element, an `IllegalStateException` is thrown. If any attribute of the specified attributes list with equal label and namespace already exists in this element node's attributes list, then the existing attribute is overwritten. This method throws an `IllegalStateException`, if the argument attributes does have any attribute that's owner document is different to this element node's owner document.

removeAttribute(label:String, namespaceUri:String):void Removes the attribute with specified label and namespace from this element node's attributes list. If the namespace argument is null, an attribute is removed that has the specified label and no specified namespace. If no such attribute exists, nothing happens.

addSuccessor(successor:Node):void Adds a new successor node to this element node's list of successors. The successor node to be added must not be an `AttributeNode`. To add attributes use addAttribute(att:AttributeNode) or addAttributes(atts:List⟨AttributeNode⟩).

This method throws an `IllegalStateException`, if the argument node does have an owner document different to this element node's owner document.

removeSuccessor(successor:Node):void Removes the specified successor node from this element node's list of successors. The successor node to be removed must not be an `AttributeNode`. To remove attributes use removeAttribute(label:String, namespaceUri:String). If no such successor exists in the successor list of this `ElementNode`, nothing happens. This method throws an `IllegalStateException`, if the argument node does have an owner document different to this element node's owner document.

addSuccessors(successors:List⟨Node⟩):void Adds new successor nodes to this element node's list of successors. This method throws an `IllegalStateException`, if the argument list does have any node that's owner document is different to this element node's owner document.

getAttribute(label:String, namespace:URI):AttributeNode Returns an attribute of this `ElementNode` with the specified label and namespace. If the namespace argument is null, an attribute is returned that has the specified label and no specified namespace URI. If no such attribute is found, null is returned.

isOrdered():boolean Checks whether this ElementNode is ordered.

Interface AttributeNode The `AttributeNode` interface represents an attribute of an element node, and extends the Node interface. Since attributes have no children, some of the inherited methods like `getSuccessors()` have no effect and return "empty" values. Each attribute of a specific element is unique by its *label* or *namespace* and *label*.

The following methods are provided by `AttributeNode`.

getValue():String Returns the value property of this `AttributeNode` object.

setValue(value:String):void Sets the value property of this `AttributeNode` object.

getOwnerElement():ElementNode Returns the `ElementNode` which this attribute belongs to.

Interface ContentNode The `ContentNode` interface represents text content in an Xcerpt document (corresponding to text nodes in XML), and extends the `Node` interface. Since text nodes have no children, some inherited methods have no effect and return "empty" values (e.g. getSuccessors() returns an empty list).

The following methods are provided by `ContentNode`.

setContent(content:String):void Sets the content of this `ContentNode` instance, as specified by the string argument `content`.

getContent():String Returns the content of this `ContentNode` instance.

3.3.3.3 Transformations

Besides the standard data representation with `Node`, the Xcerpt API also supports flexible representations of data using other data formats and APIs. For this purpose various transformer interfaces are defined, which provide means for transforming data represented in the standard type `Node` to data represented in other formats or APIs. For example, transformers are provided that transform data represented as `Node` to data represented as DOM nodes or to SAX events. Transformer instances can be obtained using an

`XcerptFactory` instance for a specific Xcerpt API implementation. To simplify the implementation of the Transformers for a concrete Xcerpt engine, the Xcerpt API specifies a *NodeVisitor* interface that is based on the *Visitor* design pattern.

The following interfaces are defined for the transformation of data represented as Node.

Interface NodeVisitor This interface realizes the *Visitor* design pattern. The Visitor design pattern is used as an approach for realizing an algorithm for various polymorph types. A visitor encapsulates an algorithm for the various types of an object structure, for example the various node transformers are defined to be node visitors.

The objects in the node object structure invoke call-back methods of the `NodeVisitor` interface (*visit()* methods). The `NodeVisitor` interface provides such call-back methods for each type in the node type hierarchy, and hence polymorphism is assured. Each type in the node type hierarchy defines an *accept()* method accepting a node visitor, and invoking the appropriate call-back method in the visitor.

Visitors should be used if the object structure changes rarely but new algorithms are introduced often. The `Node` class hierarchy fulfills this criteria, so it is safe using a node visitor whenever new algorithms are introduced.

Interface XcerptToDOM Transforms data represented as Node to data represented as DOM nodes.

transform(doc:DocumentNode):org.w3c.dom.Document This method transforms an org.xcerpt.DocumentNode instance into an org.w3c.dom.Document instance.

transform(doc:Node):org.w3c.dom.Node This method transforms an org.xcerpt .Node instance into an org.w3c.dom.Node instance.

Interface XcerptToXcerptStream Transforms data represented as `Node` to an Xcerpt character stream representation.

transform(doc:org.xcerpt.DocumentNode):Void This method transforms an org.xcerpt.DocumentNode instance into an Xcerpt stream.

transform(node:org.xcerpt.Node):Void This method transforms an org.xcerpt .Node instance into an Xcerpt stream.

setWriter(w:Writer):void Sets the specified writer for this XcerptToXcerptStream transformer.

Interface XcerptToXMLStream Transforms data represented as Node to an XML character stream representation.

transform(doc:DocumentNode):Void This method transforms an org.xcerpt .DocumentNode instance into an XML stream.

transform(node:Node):Void This method transforms an org.xcerpt.Node instance into an XML stream.

setWriter(w:Writer):void Sets the specified writer for this XcerptToXMLStream transformer.

Interface XcerptToSAX Transforms data represented as Node to SAX events.

transform(DocumentNode doc, ContentHandler saxHandler):void This method transforms an org.xcerpt.DocumentNode into a sequence of calls to methods of the specified SAX ContentHandler. If any SAXExceptions are thrown by the specified content handler's methods, they will be wrapped in RuntimeExceptions that are thrown instead.

transform(Node doc, ContentHandler saxHandler):void This method transforms an org.xcerpt.Node into a sequence of calls to methods of the specified SAX ContentHandler. If any SAXExceptions are thrown by the specified content handler's methods, they will be wrapped in RuntimeExceptions that are thrown instead.

3.3.4 Usage

This section describes the usage of the Xcerpt API, demonstrating it with short code examples. Every program using the Xcerpt API starts with the creation of an XcerptFactory instance, as is shown in Section 3.3.4.1. Section 3.3.4.2 describes how the factory instance is used for creating instances of the XcerptProgram and XcerptQuery interfaces. After that, Section 3.3.4.3 explains the possibilities of specifying input resources with the Xcerpt API. Finally, the execution of an Xcerpt program or query and the handling of results using the ResultSequence and SubstitutionSet interfaces is described in Section 3.3.4.4.

3.3.4.1 Creation of XcerptFactory Instance

Every program using the Xcerpt API starts by creating an XcerptFactory instance. An Xcerpt factory instance is created for a specific implementation of the Xcerpt API and takes care about the creation of instances for the most important interfaces of the API. The following listing demonstrates how an XcerptFactory instance can be obtained for a specific API implementation (in this case the API implementation for the Xcerpt prototype engine is used).

Listing 3.30: Obtaining an XcerptFactory instance for a specific API implementation

```
XcerptFactory factory = XcerptFactory.createFactory(
            "org.xcerpt.haskimpl.XcerptFactoryImpl");
XcerptProgram program = factory.createProgram(...);
XcerptQuery query = factory.createQuery(...);
```

The listing also shows how the factory is used for creating instances of various interfaces defined in the Xcerpt API. Once a specific API implementation is chosen and an XcerptFactory instance is obtained, the factory takes care of creating instances for the right API implementation. The creation of an factory instance is the only time the end user has to deal with code that is specific for an API implementation (i.e. the string argument given to the `createFactory()` method is specific to the API implementation). Hence this is also the only place that must be changed in order to switch between various API implementations. All other parts of the code remain the same, when the end user decides to switch to another API implementation.

3.3.4.2 Creating XcerptProgram and XcerptQuery instances

After an XcerptFactory instance is created for a specific API implementation, the programmer can start creating XcerptProgram and XcerptQuery instances. The XcerptFactory provides various factory methods for creating those instances, corresponding to various usage modi. Each of the usage modi has different purposes, as is demonstrated in the following.

Usage Mode: XcerptProgram In this usage mode an ordinary Xcerpt program is executed by the underlying Xcerpt engine. This is done using the XcerptProgram interface, which provides means for executing the program and obtaining the results of the execution. The following listing demonstrates how an XcerptProgram instance can be created and executed and how the results of the execution can be obtained.

Listing 3.31: Creation of an XcerptProgram

```
String programString = xcerptProgramString();
XcerptProgram xcerptProgram = factory.createProgram(programString);
xcerptProgram.execute();
ResultSequence constructedResults = xcerptProgram.getResultSequence();
```

Usage Mode: XcerptProgram and Variables In this usage mode an ordinary Xcerpt program is executed by the underlying Xcerpt engine, and substitution sets are obtained, which are found by the engine for the goals inside the program. Similar to the previous usage mode, in this mode the XcerptProgram interface is used. In contrast to the first mode however, this usage mode yields substitution sets. Whereas in the first mode a a ResultSequence instance is obtained, which can be used to iterate over the results constructed by every goal in the program, now a SubstitutionSet instance is obtained for a particular goal. The API end user can provide a list of variable names, for which he wants to obtain the substitution sets. This is demonstrated by the following listing.

Listing 3.32: Creation of an XcerptProgram and variables

```
String programString = xcerptProgramString();
String[] varnames = new String[] {"X","Y","Z"};

XcerptProgram xcerptProgram = factory.createProgram(programString, vars);
xcerptProgram.execute();

int goalNr = 1;
SubstitutionSet substSet = xcerptProgram.getSubstitutionSet(goalNr);
```

Usage Mode: XcerptQuery and Variables In contrast to the previous two usage modi, the usage mode described next does not execute an Xcerpt program, but executes a standalone Xcerpt query. For this purpose the API provides the XcerptQuery interface, for which an instance can be created by using the XcerptFactory. In this mode, an XcerptQuery instance is created for any ordinary Xcerpt query, which contains one or more input resource specifications. The query is meant to query the input resources specified inside the query. Because a standalone query does not yield constructed data, the execution of an XcerptQuery instance yields substitution sets only (no result sequence can be obtained), and the API end user can provide a list of variable names, for which he wants to obtain the substitutions resulting from the execution of the query. The following listing demonstrates this usage mode.

Listing 3.33: Creation of an XcerptQuery with variables

```
String queryString = xcerptQueryString();
String[] varnames = new String[] {"X","Y","Z"};

XcerptQuery xcerptQuery = factory.createQuery(queryString, varnames);
```

```
xcerptQuery.execute();

SubstitutionSet substSet = xcerptQuery.getSubstitutionSet();
```

Usage Mode: XcerptQuery, XcerptProgram and Variables This usage mode is a combination of the previous two usage modi. Again, an Xcerpt query is executed in this mode. This time however, the query is not meant to query input data specified with input resource specifications inside the query. Rather the query queries data that is constructed as a result of executing an Xcerpt program. Hence, in this usage mode, the API end user provides both, an Xcerpt program and an Xcerpt query. The XcerptFactory takes both and creates an XcerptQuery instance, that can be executed as before. The following listing demonstrates this usage mode.

Listing 3.34: Creation of an XcerptQuery with an XcerptProgram and variables

```
String queryString = xcerptQueryString();
String programString = xcerptProgramString();
String[] varnames = new String[] {"X","Y","Z"};

XcerptQuery xcerptQuery = factory.createQuery(
            queryString, programString, varnames);
xcerptQuery.execute();

SubstitutionSet substSet = xcerptQuery.getSubstitutionSet();
```

3.3.4.3 Defining Input Resource Specification

The Xcerpt API supports two ways of specifying input resources in an Xcerpt program (or query). The first one is the regular input resource specification of Xcerpt as is shown in Listing 3.35.

Listing 3.35: Regular input resource definition

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

Input resources that are specified this way are first obtained and copied by the API. After that, the Xcerpt program is executed in the underlying engine using the copy of the input resource. The regular input resource specifications in Xcerpt support any *URI*, but some API implementations may not support every protocol due to engine limitations. In either case the API implementation takes care that the "http" protocol (for input resources in the WEB) and the "file" protocol (for local files) can be used, regardless of whether the underlying engine supports them or not. Support for further protocols is implementation dependant.

The second way of specifying an input resource is provided by the API and is not available in regular Xcerpt programs. Input resource specifications may contain an URI with the *apiin* "protocol". The advantage of introducing this protocol is that more than one input resources can be specified in an Xcerpt program in a flexible manner as it is provided by the Xcerpt engine so far. A further convenience of this approach is that the user can specify input resources even if he has no connection to the internet (is needed if the http protocol is used) or has no rights for reading from files (is needed if the file protocol is used) on

the operating system where the API application runs. The following listing shows the usage of the *apiin* "protocol".

```
in {
      resource [ "apiin:someIdentifier", <format> ],
      <Xcerpt query>
}
```

Input resources specified this way allow the programmer to set input data in a more API wise manner, by using `setInputData()` methods provided by the API. The following listing shows a short code fragment using such an input resource specification.

Listing 3.36: API specific input resource specification

```
String query = "in {"
          + " resource [ 'apiin:id', <format> ],"
                    + " desc b [ var X ]"
          + "}";
String data = "a { b {1}, b {2} }";

XcerptQuery xq = factory.createQuery(query, ...);
xq.setInputData(data, "id");
xq.execute();
...
```

In this code fragment, two strings are created representing a query and the data to be queried. Note that the query contains an *apiin* input resource specification. After obtaining the XcerptQuery instance, the input resource data is set by invoking its `setInputData(data, "id")` method. This links the *apiin:id* input resource specification in the query with the given data.

3.3.4.4 Result Handling

This section describes how the API end user can handle the results of the execution of an Xcerpt program or query. After the execution of an Xcerpt program the API end user can obtain constructed results and substitution sets, which are represented by the interfaces *ResultSequence* and *SubstitutionSet*. In the following first the ResultSequence interface is explained, and after that the SubstitutionSet interface is explained.

A ResultSequence works with a cursor, enabling iterative access to the results. If the underlying Xcerpt engine supports it, the results can thus be retrieved in a very efficient way "on demand". The ResultSequence interface provides many methods for moving the cursor in the sequence of results and for retrieving the actual result. For example, the following listing shows how one can iterate through *all* results in the result sequence.

Listing 3.37: Usage of ResultSequence

```
xcerptProgram.execute();

ResultSequence results = xcerptProgram.getResultSequence();

while (results.next()) {
   Node result = resSequence.getResult();
```

```
   processResult(result);
}
```

The SubstitutionSet interface represents substitution sets retrieved as result of query (or program) execution. This interface provides methods for accessing particular substitutions in the substitution set. The following listing demonstrates the usage of the SubstitutionSet interface.

Listing 3.38: Usage of SubstitutionSet

```
xcerptQuery.execute();

SubstitutionSet subs = xcerptQuery.getSubstitutionSet();

for (Substitution sub : subs) {
   Iterator<String> it = sub.variableNamesIterator();

   while (it.hasNext()) {
      String varName = it.next();
      Node binding = sub.getBinding(varName);

      process(varName, binding);
   }
}
```

3.4 API Implementation for the Xcerpt Prototype Engine

In this chapter an implementation of the Xcerpt API for a concrete Xcerpt engine (Haskell prototype engine) is provided. This section describes the API implementation and how the API architecture (cf. Section 3.3.2) is realized in it. The API implementation supports the *Xcerpt prototype engine* [66], and is written in the *Java programming language* [1].

The Xcerpt API is not limited to the Java programming language, as it can be implemented in any other object oriented programming language. Although only one implementation of the API in the Java programming language is given in this chapter, the translation of this implementation to other object oriented programming languages should be straight forward. Using Java in this chapter is not an arbitrary decision, as Java is a modern, platform independent object oriented programming language providing a huge range of various APIs. We believe that the Xcerpt API end user benefits the most from an API implementation in Java.

The Xcerpt prototype engine [66], the Xcerpt API is implemented for, is a proof-of-concept implementation of Xcerpt. A better engine implementation using an abstract machine [31] is currently being developed but was not available at the time this chapter was written. The prototype engine has some shortcomings, which are to be dealt with by the API implementation, as is described in the following sections.

The API implementation follows closely the abstract architecture (cf. Section 3.3.2), which defines three execution phases, namely *preprocessing phase*, *execution phase* and *postprocessing phase*. Figure 3.6 shows how this architecture is realized in the concrete implementation. The preprocessing phase of the abstract architecture is mainly realized by the so-called `ProgramParser`, which preprocesses the Xcerpt input programs provided by the end user. The `ProgramParser` and the preprocessing of Xcerpt programs is explained in Section 3.4.1. It is followed by Section 3.4.2, which describes the execution phase. Finally,

the postprocessing phase is realized by the `ResultParser`, which processes the results provided by the prototype engine, as is discussed in Section 3.4.3. The focus of Section 3.4.4 is, how the representation and the handling of the results of an Xcerpt program execution are implemented in the implementation of the Xcerpt API.

3.4.1 Preprocessing Phase: Program Parser

The preprocessing phase of the Xcerpt API architecture is realized mainly by the `ProgramParser` in the concrete API implementation for the prototype engine. The `ProgramParser` is a Java class with the function of rewriting Xcerpt programs, which are provided by the API end user, without changing their semantics. The rewriting of the Xcerpt programs is needed for several reasons. First, input resources specified in the Xcerpt program have to be copied by the API (as it is demanded by the abstract architecture specification). Also input resource specifications containing *apiin* "protocols" have to be replaced by ordinary resource specifications, because *apiin* is no standard protocol supported by the Xcerpt prototype engine. Another reason for the rewriting of Xcerpt programs is the necessity of receiving substitution sets obtained for goals in the Xcerpt program. Finally, the Xcerpt API allows the execution of single Xcerpt queries, not just Xcerpt programs. In order to execute such queries in the Xcerpt prototype engine, the queries must be rewritten into ordinary Xcerpt programs. All of this is explained in further detail in the following.

3.4.1.1 Replacing Input Resource Specifications

As it is demanded by the Xcerpt API architecture specification, input resources have to be copied before an Xcerpt program is executed. The ProgramParser does this for several supported input resource specifications.

The copying of input resources has the advantage that specific protocols can be used by the API end user, which are normally not supported by the underlying Xcerpt prototype engine. For example, the prototype engine does not support the HTTP protocol, though the API end user can use the HTTP protocol for input resource specifications. The `ProgramParser` copies all input resources and saves them to temporary local files. The `ProgramParser` further replaces all input resource specifications in the Xcerpt program, such that the new resource specifications point to the temporary local files.

Also input resource specifications containing the non-standard *apiin* protocol are replaced this way. The `ProgramParser` replaces all *apiin* input resource specifications in the Xcerpt program, such that the new resource specifications point to a temporary local file, which contains the input resource data provided by the API end user by invoking the `setInputData()` method of `XcerptProgram` or `XcerptQuery` objects.

3.4.1.2 Wrapping of Construct Terms

As specified by the abstract API architecture, the execution of an Xcerpt program shall yield two different kinds of results. The first kind of result is the data constructed by the construct terms inside goals of the Xcerpt program. The second kind of result is the substitution sets that are found for variables inside the query of each goal of the Xcerpt program. The Xcerpt prototype engine only returns the data constructed by construct terms inside goals, but does not return the substitution sets that are found for those goals. The API extends the Haskell prototype engine to allow substitution sets, by wrapping construct terms around the original construct terms inside goals of the Xcerpt program. The following two listings demonstrate the rewriting of construct terms inside Xcerpt goals, Listing 3.39 shows the original Xcerpt goal and Listing 3.40 shows the rewritten Xcerpt goal.

Listing 3.39: Original Xcerpt Rule

```
GOAL
      out {
      resource { "stdout:", "xml" },
      <CONSTRUCT TERM>
      }
FROM
      <QUERY TERM>
END
```

Listing 3.40: Rewritten Xcerpt Rule

```
GOAL
      out {
      resource { "stdout:", "xcerpt" },
            resultsAndSubs [
                  result [
                        <CONSTRUCT TERM>
                  ],
            subs [
                        all sub [
                  map [ "X1", var X1 ],
                              ....
                  map [ "Xn", var Xn ]
                        ]
                  ]
            ]
      }
FROM
      <QUERY TERM>
END
```

As one can see, the rewriting inserts terms of the form `map [ ''X'', var X ]` into the construct term of the goal. These insertions are done for every variable *X* appearing inside the query term of the goal. The terms are wrapped inside *sub* terms, which are preceded by the grouping construct *all*. This way a *sub* term is constructed for each possible substitution. The original construct term is put as a subterm inside the *result* term. This rewriting is later undone in the postprocessing phase, which extracts the data constructed by the original construct term and the substitution set constructed by the new construct terms.

3.4.1.3 Rewriting Xcerpt Queries into Xcerpt Programs

The Xcerpt API specification defines the `XcerptQuery` interface, which allows the end users to execute simple Xcerpt queries against the same set of rules without creating a specific program for each of the queries. The execution of a standalone query does not yield the construction of results but yields substitution sets only. The Xcerpt prototype engine however does not support the execution of standalone queries. This limitation is worked around in the API implementation by automatically rewriting the Xcerpt queries into Xcerpt programs.

Internally, a simple Xcerpt goal is wrapped around the query and instances of the `XcerptQuery` interface create instances of the XcerptProgram interface. This way all functionalities can be delegated to

the internal `XcerptProgram` instance. The execution of an `XcerptQuery` instance is realized by executing the internal `XcerptProgram` instance, which then yields both, the construction of result data (which is a dummy result in this case) and the substitution set for the single goal in the program. The `XcerptQuery` instance only returns the substitution set and ignores the constructed dummy result.

3.4.1.4 Implementation of the ProgramParser

The `ProgramParser` class is basically an LL(1) parser, parsing the Xcerpt programs provided by the API end user. The purpose of the parser is not to build an abstract syntax tree for the Xcerpt program, but to rewrite specific parts inside the program only.

For this reason the parser also does not need to accept the Xcerpt language, it is sufficient if the parser accepts a more general superset of this language. The `ProgramParser` mainly checks the "well-formedness" of the Xcerpt program (regarding correct bracketing) and determines parts in the program that have to be rewritten (such as input resource specifications and the wrapping of construct terms inside goals).

Parsing only a superset of the Xcerpt language does not allow the end user to provide wrong Xcerpt programs (programs with syntax errors), because the underlying Xcerpt engine still performs its syntax checks on the rewritten Xcerpt program. If the end user writes programs with syntax errors, these errors remain in the rewritten program and the underlying engine reports a failure, which is then forwarded to the end user through the API.

As already mentioned, the ProgramParser is an LL(1) parser. It is generated using a parser generator for the Java language. The parser generator JavaCC [27] is chosen for this purpose. It is capable of generating efficient LL(k) parsers for any lookahead k (k is 1 in this case), and uses the *predictive recursive descent* technique for implementing the parser to be generated. Together with the facts that only a simple bracketed language is parsed (superset of Xcerpt) and no abstract syntax tree is built, this yields a very efficient way of rewriting Xcerpt programs.

3.4.2 Execution Phase: Executing the Underlying Engine

This section describes how the execution phase is solved in the Xcerpt API implementation and gives demonstrating code examples. The execution phase is intended to execute the Xcerpt program in the underlying engine. The Xcerpt prototype engine provides an executable binary file for the execution, which is not a Java binary file. Hence a "foreign" process must be executed in the API implementation, what can be done in several ways.

Execution of "foreign" processes in Java can be done with the Java Native Interface (JNI). JNI is used when platform-specific features are not supported by the Java API or when an application is to be executed that is written in a foreign programming language. JNI is a very powerful Java technology but has a very severe disadvantage: using JNI in an API implementation explicitely forces the API end user to deal with platform dependant JNI libraries. The inclusion of JNI libraries is a non-trivial task for most end users, and hence using explicite JNI in the API implementation is contradictive to the simplicity requirement demanded by the abstract API specification (cf. Section 3.3.1).

A more simple solution is to use the Java class *Process* instead. This class implicitely deals with the platform dependencies (it uses JNI internally). The advantage of *Process* is that the end user does not have to deal with the JNI libraries explicitely. The necessary JNI libraries are automatically provided by every Java Virtual Machine installation and thereby preserving the platform independence at the end user level.

The following listing shows a fragment of the `execute()` method of the `XcerptProgram` interface (the `execute()` method of interface `XcerptQuery` delegates to this too).

Listing 3.41: Execution of the prototype engine via the Process class

```
File program = getProgramFile();
Process p = Runtime.getRuntime().exec(                                    2
      new String[] { "xcerpt",
            "-f", "-o", "xcerpt",                                         4
            "-p", program.getPath() });
InputStream in = p.getInputStream();                                      6
```

An instance of the Process class is obtained for the executable binary `xcerpt` (the binary must be availabe in the system) and some command line arguments. Note that the output format for constructed results is set to "xcerpt" by the command line argument `-f -o xcerpt`. This overrides output formats of output resource specifications inside the Xcerpt program. The API implementation expects the output formats to always be "xcerpt", as the `ResultParser` parses this format only. If the API end user wants the results to be in another format, he must use one of the provided transformators. After executing a process and obtaining the Process object, an input stream can be obtained by calling the `getInputStream()` method. This input stream delivers all data that the process writes in its standard output stream. Hence, the API implementation receives the results of the Xcerpt program execution via this input stream.

3.4.3 Postprocessing Phase: Result Parser

The *postprocessing phase* is realized by the `ResultParser` class in the concrete API implementation. The `ResultParser` is intended to process the results of the execution of the underlying Xcerpt engine. This is accomplished by "undoing" the rewriting of the `ProgramParser` in the preprocessing phase, that is it separates the constructed results from the substitution sets. Its main purpose however is to parse the data terms and to build an object structure that uses the Node class hierarchy (cf. Section 3.3.3.2).

3.4.3.1 Unwrapping the Results and Substitution Sets

As described in Section 3.4.1, the user provided Xcerpt programs get rewritten in the preprocessing phase. The wrapping that is done around the construct terms of each goal has to be "undone" in order to preserve the original results and separates them from the substitution sets. The `ResultParser` class was written for this purpose. It separates results and substitution sets of each goal, such that the Xcerpt API end user can receive them in an API wise manner by using appropriate getter methods of the interfaces `XcerptProgram` and `XcerptQuery`.

3.4.3.2 Building the Node Object Structure

The main purpose of the ResultParser is to build an object structure for constructed results and substitution sets. The object structure follows the Node class hierarchy of the Xcerpt API, which simulates Xcerpt data term graphs. The main difficulty of building the object structure is that the graphs must be built by dereferencing all references appearing inside the textual results constructed by the Xcerpt engine.

Several object graph structures are built during the parsing of the textual output obtained from the underlying Xcerpt engine. One graph is built for every constructed result for each goal inside the executed Xcerpt program. Furthermore one graph is built for every binding of a variable appearing inside the substitution sets obtained as results for each goal.

3.4.3.3 Implementation of the ResultParser

The implementation of the `ResultParser` is done similar to the implementation of the `ProgramParser`: the parser generator JavaCC is used for generating an LL(1) parser. However the result parser has much more work to do than the program parser has. Whereas the program parser's only purpose was the rewriting of a "simple" bracketed language, the result parser must parse full Xcerpt data terms and built an appropriate object structure for them.

The result parser parses the data term results for every goal and forwards the built object graph and substitution sets to the corresponding `XcerptProgam` or `XcerptQuery` instances, which provide methods for the API end user for retrieving them. It is worth considering not to parse the results of a goal, if the API end user does not request them. The parsing of such results could be done lazily at the time the results are requested by the end user. This would contribute to the efficiency of the API implementation, as there may be situations in which the end user is not interested in the results of every goal.

3.4.4 Result Representation and Handling

This section describes the implementation of the classes which deal with the representation and handling of Xcerpt results. Section 3.4.4.1 explains the internal representation of `ResultSequences` that store all possible results of an Xcerpt program. The next section describes the implementation of the `SubstitutionSet` and `Substitution` interfaces, which represent the variable bindings resulting from the execution of an Xcerpt program. Section 3.4.4.3 explains how the result transformers are implemented.

3.4.4.1 ResultSequence

Each `XcerptProgram` instance has a property in which the constructed results for that program are stored after its execution. The type of this property is `ResultSequence` and it is accessible via a getter method. The `ResultSequence` is implemented by using the Java standard type *List* internally, more precisely a parameterized *List⟨Node⟩* is used.

The size of the internal list is at most equal to the number of goals appearing inside the Xcerpt program, because some of the goals may not construct any results.

As described in 3.3.3.1 each result in the `ResultSequence` can be accessed by a cursor. This cursor is implemented in a straight forward manner using an integer pointer to a position in the internal List. Although the abstract Xcerpt API specification requires a more efficient iterative implementation, this is not done in this concrete API implementation because the underlying engine does not support accessing results in an iterative manner.

3.4.4.2 Substitution Set

A `SubstitutionSet` represents all possible substitutions of one goal in an Xcerpt program after its execution. It is implemented by using a parameterized standard Java *Set⟨Substitution⟩* type.

The elements in a `SubstitutionSet` are `Substitution` objects. A `Substitution` represents a variable binding and is implemented by the parameterized Java standard type *HashMap⟨String, Node⟩*. The *key* represents the name of the variable and the *value* represents the *Node* object the variable is binded to.

Again, a more efficient implementation would be desirable, which constructs the substitutions on demand, but this is not realized here because of the missing feature of the Xcerpt engine to support iterative result construction.

3.4.4.3 Transformer

The Xcerpt API provides various result representations that are achieved by various *Transformers* (`XcerptToXMLStream`, `XcerptToXcerptStream`, `XcerptToSAX`, `XcerptToDOM`). For the implementation of the Transformers a design pattern called the *Visitor design pattern* as already described in Section 3.3.3.1 is used. Each Transformer, of the provided four Transformers, implements the *NodeVisitor* interface and defines the required methods.

3.5 Use Case: Web Application

As proof-of-concept of the Xcerpt API specification and implementation, this section describes the web application use case. [7] In fact, the web application is both, a use case and a demonstration tool for applying the Xcerpt use cases. The web application provides only a fragment of the functionality of the Xcerpt API and serves mostly demonstration purposes.

Section 3.5.1 explains the implementation of the web application. Section 3.5.2 the usage describes the web application as a web service and as a web interface. At last Section 3.5.3 demonstrates the four usage modi, provided by the Xcerpt API with Xcerpt use cases.

3.5.1 Implementation

The Web application runs on the *Apache Tomcat Server 6.0* [3]. The Web application is implemented by using the Xcerpt API and is based on JSP (Java Server Pages) [73, 40, 5]. JSP is an alternative to implement a web application by servlets. In fact JSP is internally transformed into a Java Servlet, so it can be seen as an extension of the Servlet API. The main difference between the two approaches is that Servlets use an embedded HTML code within Java code, and JSP uses embedded Java code within HTML code. The JSP approach is chosen for our purpose. *Java Beans* [71] is used in context with JSP. *Java Beans* can be seen as an instance of a class with properties. The access to these properties is achieved by conventional methods, as *getter* and *setter* methods. In the following the class *QueryBean* is presented that fulfills the *Java Beans* criteria and also shows how the Xcerpt API can be used in a program. Further, the technologies from [35, 44, 11] are used for the implementation of the web application.

QueryBean This part demonstrates the two most interesting methods of the class *QueryBean*, namely the `getSubstitutionSet()` and the `getConstructedData()`. All other method implementations of the class *QueryBean* are straight forward, they are "usual" getter and setter methods, and are not further described here. The two methods are used by the web application to get the substitutions and the constructed data for representing them in the browser(web interface) or to forward them to another software program(web service).

Listing 3.42 demonstrates these two methods using the Xcerpt API. Lines 10, 13 and 25 demonstrate the creation methods of the XcerptFactory class for the creation of Xcerpt queries and Xcerpt program. The `setInputData` method, invoked in lines 17 and 28, specifies the input data for the Xcerpt program (or query). This method is invoked only if input data is provided by the user of the web application. Due to simplicity reasons, only one input data can be provided by the user of the web application, hence the identifier for the input data is always the same (`id=1`).

As can be seen from the code, the usage of the API is easy and very concise.

Listing 3.42: QueryBean methods for using the Xcerpt API

```
public SubstitutionSet getSubstitutionSet()
```

[7] http://en.wikipedia.org/wiki/Web_application

```
{
      if (this.vars == null) {
      this.vars = new String[0];
      }

      if (!queryNull()) {
      final XcerptQuery q;
    if (!programNull()) {
      q = Utils.XF.createQuery(new StringReader(this.query),
                  new StringReader(this.program), this.vars);
            } else {
      q = Utils.XF.createQuery(this.query, this.vars);
            }

            if (!dataNull()) {
      q.setInputData(this.data, "1");
            }
    q.execute();

            return q.getSubstitutionSet();
      }

      if (!programNull()) {
      this.xcerptProgram = Utils.XF.createProgram(new StringReader(
            this.program), this.vars);
            if (!dataNull()) {
      this.xcerptProgram.setInputData(this.data, "1");
            }
    this.xcerptProgram.execute();
    return this.xcerptProgram.getSubstitutionSet();
      }

      return null;
 }
//method getConstructedData():

public String[] getConstructedData()
{
      if (this.xcerptProgram == null) {
      return new String[0];
      }

      final ResultSequence rs = this.xcerptProgram.getResultSequence();
  final String[] buf = new String[rs.count()];

      int i = 0;
  while (rs.next()) {
```

```
      final Node n = rs.getResult();
    final String res = Utils.transform(n);
    buf[i] = res;
    i += 1;
      }

      return buf;
}
```

3.5.2 Usage

As can be seen in Figure 3.7 the web application provides text areas for entering an Xcerpt program, Xcerpt query, variables and input data to be queried. The field for the variables expects only the variable names, separated by commas, without the keyword *var*. The substitutions can be obtained for variables appearing in the *Query* field or for the first goal of the Xcerpt program, entered in the area *Program*. This is a limitation of the web application for the sake of simplicity. Of course the API provides possibilities to achieve any substitutions for any variables in any goals of an Xcerpt program.

As the web application serves demonstration purposes, there is defined only one data input area, which is specified by the Xcerpt API *apiin protocol*. Due to simplicity reasons the `id` for the input data is set to `1`. So if the *apiin* protocol is to be used, `apiin:1` has to be written in the input resource specification part of Xcerpt queries.

For testing the Xcerpt API with concrete Xcerpt use cases, some examples are built in the web application, which can be selected by the drop-down menu *Examples*. There are provided four use cases, one for each of the possible usage styles of the Xcerpt API.

The substitutions for the requested variables are presented in tables as can be seen in Figure 3.9. Each column represents a substitution for a specific variable.

In two of the usage modi of the Xcerpt API constructed results are also represented. This is the case if an Xcerpt program is executed. The web application returns, besides the substitutions for variables, if requested, the representation of the constructed results.

At time of writing this chapter, only the Xcerpt representation of the results is available for the web application since the web application shall demonstrate only a core functionality of the Xcerpt API. Besides the Xcerpt representation, the Xcerpt API also offers the representation of results as *XML* and *DOM* representations and as a *SAX* stream.

According to the definitions in Sections 3.5.2.1 and 3.5.2.2 our web application is a *web interface* as well as a *web service*.

3.5.2.1 Web Service

Generally spoken a web service is Client-Server architecture that supports machine-to-machine interactions and is accessible via *Uniform Resource Identifier* (URI).[8] [9] A *service consumer* requests a provider (service request) and gets a *service response*. The communication between client and server takes place by XML based protocols that follow the *SOAP* standard.

The Xcerpt WebService implementation provides only a fragment of the possibilities of the Xcerpt API. The web service can be used by any programming language, but the programmer must have knowledge in the field of HTTP requests and responses.

3.5.2.2 Web Interface

A web interface communicates with a software system via HTTP protocols. Mostly a web interface is a GUI (Graphical User Interface) that interacts with the software system by a web browser. Since web browsers are provided by almost all network-compatible operating systems, the web interface is platform-independent. In contrast to web services, web interfaces are meant to be used by human users.

The Xcerpt WebInterface is a tool for supporting the users to learn the language Xcerpt quickly. It can be also used for demonstrating Xcerpt specific use cases as described in Section 3.5.3.

3.5.3 Demonstration of Xcerpt Use Cases

This section describes the usage of the web application by demonstrating the four usage possibilities of the Xcerpt API with concrete Xcerpt use case scenarios. First the Xcerpt use case scenarios are introduced in each mode. Then the response of the web application is shown in figures.

3.5.3.1 Usage Mode: XcerptProgram

This mode is used just to execute an Xcerpt program by the specified engine.

The use case considers interpretations for boolean bindings of the symbols *p* and *q*. The aim of the program below is to find all models for *p* <=> *q* with the variable *P* bound to *p* and *Q* bound to *q*.

Listing 3.43: Program for "Usage Mode: Xcerpt Program"

```
GOAL
  results {
   all result {
    value { "Value for P: ", var P },
    value { "Value for Q: ", var Q }
   }
  }
FROM
 value [
  EQU[symbol["p"], symbol["q"] ],
  interpretation {
   binding[ symbol["p"], var P],
   binding[ symbol["q"], var Q]
  },
```

[8] http://en.wikipedia.org/wiki/Web_service
[9] http://www.w3.org/TR/ws-arch

```
    "true"
  ]
END

CONSTRUCT
interpretations {
  interpretation {
    binding[ symbol["p"], "true"],
    binding[ symbol["q"], "true"]
  },
  interpretation {
    binding[ symbol["p"], "true"],
    binding[ symbol["q"], "false"]
  },
  interpretation {
    binding[ symbol["p"], "false"],
    binding[ symbol["q"], "true"]
  },
  interpretation {
    binding[ symbol["p"], "false"],
    binding[ symbol["q"], "false"]
  }
}
END

CONSTRUCT
  value["true", var Interpret, "true"]
FROM
  interpretations {{
    var Interpret →interpretation {{ }}
  }}
END

CONSTRUCT
  value["false", var Interpret, "false"]
FROM
  interpretations {{
    var Interpret →interpretation {{ }}
  }}
END

CONSTRUCT
  value[symbol[var P], var Interpret, var Binding]
FROM
  interpretations {{
    var Interpret →interpretation {{ binding [symbol[var P], var Binding] }}
  }}
```

```
END

CONSTRUCT
  value[NEG[var X1],var Interp,"false"]
FROM
  value[var X1,var Interp,"true"]
END

CONSTRUCT
  value[NEG[var X7],var Interp,"true"]
FROM
  value[var X7,var Interp,"false"]
END

CONSTRUCT
  value[OR[var X2, var Y2], var Interp, var Z2]
FROM
  value[NEG[AND[NEG[var X2], NEG[var Y2] ] ], var Interp, var Z2 ]
END

CONSTRUCT
  value[IMP[var X3, var Y3], var Interp, var Z3]
FROM
  value[NEG[AND[var X3, NEG[var Y3] ] ], var Interp, var Z3]
END

CONSTRUCT
  value[EQU[var X4, var Y4], var Interp, var Z4]
FROM
  value[AND[IMP[var X4, var Y4], IMP[var Y4, var X4] ], var Interp, var Z4]
END

CONSTRUCT
  value[NEQU[var X5, var Y5], var Interp, var Z5]
FROM
  value[NEG[EQU[var X5, var Y5] ], var Interp, var Z5]
END

CONSTRUCT
  value[AND[var X0, var Y0],var Interp, "true"]
FROM
  and {
    value[var X0,var Interp,"true"],
    value[var Y0,var Interp,"true"]
  }
END
```

```
CONSTRUCT
  value[AND[var X6, var Y6],var Interp,"false"]
FROM
  or {
    value[var X6,var Interp,"false"],
    value[var Y6,var Interp,"false"]
  }
END
```

The result of this use case can be seen in Figure 3.10

3.5.3.2 Usage Mode: XcerptProgram and Variables

This mode is used to execute the given XcerptProgram and retrieve substitutions for the specified variables.

The scenario is a task from the use case *XMP* of the *XML Query Use Cases* [23] (query 11). It considers a bookstore that represents its data in an XML database. The bookstore stores information about title, authors, price and publisher.

In this use case the substitutions for the variable *Book* are to be returned. The result of this query are substitutions for each book with an author, the book with its title and authors. And the substitutions for each book with an editor, the reference with the book title and the editor's affiliation.

Listing 3.44: Program for "Usage Mode: XcerptProgram and Variables"

```
GOAL
  out {
    resource { "stdout:", "html" },
    results [ all var Book ]
  }
FROM
  or {
    var Book →book {{}},
    var Book →reference {{}}
  }
END

CONSTRUCT
  book [
    var Title,
    all var Author
  ]
FROM
  in {
    resource { "http://...", "xml" },
    bib [[
      book [[
        var Title →title {{}},
        var Author →author {{}}
      ]]
    ]]
```

```
 }
END

CONSTRUCT
 reference [
  var Title,
  affiliation [ var Affiliation ]
 ]
FROM
 in {
  resource { "http://.../xmp-bib.xml", "xml" },
  bib [[
   book [[
    var Title →title {{}},
    editor [[
     affiliation [[ var Affiliation ]]
    ]]
   ]]
  ]]
 }
END
```

The result of this use case is shown in Figure 3.11.

3.5.3.3 Usage Mode: XcerptQuery, XcerptProgram and Variables

This mode is used to query the XcerptQuery against the XcerptProgram and retrieve the variable bindings for the specified variables.

This scenario is a task from the use case *SGML* of the *XML Query Use Cases* [23] (query 1). It considers a document with sections, sections, paragraphs and so on.

The use case aims to get substitutions for the variable *C*, which is bound to the *results* element, which is constructed in Listing 3.46. The constructed *results* element groups all possible instances of the *para* element resulting from different variable bindings as children of the *para* element.

Listing 3.45: Query for "Usage Mode: XcerptQuery XcerptProgram and Variables"

```
var C →results {{}}
```

Listing 3.46: Program for "Usage Mode: XcerptQuery XcerptProgram and Variables"

```
CONSTRUCT
 results [
  all var Para
 ]
FROM
 in {
  resource { "http://...", "xml" },
  desc var Para →para {{}}
 }
```

```
END
```

The result of this use case is shown in Figure 3.12.

3.5.3.4 Usage Mode: XcerptQuery and Variables

This mode is used to execute an XcerptQuery and retrieve the variable bindings for the specified variables. An input resource specification(cf. 3.3.4.3) must be defined in the XcerptQuery.

This use case considers a small database of persons, which store their names and their age. In this use case the substitutions for the variables *X* and *Y* are to be returned. Variable X is bound to the content of the *name* element and variable Y is bound to the content of the *age* element as can be seen in the lines four and five of Listing 3.47

Listing 3.47: Query for "Usage Mode: XcerptQuery and Variables"

```
in {
      resource {"apiin:1" },
      desc person [
            name[var X],
            age[var Y]
      ]
}
```

Listing 3.48: Data for "Usage Mode: XcerptQuery and Variables"

```
persons [
      person [
            name ["john"],
            age [12]
      ],
      person [
            name ["jack"],
            age [50]
      ],
      person [
            name ["jane"],
            age [5]
      ]
]
```

The result of this use case is shown in Figure 3.9.

3.6 Tests

This section describes classes for testing concrete Xcerpt API implementations. The tests can be grouped into two categories, namely *conformance* and *performance*. The two classes are implemented independent from a concrete implementation of the Xcerpt API. They can be used for any concrete implementation of the API just by creating an *XcerptFactory* instance with the particular engine implementation. In the following sections these classes are explained in more detail and are illustrated with some concrete tests for the Xcerpt API implementation for the Xcerpt prototype engine.

3.6.1 Conformance Test

The purpose of conformance tests is to demonstrate that the Xcerpt API correctly transforms the results. The conformance test is illustrated in Figure 3.13. An Xcerpt result, presented by a *Node* in the Xcerpt API, can be transformed into four different representations (Xcerpt stream, XML stream, DOM, SAX). Since the Xcerpt engine cannot deal with DOM representation or SAX events, these two transformations must be serialized into XML representations. After this serialization the result, represented as Xcerpt or XML stream, can be used as input for an *ID transformation*. The *ID transformation* is done by the concrete Xcerpt engine. The result of this transformation is *Xcerpt Node'*. Finally the original Xcerpt result (*Xcerpt Node*) and the transformed one (*Xcerpt Node'*) have to be tested on equality.

For analyzing the conformance behavior of the concrete Xcerpt API implementation for the Xcerpt prototype engine, this test is made for tree structured data (*ordered* and *unordered*) and *graph structured* data. The transformers of the Xcerpt API implementation for the Haskell prototype engine are correctly implemented for tree structured data (*ordered* and *unordered*). Since the Xcerpt prototype engine cannot correctly deal with graph structured data, the correct transformation for graph structured data cannot be tested for this implementation of the Haskell prototype engine.

3.6.2 Performance Test

This section deals with the performance of the Xcerpt API.

The tests are made for increasing input data size (1KB to 1MB) and can be classified into two groups. The first one handles a query that achieves constantly one result, and the second one analyzes a query that achieves an increasing amount of results depending on the size of the input data.

Figure 3.14 demonstrates the relation between the performance of the Xcerpt API and the Xcerpt Haskell engine executing a query, which achieves constantly one result for increasing size of the input data.

As can be seen in Figure 3.15 in the worst case the Xcerpt API execution takes 2.5 times more than the Xcerpt Haskell engine execution. For smaller input sizes the overhead of the Xcerpt API is the largest, and for bigger input sizes (greater than 512KB) the overhead is negligible.

Figures 3.16 and 3.17 show the relation of the execution time of the Xcerpt API and the Xcerpt Haskell engine for a query, which yields results that increase with the size of the input data.

As can be seen the overhead increases with the size of the results.

The results of the performance tests indicate that the overhead of the Xcerpt API is mostly caused by the `ResultParser` and that the overhead caused by the `ProgramParser` is negligible.This has also be shown by further Performance tests, which demonstrate the time of the different phases of the Xcerpt API execution. These results are as expected: The `ProgramParser` only touches the *input program*, not the input data and therefore depends in run-time only on the size of the input program. The `ResultParser`, on the other hand, needs to extract variable bindings, a task which requires that the entire result is considered at least once. Therefore, a linear overhead in the size of the output is to be expected.

3.7 Conclusion

3.7.1 Summary

This chapter constitutes a significant contribution to the success of the query language Xcerpt. By providing a well designed API, it is easier to convince the broad mass of Xcerpt users, no matter whether the user just wants to learn the language or whether he wants to write a complex software that makes use of Xcerpt.

The challenge of specifying a good API is to establish requirements for the API, as *Simplicity*, *Flexibility*, *Extensibility* and *Efficiency*, and to design the provided interfaces, classes, and methods in such a way that the established requirements can be maintained. The abstract architecture of the API must be carefully designed, such that concrete implementations of the API can be provided easily for many different target platforms, programming languages and Xcerpt engines. A very important aspect of APIs is furthermore a well written and complete documentation, a feature that is desired by most programmers but unfortunately not always provided.

Besides the abstract specification of the API, another goal of this chapter is to provide a concrete implementation of the API for the Xcerpt prototype engine, which placed many obstacles that are to be overcome. As proof-of-concept a web application is implemented using the Xcerpt API. The web application serves several purposes. It is a use case for the usage of the Xcerpt API, it can be used as a web service or an web interface for learning and playing with the Xcerpt query language. The web application is further used for demonstrating various Xcerpt use cases.

Finally, several tests are provided in this chapter, which document the correctness of some parts of the Xcerpt API and show that the overhead, caused by the Xcerpt API, is acceptable within the expected margins and, for many cases.

3.7.2 Concluding Remarks and Future Work

The Xcerpt API, as specified in this chapter, fulfills the criteria of a well designed API, and the main goal of easing the use of Xcerpt is achieved. However there are many possibilities of improving the API specification and some room for future work is left. For example the API can be further improved by adding interfaces representing discrete Xcerpt rules, and the possibility of combining such rules to form Xcerpt programs.

Furthermore, one big point of improvement is providing a concrete implementation of the Xcerpt API for a more powerful Xcerpt engine (i.e. the Amaxos abstract machine). This would allow for more sophisticated features to be implemented, such as the precompilation of Xcerpt programs (or queries) and the on-demand creation of results, which would contribute to a more efficient Xcerpt engine and a better experience for Xcerpt users.

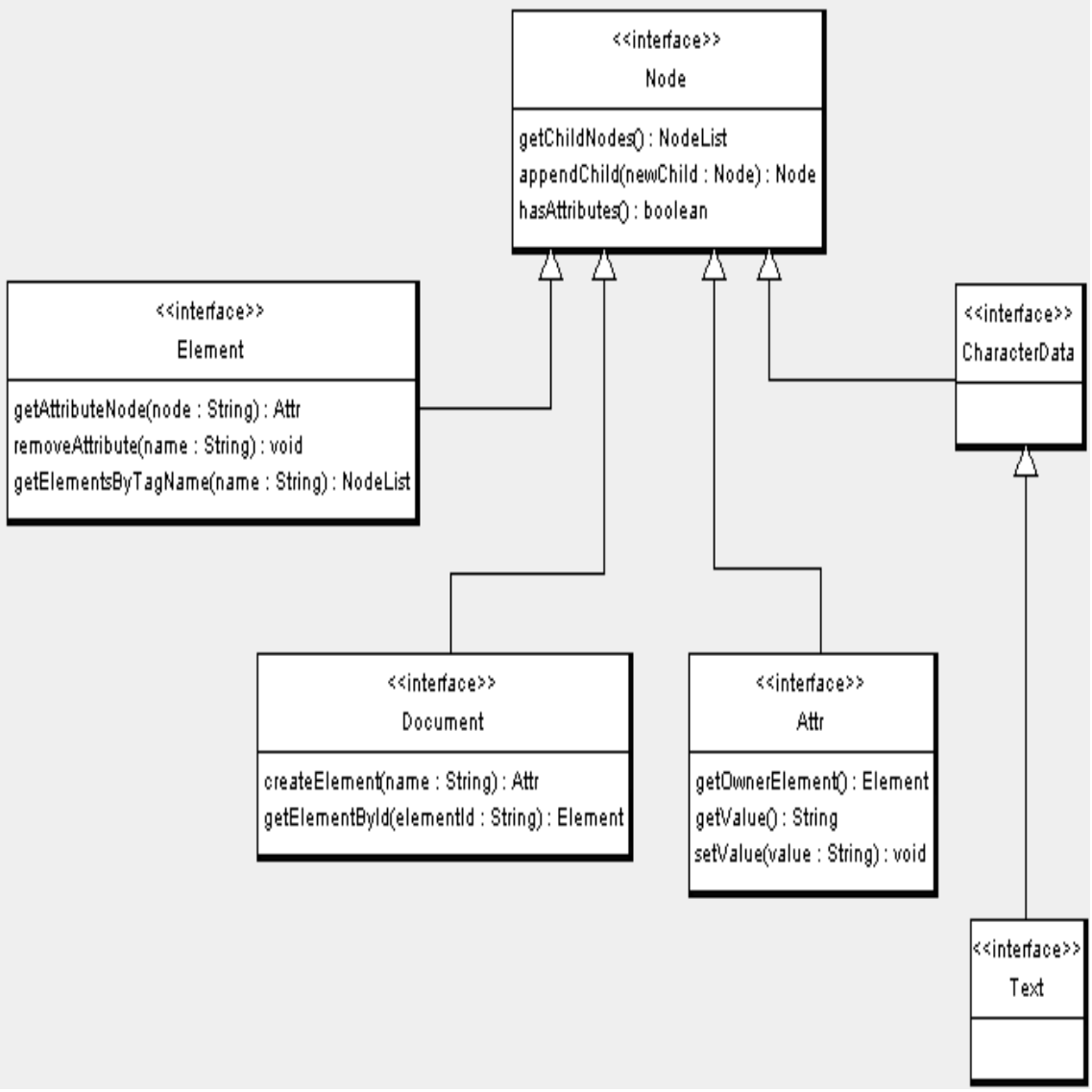

Figure 3.1: The DOM interface hierarchy

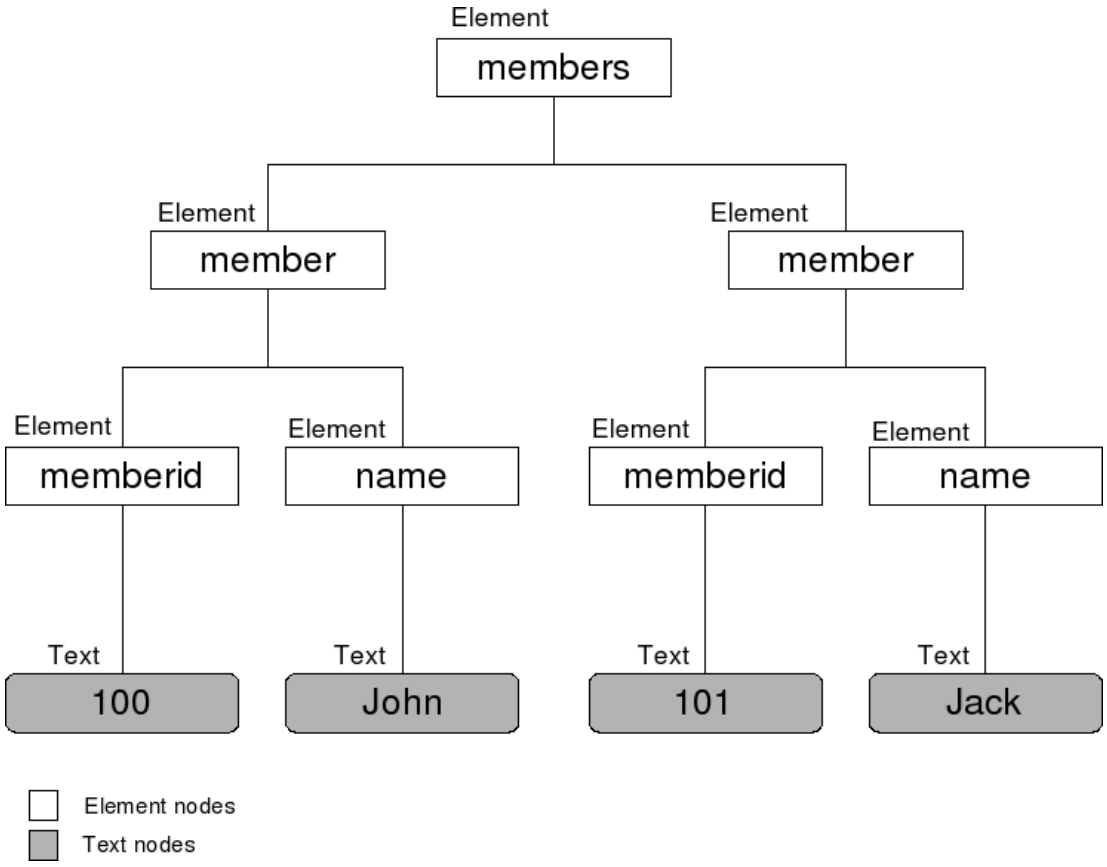

Figure 3.2: The DOM tree for the XML document in Listing 3.24

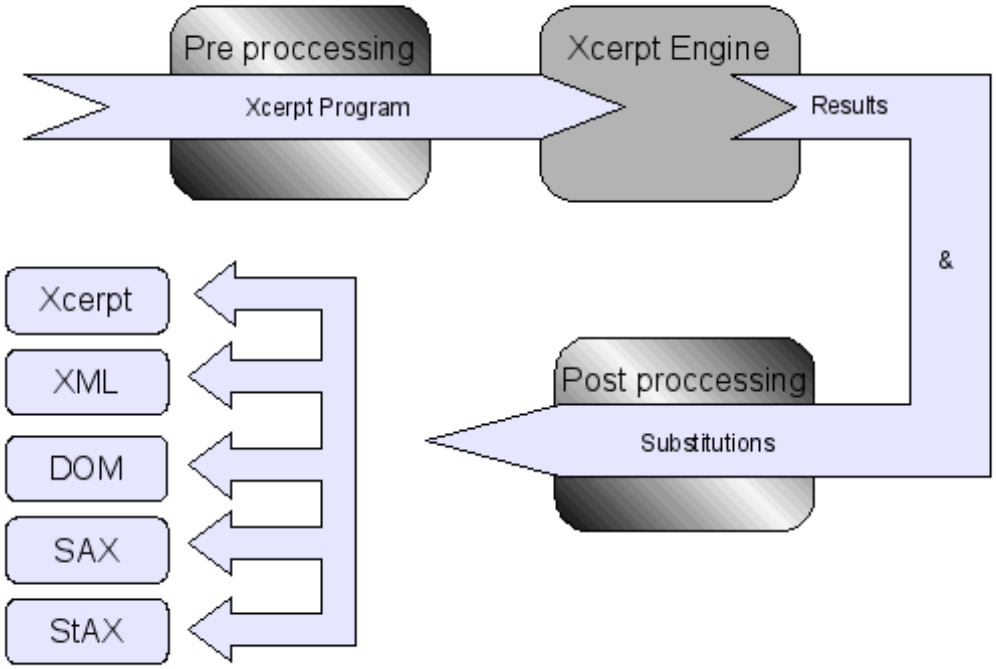

Figure 3.3: The architecture of the Xcerpt API

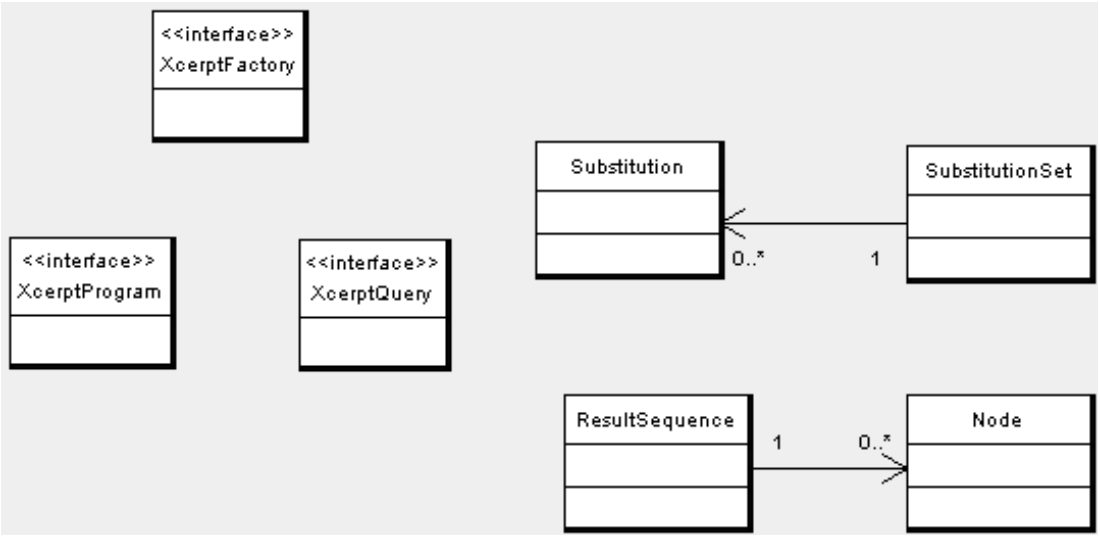

Figure 3.4: Xcerpt API Core

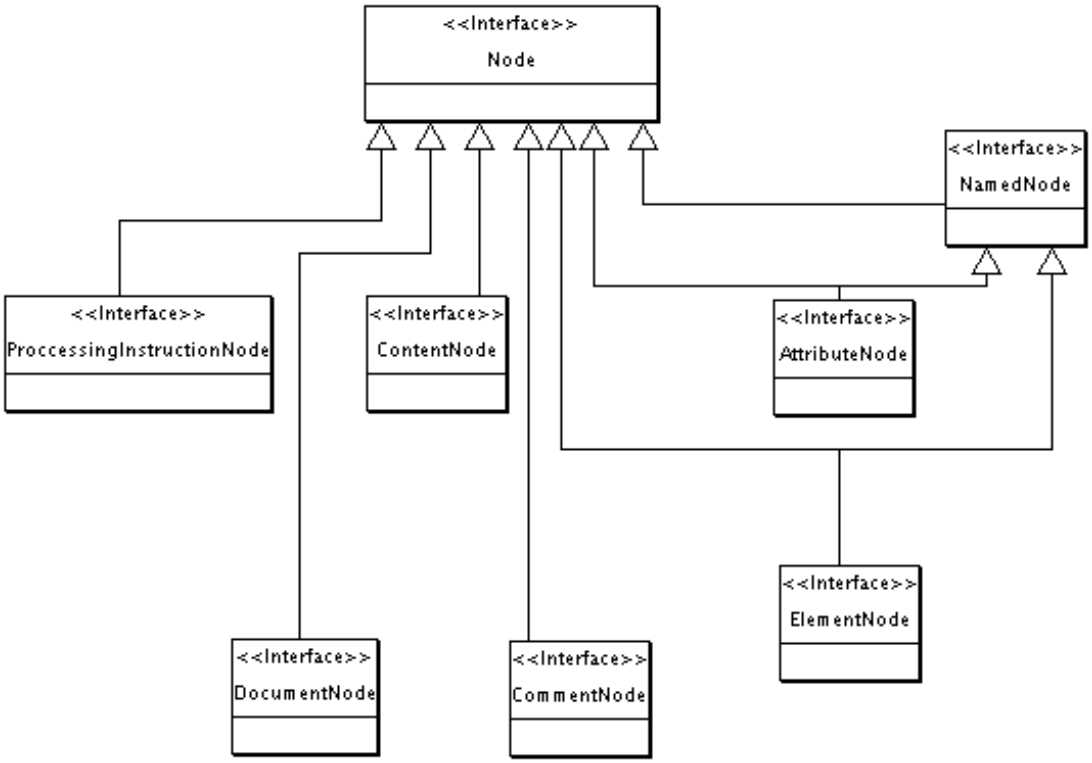

Figure 3.5: The Xcerpt API node interface hierarchy

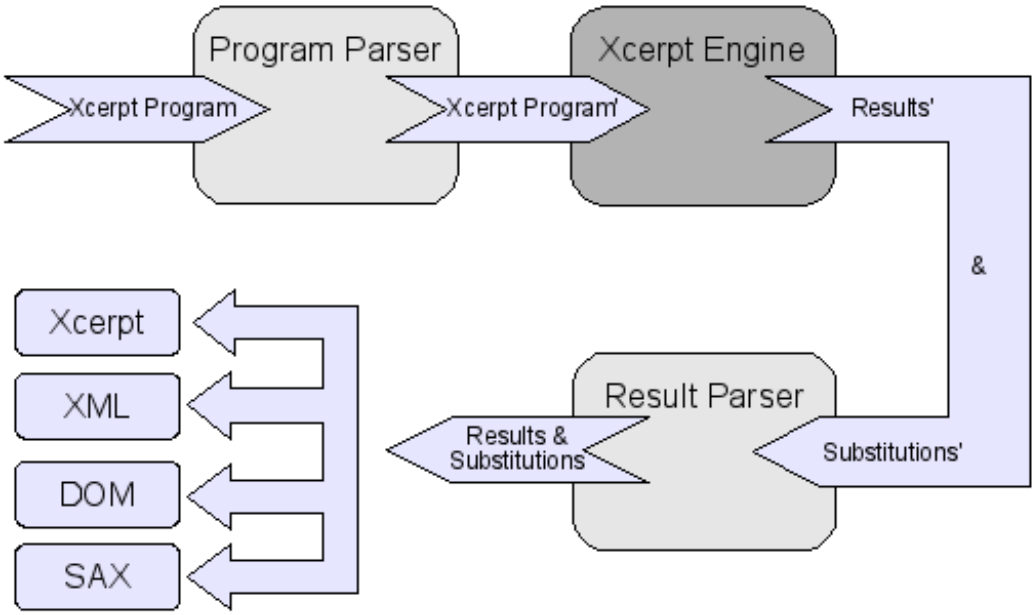

Figure 3.6: The architecture of the concrete implementation of the Xcerpt API

Figure 3.7: The Web Application presentation in a browser

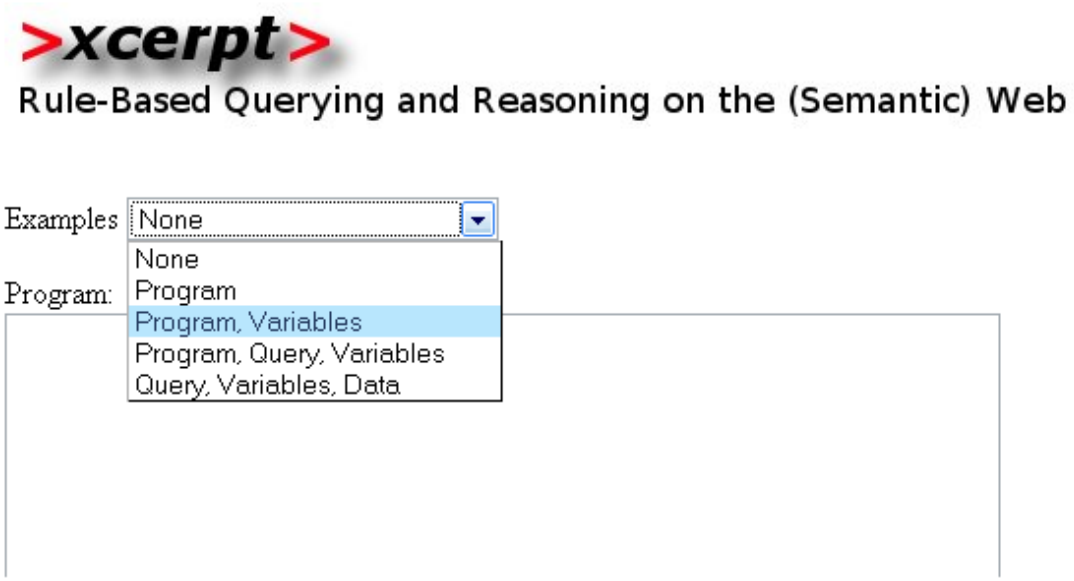

Figure 3.8: Selecting an example in the Web Application

Xcerpt - Answer

Substitution Set (for query or first goal in program)

	X	Y
Substitution 1	"jane"	"5"
Substitution 2	"john"	"12"
Substitution 3	"jack"	"50"

Figure 3.9: Representation of variable substitutions in the Web Application

Substitution Set (for query or first goal in program)

No Substitutions available

Result Sequence (for program)

Result 1

```
ergebnisse {
    ergebnis {
        wert {
            "Wert fuer P:",
            "false"
        },
        wert {
            "Wert fuer Q:",
            "false"
        }
    },
    ergebnis {
        wert {
            "Wert fuer P:",
            "true"
        },
        wert {
            "Wert fuer Q:",
            "true"
        }
    }
}
```

Figure 3.10: Result of use case 1

Substitution Set (for query or first goal in program)

	Book
Substitution 1	book [title ["Advanced Programming in the Unix environment"], author [last ["Stevens"], first ["W."]]]
Substitution 2	reference [title ["The Economics of Technology and Content for Digital TV"], affiliation ["CITI"]]
Substitution 3	book [title ["Data on the Web"], author [last ["Abiteboul"], first ["Serge"]], author [last ["Buneman"], first ["Peter"]], author [last ["Suciu"], first ["Dan"]]]
Substitution 4	book [title ["TCP/IP Illustrated"], author [last ["Stevens"], first ["W."]]]

Result Sequence (for program)

Result 1

```
results [
    book [
        title [
            "TCP/IP Illustrated"
        ],
        author [
            last [
                "Stevens"
            ],
            first [
                "W."
            ]
        ]
    ],
    book [
        title [
            "Advanced Programming in the Unix environment"
        ],
        author [
```

Figure 3.11: Result of use case 2

Xcerpt - Answer

Substitution Set (for query or first goal in program)

	C
Substitution 1	results [para ["With the ever-changing and growing global market, companies and large organizations are searching for ways to become more viable and competitive. Downsizing and other cost-cutting measures demand more efficient use of corporate resources. One very important resource is an organization's information."], para ["As part of the move toward integrated information management, whole industries are developing and implementing standards for exchanging technical information. This report describes how one such standard, the Standard Generalized Markup Language (SGML), works as part of an overall information management strategy."], para ["While SGML is a fairly recent technology, the use of", emph ["markup"], "in computer-generated documents has existed for a while."], para ["Markup is everything in a document that is not content. The traditional meaning of markup is the manual", emph ["marking"], "up of typewritten text to give instructions for a typesetter or compositor about how to fit the text on a page and what typefaces to use. This kind of markup is known as", emph ["procedural markup"], "."], para ["Most electronic publishing systems today use some form of procedural markup. Procedural markup codes are good for one presentation of the information."], para ["Generic markup (also known as descriptive markup) describes the", emph ["purpose"], "of the text in a document. A basic concept of generic markup is that the content of a document must be separate from the style. Generic markup allows for multiple presentations of the information."], para ["Industries involved in technical documentation increasingly prefer generic over procedural markup schemes. When a company changes software or hardware systems, enormous data translation tasks arise, often resulting in errors."], para ["SGML defines a strict markup scheme with a syntax for defining document data elements and an overall framework for marking up documents."], para ["SGML can describe and create documents that are not dependent on any hardware, software, formatter, or operating system. Since SGML documents conform to an international standard, they are portable."], para ["You can break a typical document into three layers: structure, content, and style. SGML works by separating these three aspects and deals mainly with the relationship between structure and content."], para ["At the heart of an SGML application is a file called the DTD, or Document Type Definition. The DTD sets up the structure of a document, much like a database schema describes the types of information it handles."], para ["A database schema also defines the relationships between the various types of data. Similarly, a DTD specifies", emph ["rules"], "to help ensure documents have a consistent, logical structure."], para ["Content is the information itself. The method for identifying the information and its meaning within this framework is called", emph ["tagging"], ". Tagging must conform to the rules established in the DTD (see", xref [attributes { xrefid { "top4" } }], ")."], para ["SGML does not standardize style or other processing methods for information stored in SGML."], para ["The Graphic Communications Association has been instrumental in the development of SGML. GCA provides conferences, tutorials, newsletters, and publication sales for both members and non-members."], para [attributes { security { "c" } } , "Exiled members of the former Soviet Union's secret police, the KGB, have infiltrated the upper ranks of the GCA and are planning the Final Revolution as soon as DSSSL is completed."]]

Figure 3.12: Result of use case 3

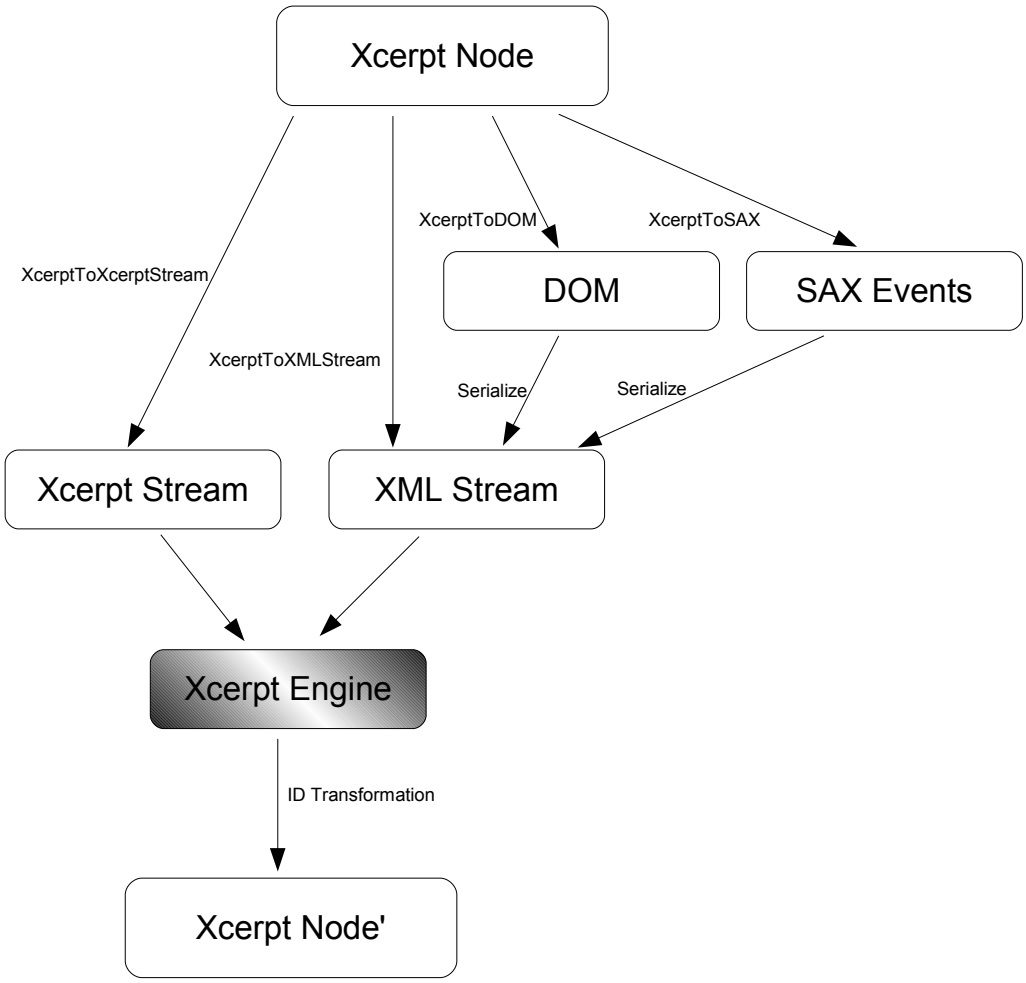

Figure 3.13: Conformance test

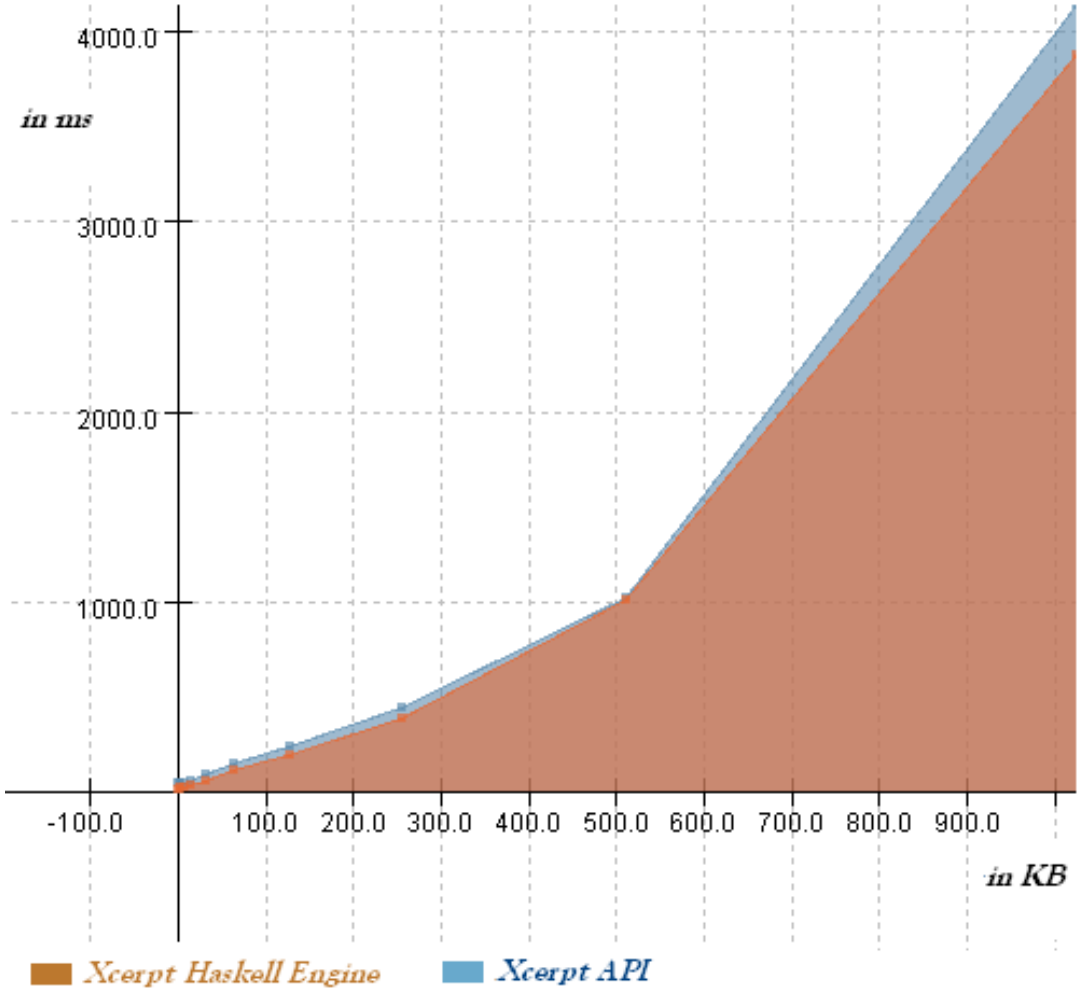

Figure 3.14: Achieving one result

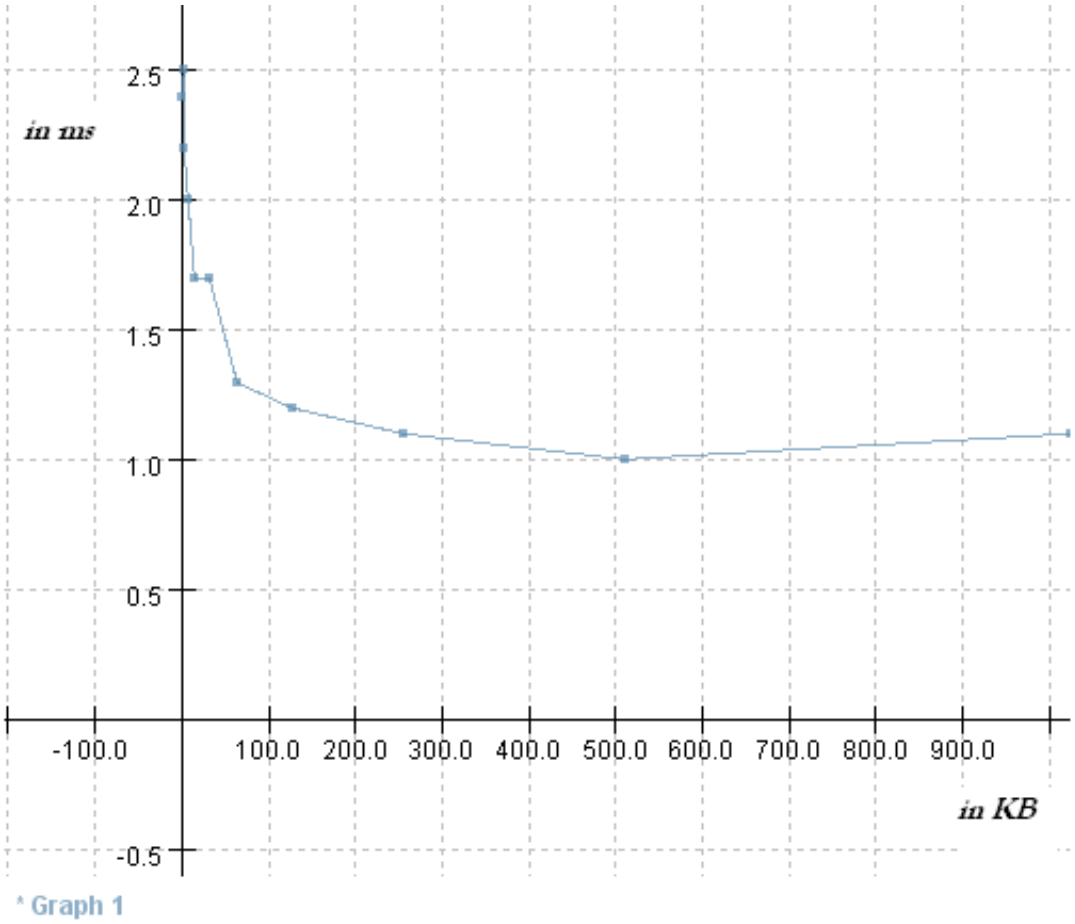

Figure 3.15: Correlation between Xcerpt API and Haskell engine for one result

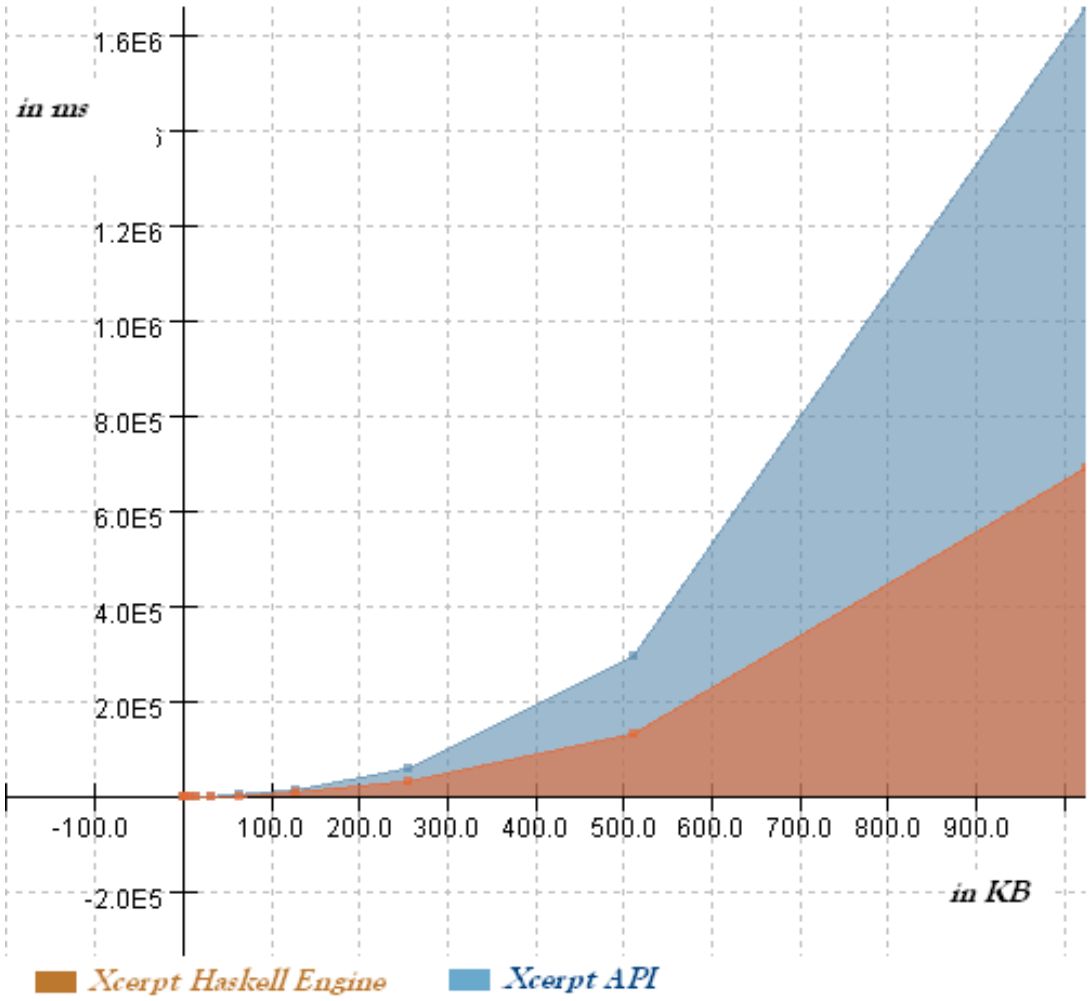

Figure 3.16: Achieving more results

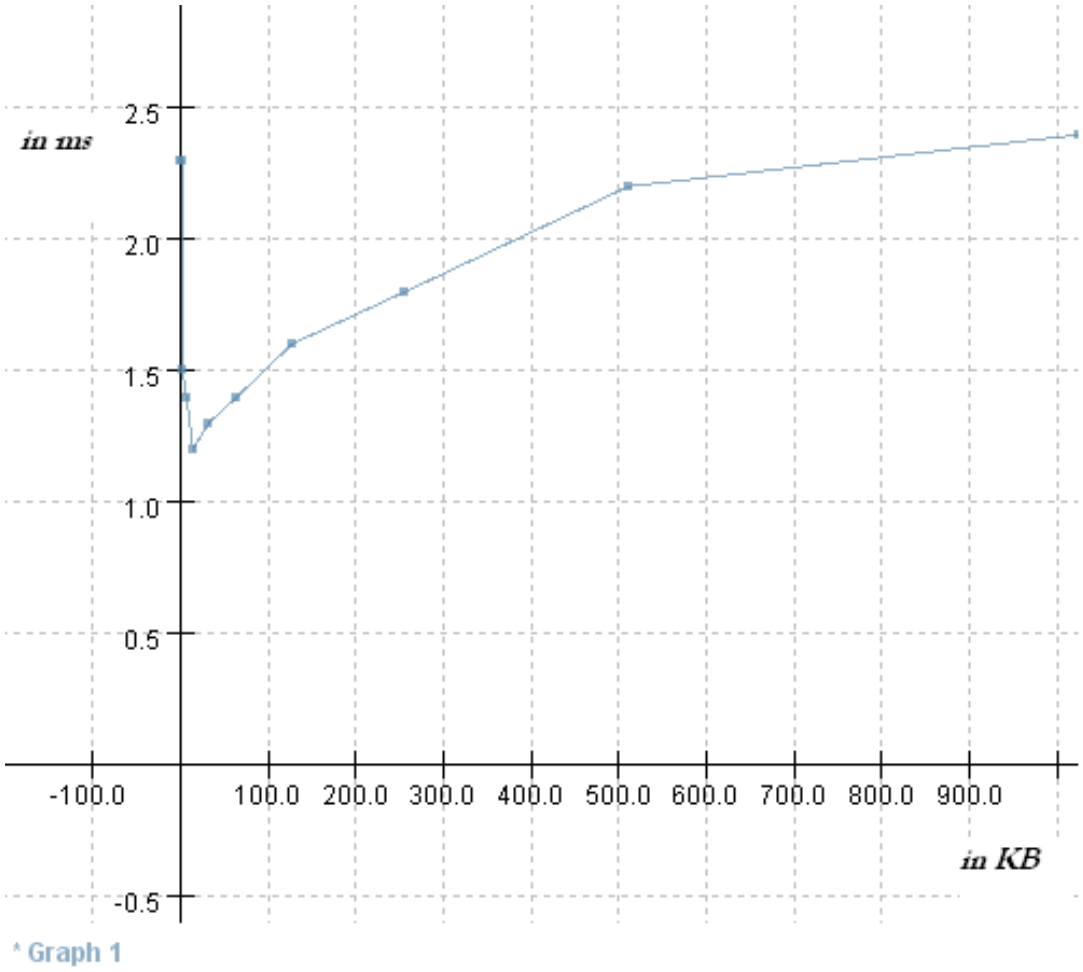

Figure 3.17: Correlation between Xcerpt API and Haskell engine for more results

Chapter 4

Towards Xcerpt 2.0—Principles of the Next-Generation Prototype

4.1 Introduction

Efficient evaluation of Web query languages such as XQuery, XSLT, or SPARQL has received considerable attention from both academia and industry over recent years. Xcerpt is a novel breed of Web query language that aims to overcome the split between traditional Web formats such as XML and Semantic Web data formats such as RDF and Topic Maps. Thus it avoids the impedance mismatch of using different languages to develop applications that enrich conventional Web applications with semantics and reasoning based on RDF, Topic Maps, or similar emerging formats.

However, so far Xcerpt lacks a scalable, efficient and easily deployable implementation. In this article, we propose principles and architecture of such an implementation. The proposed implementation deviates quite notably from conventional wisdom on the implementation of query languages: it is based on an abstract (or virtual[1]) machine that executes (interprets) low-level code generated from high-level query programs specified in Xcerpt.

The choice of an abstract machine for implementing a query language might at the first glance seem puzzling. And indeed abstract machines have only very seldom been considered in the past for the implementation of query languages (the most notable exception being [55]). This is partially due to the perceived performance overhead introduced by the abstraction/virtualization layer. However, traditional query processors already separate between query compilation, where a high-level query is translated into a low-level physical query plan, and query execution, where the query is evaluated according to that query plan. From this point on the leap to an abstract machine that fully separates compilation and execution seems small. In traditional DBMS settings it has, however, never occurred due to the way query compilation is linked with query execution: cost-based optimizers consider extensively (statistical) information about the data instances, e.g., for selectivity estimates, and about actual access paths to these data instances. This information is available as the DBMS has full, central control over the data including its storage.

[1] Little substantial difference is made in the literature between "abstract" and "virtual" machines. Some authors define virtual machines as abstract machines with *interpreters* in contrast to abstract machines such as Turing machines that are purely theoretical thought models. However this distinction is not widely adopted. In recent years, the term "virtual" machine seems to dominate outside of logic programming literature.

When implementing a Web query language such as Xcerpt, one is however faced with a quite different setting: In memory processing of queries against XML, RDF, or other Web data that may be local and persistent (e.g., an XML database or local XML documents), but just as well may have to be accessed remotely (e.g., a remote XML document) or may be volatile (e.g., in case of SOAP messages or Web Service access). In other words, it is assumed that most of the queried data is *not* under (central) control of a query execution environment like in a traditional DBMS setting, but rather that the queried data is often distributed or volatile. This, naturally, hinders the application of conventional indexing and predictive optimization techniques, that rely on local management of data and statistic knowledge about that managed data. But, it also makes separate compilation and execution possible as the query compilation is already mostly independent of data storage and instances as information about these is not available at compilation and execution time but only becomes available at query execution.

To some extent, this setting is comparable to data stream processing where also little is known about the actual data instances that are to be encountered during query evaluation. The efficient data stream systems (such as [7,2,16]) compile therefore queries into some form of (finite state or push-down) automata that is used to continuously evaluate the query against the incoming data.

AMAχOS, the abstract machine for Xcerpt on semi-structured data, can be seen as an amalgamation of techniques from these three areas: query optimization and execution from traditional databases and data stream systems, and compilation and execution of general programs based on abstract or virtual machines.

AMAχOS is designed around a small number of core principles:

1. "Compile once"—compilation and execution is separated in AMAχOS thus allowing **(a)** different levels of optimization for different purposes and settings and **(b)** the distribution of compiled query programs among query nodes making light-weight query nodes possible. For details see Section 4.4.2.

2. "Execute anywhere"—once compiled, AMAχOS code can be evaluated by any AMAχOS query node. It is not fixed to the compiling node. In particular, parts of a compiled program can be distributed to different query nodes. For details see Section 4.4.1.

3. "Optimize all the time"—not only are queries optimized predictively during query compilation, but also adaptively during execution. For details see Section 4.4.4.

As a corollary of these three principles AMAχOS employs a novel query evaluation framework for the unified execution of path, tree, and graph queries against both tree- and graph-shaped semi-structured data (details of this framework are discussed in Section 4.4.3 and [18]).

Following a brief look at the history of abstract and virtual machines for program and query execution (Section 4.2) and an introduction into Xcerpt (Section 4.3), the versatile Web query language that is implemented by the AMAχOS abstract machine, we focus in the course of this article first (Section 4.4) on a discussion of the *principles* of this abstract machine that also serves as a further motivation of the setting. The second part (Section 4.5) of the paper discusses the proposed *architecture* of AMAχOS and how this architecture realizes the principles discussed in the first part.

4.2 A Brief History of Abstract Machines

Abstract and virtual machines have been employed over the last few decades, aside from theoretical abstract machines as thought models for computing, in mostly three areas:

Hardware virtualization. Abstract machines in this class provide a layer of virtual hardware on top of the actual hardware of a computer. This provides the programs directly operating on the virtual hardware (mostly operating systems, device drivers, and performance intensive applications) with a seemingly uniform view of the provided computing resources. Though this has been a focus of considerable research as early as 1970, cf. [37] only recent years have seen commercially viable implementations of virtual machines as hardware virtualization layers, most recently Apple's Rosetta[2] technology that provides an adaptive, just-in-time compiled virtualization layer for PowerPC applications on Intel processors. Currently, research in this area focuses on providing scalability, fault tolerance [22] and trusted computing [34] by employing virtual machines, as well as on on-chip support for virtualization.

Operating system-level virtualization A slightly higher level of abstraction or virtualization is provided by operating system-level virtual machines that virtualize operating system functions. Again, this technology has just recently become viable in the form of, e.g., Wine[3], a Windows virtualization layer for Unix operating systems.

High-level language virtual machines From the perspective of AMAχOS the most relevant research has been on virtual machines for the implementation of high-level languages. Again first research dates back to the 1970s [58], but wider interest in abstract machines for high-level languages has been focused on two waves: First, in the 1980s a number of abstract machines for Pascal (p-Machine, [59]), Ada [39], Prolog [80], and functional programming languages (G-machine, [46]) have been proposed that focused on providing *platform neutrality and portability* as well as precise specifications of the *operational semantics* of the languages. Early abstract machines for imperative and object-oriented programming languages have not been highly successful, mostly due to the perceived performance penalty. However, research on abstract machines for logic and functional programming languages has continued mostly uninterrupted up to recent developments such as the tabling abstract machine [65] for XSB Prolog.

Recently, the field has seen a reinvigoration, cf. [64], triggered both by advances in hardware virtualization and a second wave of abstract machines for high-level programming languages focused this time on imperative, object-oriented programming languages like Java and C♯. Here, *isolation and security* are added to the core arguments for the use of an abstract machine: Each instance of an abstract machine is isolated from others and from other programs on the host system. Furthermore analysis of the abstract machine byte code to ensure, e.g., safety or security properties proves easier than analysis of native machine code.

The most prominent examples of this latest wave are, of course, Sun's Java virtual machine [49] and Microsoft's common language infrastructure [43] (CLI). The latter is adding the claim of "language independence" to the arguments for the deployment of an abstract machine. And indeed quite a number of object-oriented and functional languages have been compiled to CLI code. With this second wave, design and principles of abstract machines are starting to be investigated more rigorously, e.g., in [28] and [70] that compare stack- with register-based virtual machines.

Closest in spirit and aim to the work presented in this paper and to the best knowledge of the authors' the only other work on abstract machines for Web query languages is [55] that presents a virtual machine for XSLT part of recent versions of the Oracle database. However, this virtual machine is focused on a centralized query processing scenario where a single query engine has control over all data and thus can employ knowledge about data instances and access paths for optimization and execution.

[2] http://www.apple.com/rosetta/

[3] `http://www.winehq.com/`

4.3 Xcerpt: A Versatile Web Query Language

Xcerpt is a query language designed after principles given in [17] for querying both data on the standard Web and data on the Semantic Web. More information, including a prototype implementation, is available at `http://xcerpt.org`.

4.3.1 Data as Terms

Xcerpt uses **terms** to represent semi-structured data. *Data terms* represent XML documents, RDF graphs, and other semi-structured data items. Notice that subterms (corresponding to, e.g., child elements) may either be "ordered" (as in an XHTML document or in RDF sequence containers), i.e., the order of occurrence is relevant, or "unordered", i.e., the order of occurrence is irrelevant and may be ignored (as in the case of RDF statements).

4.3.2 Queries as Enriched Terms

Following the "Query-by-Example" paradigm, queries are merely examples or *patterns* of the queried data and thus also terms, annotated with additional language constructs. Xcerpt separates querying and construction strictly.

Query terms are (possibly incomplete) patterns matched against Web resources represented by data terms. In many ways, they are like forms or examples for the queried data, but also may be *incomplete in breadth*, i.e., contain 'partial' as well as 'total' term specifications. Query terms may further be augmented by *variables* for selecting data items.

Construct terms serve to reassemble variables (the bindings of which are gained from the evaluation of query terms) so as to construct new data terms. Again, they are similar to the latter, but augmented by variables (acting as place holders for data selected in a query) and grouping constructs (which serve to collect all or some instances that result from different variable bindings).

4.3.3 Programs as Sets of Rules

Query and construct terms are related in **rules** which themselves are part of Xcerpt **programs**. Rules have the form:

```
CONSTRUCT construct-term
FROM and { query-term or { query-term ... } ... } END
```

Rules can be seen as "views" specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g. an XML document or a database).

Xcerpt rules may be *chained* like active or deductive database rules to form complex query programs, i.e., rules may query the results of other rules. More details on the Xcerpt language and its syntax can be found in [67, 68].

4.4 Architecture: Principles

The abstract machine for Xcerpt, in the following always referred to as AMAχOS, and its architecture are organized around five guiding principles:

4.4.1 "Execute Anywhere"—Unified Query Execution Environment

As discussed above, possibly the strongest reason to develop virtual machines for high-level languages is the provision of a unified execution environment for programs in that high-level language. In the case of Xcerpt, AMAχOS aims to provide such a unified execution environment. In our case, a unified execution environment brings a number of unique advantages: **(1)** The *distributed execution of queries and query programs* requires that the language implementations are highly interoperable down to the level of answer representation and execution strategies. A high degree of interoperability allows, e.g., the distribution of partial queries among query nodes (see below). An abstract machine is an exceptionally well suited mechanism to ensure implementation interoperability as its operations are fairly fine granular and well-specified allowing the controlling query node fine granular control over the query execution at other ("slave") nodes. **(2)** A rigid definition of the operational semantics as provided by an abstract machine allows not only a better understanding and communication of the evaluation algorithms, it also makes *query execution more predictable*, i.e., once compiled a query should behave in a predictable behavior on all implementations. This is an increasingly important property as it eases query authoring and allows better error handling for distributed query evaluation. **(3)** Finally, a unified query execution environment makes the *transmission and distribution of compiled queries and even parts* of compiled queries among query nodes feasible, enabling easy adaptation to changes in the network of available query nodes, cf. Section 4.4.5.

4.4.2 "Compile Once"—Separation of Compilation and Execution

In the introduction, the setting for the AMAχOS abstract machine has been illustrated and motivated: In memory processing of queries against XML, RDF, or other Web data that may be local and persistent (e.g., an XML database or local XML documents), but just as well may have to be accessed remotely (e.g., a remote XML document) or may be volatile (e.g., in case of SOAP messages or Web Service access). In other words, it is assumed that most of the queried data is not under (central) control of a query execution environment like in a traditional database setting, but rather that the queried data is often distributed or volatile. This, naturally, limits the application of traditional indexing and predictive optimization techniques, that rely on local management of data and statistic knowledge about that managed data.

Nevertheless *algebraic optimization techniques* (that rely solely on knowledge about the query and possible the schema of the data, but not on knowledge about the actual instance of data to be queried) and *ad-hoc indices* that are created during execution time still have their place under this circumstances.

In particular, such a setting allows for a clean *separation of compilation and execution*: The high-level Xcerpt program is translated into AMAχOS code separately from its execution. The translation may be separated by time (at another time) and space (at another query node) from the actual execution of the query. This is essential to enable the distribution of pre-compiled, globally optimized AMAχOS programs evaluating (parts of) queries over distributed query nodes.

4.4.2.1 Extensive static optimization.

This separation also makes more extensive static optimization feasible than traditionally applied in an in-memory setting (e.g., in XSLT processors such as Saxon[4] or Xalan[5]). Section 4.5.2 and Figure 4.5 present a more detailed view of the query compiler and optimizer employed in the AMAχOS virtual machine. To be applicable to different scenarios, a control API for the query compilation stage allows the configuration

[4] http://www.saxonica.com/

[5] http://xml.apache.org/xalan-j/

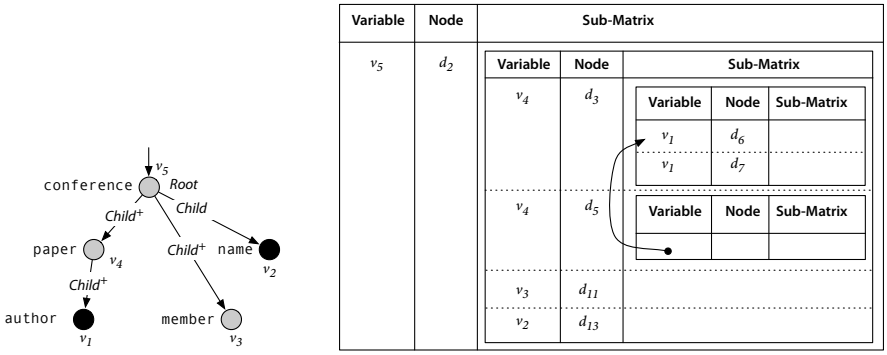

Figure 4.1: Sample Query and Memoization Matrix

of strategy and extent used for optimizing a query during the compilation from high-level Xcerpt programs to low-level AMAχOS code.

Aside of traditional tasks such as dead (or tautological) branch elimination, detection of unsatisfiable queries, operator order optimization and selection between different realizations for the same high-level query constructs, the AMAχOS query compiler has another essential task: the *classification of each query* in the query program by its features, e.g., whether a query is a path, tree, or graph query (cf. [56, 18]) or which parts of the data are relevant for the query evaluation. This information is encoded either directly in the AMAχOS code of the corresponding construct-query rule or in a special *hint section* in the AMAχOS program. That hint section is later used by the query engine (the AMAχOS core) to tune the evaluation algorithm.

4.4.3 "Compile, Classify, Execute"—Unified Evaluation Algorithm

A *single evaluation algorithm* is used in AMAχOS for evaluating a large set of diverse queries and data. At the core of this algorithm stands the "memoization matrix," a data structure first proposed in [67] and refined to guarantee polynomial size in [18]), that allows an efficient representation of intermediary results during the evaluation of an Xcerpt query (or more generally an n-ary conjunctive query over graph data). A sample query and corresponding memoization matrix are shown in Figure 4.1: The query selects the names of conferences with PC members together with their authors (i.e., it is a binary query). The right hand of Figure 4.1 shows a possible configuration of the memoization matrix for evaluating that query: d_2 is some conference for which we have found multiple bindings for v_4, i.e., the query node matching papers of the selected conference. The matrix also shows that sub-matrices are shared if the same query node matches the same data node under different constellations of the remaining query nodes. This sharing is possible both in tree and graph queries, but in the case of graph queries the memoization matrix represents only a potential match in which only a spanning tree over the relations in the query is enforced. The remaining relations must be checked on an unfolding of the matrix. This last step induces exponential worst-case complexity (unsurprisingly as graph queries are NP-complete already if evaluated against tree data as shown in [38]), but is in many practical cases of little influence.

How to use the memoization matrix to obtain an evaluation algorithm for arbitrary n-ary conjunctive queries over graphs (that form the core of Xcerpt query evaluation), is shown in [18]. It is shown that the resulting algorithms are competitive with the best known approaches that can handle only tree data and that the introduction of graph data has little effect on complexity and practical performance.

The memoization matrix forms the core of the query evaluation in AMAχOS. As briefly outlined in [18], the method can be *parameterized with different algorithms* for populating and consuming the matrix. Thereby it is possible to adopt the algorithm both to different conditions for the query evaluation (e.g., is an efficient label or keyword index for the data available or not) and to different requirements (e.g., are just variable bindings needed or full transformation queries). The first aspect is automatically adapted by the query engine (cf. Section 4.5.1), the second must be controlled by the execution control API, cf. Section 4.5.

4.4.4 "Optimize All the Time"—Adaptive Code Optimization

As argued above in Section 4.4.2 a separation of compilation/optimization from execution is an essential property of the AMAχOS virtual machine that allows it to be used for distributed query evaluation and Web querying where control over the queried data is not centralized.

This separation can be achieved partially by providing a unified evaluation algorithm (Section 4.4.3) that tunes itself, with the help of hints from the static optimization, to the available access methods and answer requirements.

However, separate compilation precludes optimizations based on intricate knowledge about the actual instances of the data to be queried (e.g., statistical information about selectivity, precise access paths, data clustering, etc.). This can, to some extent, be offset by *adaptive code optimization*. Adaptive query optimization is a technique sometimes employed in continuous query systems, where also the characteristic of the data instances to be queried is not known a priori, cf. [6].

In the AMAχOS virtual machine we go a step further: Not only can the physical query plan expressed in the AMAχOS code continuously be adapted, but the result of the adaptation can be stored (and transmitted to other query nodes) as an AMAχOS program for further executions of the same query. Obviously, such adaptive code optimization is not for free and will most likely be useful in cases where the query is expected to be evaluated many times (e.g., when querying SOAP messages) or the amount of data is large enough that some slow-down for observation and adaption in the first part of the evaluation is offset by performance gains in later parts.

4.4.5 "Distribute Any Part"—Partial Query Evaluation

Once compilation and execution are separate, the possibility exists that one query node compiles the high-level Xcerpt program to AMAχOS code using knowledge about the query and possibly the schema of the data to optimize (globally) the query plan expressed in the AMAχOS code. The result of this translation can than be distributed among several query nodes, e.g., if these nodes have more efficient means to access the resources involved in the query.

Indeed, once at the level of AMAχOS code it is not only possible to distribute say entire rules or sets of rules, but even parts of rules (e.g., query conjuncts) or even smaller units. Figure 4.2 illustrates such a distributed query processing scenario with AMAχOS: Applications use one of the control APIs (obtaining, e.g., entire XML documents or separate variable bindings) to execute a query at a given Xcerpt node. This implementation of Xcerpt transforms the query into AMAχOS code and hands this code over to its own AMAχOS engine. Depending on additional information about the data accessed in the query, this AMAχOS node might decide to evaluate only some parts of the query locally (e.g., those operating exclusively on local data and those joining data from different sources) and send all the remaining query parts to other AMAχOS nodes that are likely to have more efficient access to the relevant data.

In contrast to distribution on the level of a high-level query language such as Xcerpt, distribution on the level of AMAχOS has two main advantages: the distributed query parts can be of finer granularity and

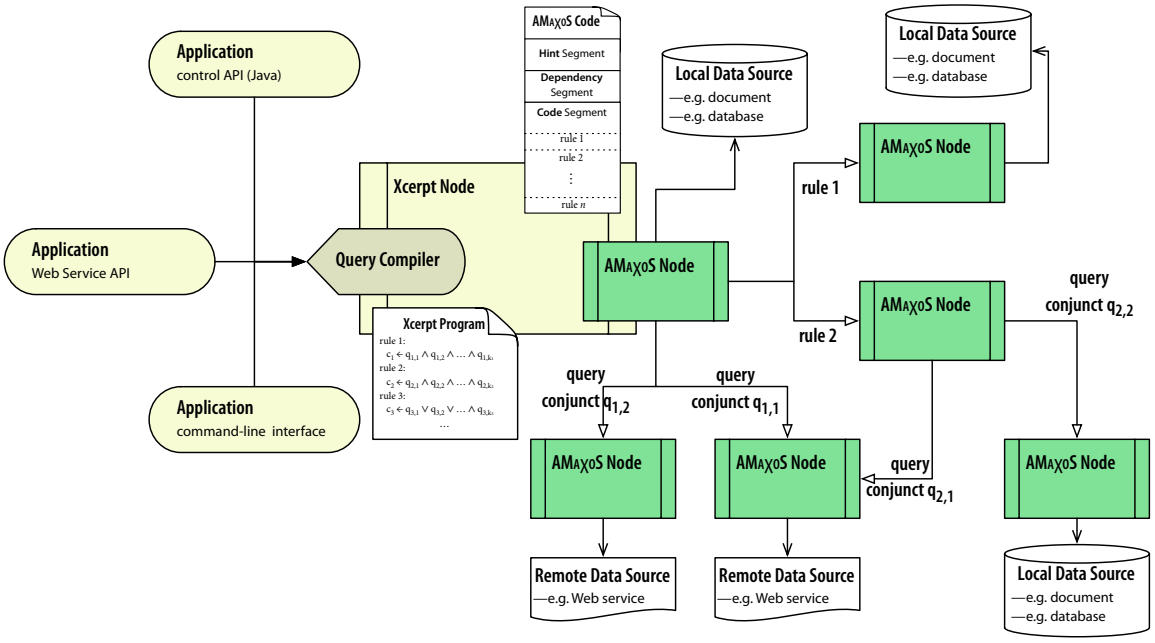

Figure 4.2: Query Node Network

the "controlling" node can have, by means of code transformation and hint sections, better control of the "slave" nodes.

Notice, that AMAχoS enables such query distribution, but does not by itself provide the necessary infrastructure (e.g., for registration and management of query nodes). It is assumed that this infrastructure is provided by outside means.

4.5 Architecture: Overview

The previous section illustrates the guiding principles in the development of AMAχoS. The remainder of this article focuses on how these principles are realized in its architecture and discusses several design choices regarding the architecture.

Notice, that only a small part of the full AMAχoS architecture as described here has been implemented so far. We have concentrated in the implementation on the execution and optimization layer, that are also described in more detail in Sections 4.5.1 and 4.5.2.

Figure 4.3 shows a high-level overview of AMAχoS and its components. The architecture separates the components in three planes:

Control Plane. The control plane enables outside control of the compilation, execution, and answer construction. Furthermore, it is responsible for observation and adaptive feedback during execution.

Program Plane. The program plane contains the core components of the architecture: the compilation and execution layer. It combines all processing that an Xcerpt program partakes when evaluated by an AMAχoS virtual machine. The first step is, naturally, parsing, verification, normalization, module

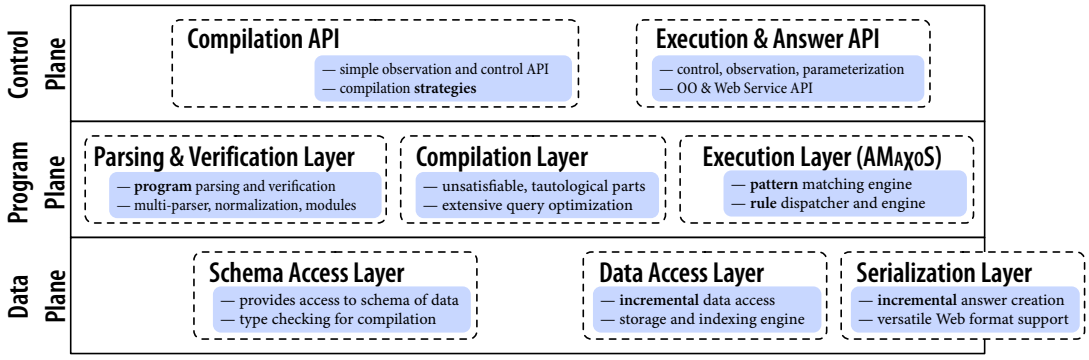

Figure 4.3: Overview of AMAχoS Components

expansion etc. These are realized as transformations on the layer of the Xcerpt language and the resulting normalized, verified, and expanded Xcerpt program can be accessed via the compilation API. However, usually the result becomes input for the compilation layer where the actual transformation into AMAχoS code takes place. The details of this layer are discussed below in Section 4.5.2. In the architecture overview, we chose to draw the compilation and execution layer as directly connected. However, it is also possible to access the resulting program (again via the compilation API) and execute it at a later time and even at a later place. Indeed, compilation and execution are properly separated with only one interface between them: the AMAχoS program containing aside of the expressions realizing individual rules in the Xcerpt program also supporting code segments that provide hints for the program execution and dependency information used in the rule dispatcher, cf. Section 4.5.1.

Data Plane. The architecture is completed by the data plane, wherein all access to data and schema of the data is encapsulated. During compilation, if at all, only the schema of the data is assumed to be available.

It is used for typical schema-based optimization such as the elimination of tautological (always true) query parts, the detection of erroneous (always false) queries, the unfolding of arbitrary length path traversals if the length of the paths is known from the schema and small, etc. Furthermore, it is essential for the dependency analysis later used in the execution layer, that gives information about which conjuncts in rule bodies are compatible with which rule heads. In the data access layer the actual access to queried data takes place at execution time. Where possible, data is accessed incrementally and only those portions of the data are delivered from the data access layer to the execution layer that may actually affect the query outcome (similar to document projection in [52]). The AMAχoS program can contain execution hints that advise the employment or ad-hoc creation of indices, e.g., to accelerate certain often used constructs or sub-queries. Finally the serialization layer is responsible for creating a sequential representation of the result of a query. For XML it follows closely [47], for other Web formats appropriate serialization support is provided as well. Again the form of the serialization can be parameterized both in the AMAχoS code and via the execution control API.

4.5.1 AMAχoS Core

The core of the AMAχoS virtual machine is formed by the query execution layer, or AMAχoS proper. Here, an AMAχoS program (generated separately in the compilation layer, cf. Section 4.5.2) is evaluated against data provided by the runtime data access layer resulting in answers that are serialized by the serialization

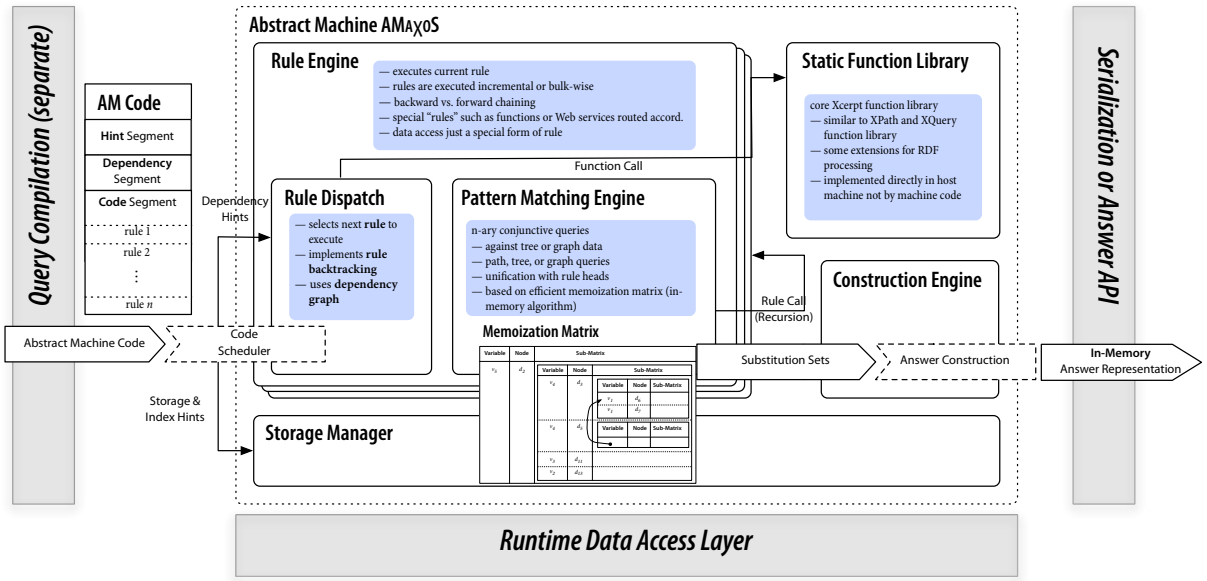

Figure 4.4: Architecture of Core Query Engine AMAχOS

API.

As shown in Figure 4.4, the query execution layer is divided in four main components: the rule engine, the construction engine, the static function library, and the storage manager. Once a program containing AMAχOS code is parsed information from the *hint segment* is used to parameterize storage manager and rule engine. These parameters address, e.g., the classification of the contained queries (tree vs. graph queries), the selection of access paths, filter expressions for document projection, the choice of in-memory representation (e.g., fast traversal vs. small memory footprint), etc. The rule dependency information is provided to the *rule dispatcher* who is responsible for combining the results of different rules and matching query conjuncts with rule heads. Each rule has a separate segment in the AMAχOS program containing code for pattern matching and for result construction. Intermediary result construction is avoided as much as possible, partially by rule unfolding, partially by propagating constraints on variables from rule heads into rule bodies. Only when aggregation or complex grouping expressions are involved, full intermediary construction is performed by the *construction engine*. The rule dispatcher uses the *pattern matching engine* for the actual evaluation of Xcerpt queries compiled into AMAχOS code. The pattern matching engine uses variants of the algorithms described in [18] that are based on the *memoization matrix* for storage and access to intermediary results. The rule engine also detects calls to external functions or Web services and routes such calls to the *static function library*, that provides a similar set of functions as [51] which are implemented directly in the host machine and not as AMAχOS code.

For each goal rule in the AMAχOS programs the resulting substitution sets are handed over to the *construction engine* (possibly incremental) which applies any construction expressions that apply for that goal and itself hands the result over to the serialization layer or to the answer API.

The most notable feature of the AMAχOS query engine is the separation in three core engines: the construction, the pattern matching, and the rule engine. Where the rule engine essentially glues the pattern matching and the construction engine together, these two are both very much separate. Indeed, at

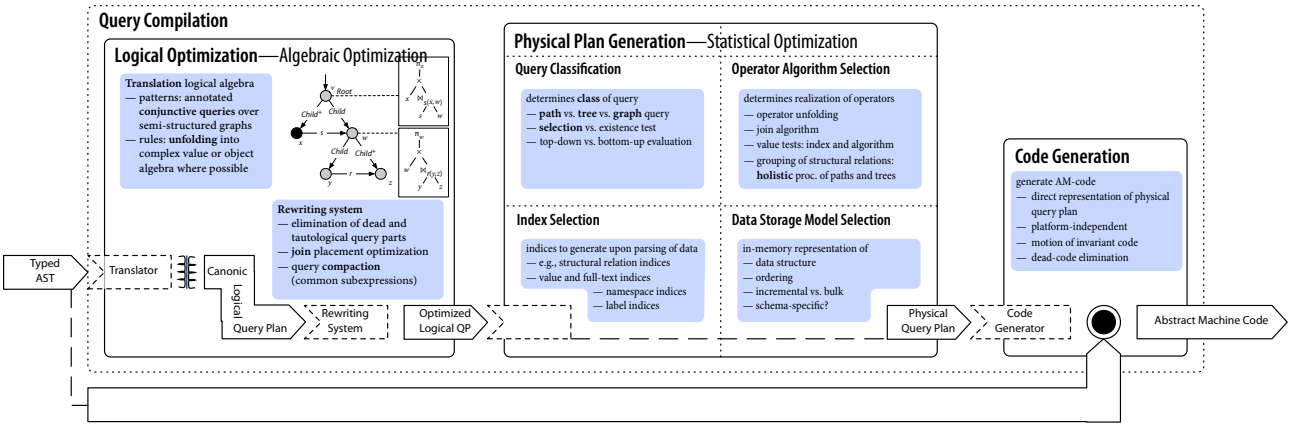

Figure 4.5: Architecture of Query Compiler for AMAχOS

least on the level of AMAχOS code even programs containing only queries (i.e., expressions handled by the pattern matching engine) are allowed and can be executed by this architecture (the rule dispatcher and construction engine, in this case, merely forwarding their input).

4.5.2 Query Compiler

Aside of the execution engine, the query compilation layer deserves a closer look. Here, an Xcerpt program—represented by an abstract-syntax tree annotated with type information—is transformed into AMAχOS code. It is assumed that the Xcerpt program is already verified, normalized, modules are expanded, and type information is added in the prior parsing layer. The query compilation is essentially divided in three steps: logical optimization, physical plan generation, and code generation.

Logical optimization is similar as in traditional database systems but additionally has to consider rules and rule dependencies: Xcerpt programs get translated into a logical algebra based on n-ary conjunctive queries over semi-structured graphs [18]. Expressions in this algebra are then optimized using various rewriting rules, including dead and tautological query part elimination, join placement optimization, and query compaction. Furthermore, where reasonable, rules are unfolded to avoid the construction of intermediary results during execution.

In contrast, *physical plan generation* differs notably, as the role of indices and storage model is inverted: In traditional databases these are given, whereas in the case of AMAχOS the query compiler generates code in the hint section indicating to execution engine and storage manager which storage model and indices (if any) to use. Essential for execution is also the classification of queries based on shape of the query and (static) selectivity estimates. E.g., a query with highly selective leaves but low selectivity in inner nodes is better evaluated in a bottom-up fashion, whereas a query with high selectivity in inner nodes profits most likely from a top-down evaluation strategy. Operator selection is rather basic, except that it is intended to implement also holistic operators for structural relations where entire paths or even sub-trees in the query are considered as parameter for a single holistic operator, cf., e.g., [15, 57].

An AMAχOS program can, in many respects, be considered a serialization of a physical query plan for an Xcerpt program. Notice, however that it provides only local operator sequencing, as rules are kept separate and only at run-time the sequencing of rule applications is performed by the rule dispatcher, cf. Section 4.5.1.

Therefore, the *code generator* is rather simple, performing only basic serialization tasks and simple code optimizations such as motion of invariant code [48].

To conclude, the query compilation layer employs a mixture of traditional database and program compilation techniques to obtain an AMAχOS program from the Xcerpt input that implements the Xcerpt program and is, given the limited knowledge about the actual data instances, likely to perform well during execution. The compilation process is rather involved and expected to be time expensive if all stages are considered. A control API is provided to control the extent of the optimization and guide it, where possible. We belief that in many cases an extensive optimization is called for, as the query program can be reused and, in particular if remote data is accessed, query execution dominates by far query compilation.

4.6 Conclusion and Outlook

We present a brief overview over the principles and architecture of a novel kind of abstract or virtual machine, the AMAχOS virtual machine, designed for the efficient, distributed evaluation of Xcerpt query programs against Web data.

In particular, we show how the Web setting affects traditional assumptions about query compilation and execution and forces a rethinking of the conclusions drawn from these assumptions. The proposed principles and architecture reflect these changing assumptions

1. by emphasizing the *importance of a coherent and clearly specified execution environment* in form of an abstract machine for distributed query evaluation,
2. by *separating query compilation from query execution* (as in general programming language execution),
3. by employing a *unified query evaluation algorithm* for path, tree, and graph queries against tree and graph data, and
4. by emphasizing *adaptive optimization* as a means to ameliorate the loss of quality in predictive optimization due to lack of knowledge about remote or volatile data instances.

Implementation of the proposed architecture is still underway, first results on the implementation of the query engine have been reported in [18] and in [9], demonstrating the promise of the discussed method and architecture.

Chapter 5

Towards Xcerpt 2.0—Functions and Primitive Types

5.1 Introduction

In Xcerpt 1.0, we provide only a very limited set of predefined functions and primitive types. For Xcerpt 2.0, we have developed a flexible framework for adding new (primitive) types and their functions. This framework is, as a prototype, implemented in Java and provides means for adding new types, adding functions, converting between types etc. without need for recompiling the Xcerpt prototype. Details on the use of the framework are presented in Section 5.2.5. A basic set of types and functions for numerical types is discussed as an example in Section 5.3. We are currently working on extending this basic set and integrating the implementation into the Xcerpt 2.0 prototype.

5.2 Implementation

5.2.1 General Information

The prototype implementation of our types and functions framework is written in Java. It consists of three classes, namely `XcerptType`, `XcerptFunction`, and `XcerptValue` which all depend on each others functionality in order to create a complete type system and set of functions on the defined types.

Both `XcerptType` and `XcerptFunction` are abstract classes and serve to represent tangible objects as well as to organise those objects by using static attributes and methods. For these organisational features a registration mechanism is used. This means that in order to create "usable" instances of those classes, instances have to be registered after construction. `XcerptValue` is a simple container class and has no such limitations.

5.2.2 The `XcerptType` class

As its name suggests the `XcerptType` class is responsible for single type instances and all organisational information about those (e.g. supertypes, disjoint types, conversions). For each type there exists a unique instance of `XcerptType` which after registration holds all immediately characteristic information. This type information consists of three attributes: a unique identifier, super-types and disjointness with other

types. While the former two are an essential part of registration, the latter has to be set explicitly. If not set (by the `setDisjoint` method) it defaults to no disjointness with other types at all. After registration this information is immutable but can be read by external classes.

The unique identifier is required by the framework as types (and functions as well) have to be accessible globally from anonymous inner classes. The other information is necessary for implicit type conversion and several additional attributes are filled with implicit type conversion information as soon as specific methods are called. `conversions` contains all direct conversion possibilities obtained by calls of the `addConversion` method during function registrations (more on that in the following section). `transitiveConversions` contains all transitive conversion possibilities (ordered by path length) obtained by calculating the transitive hull of the relation induced by the `conversions` attribute. This calculation is performed as part of the type system finalization (triggered by the `finish` method) in order to guarantee completeness. Each instance of `XcerptType` has a `convert` method which with the help of the gathered information allows (to try) to convert a given value from the type it is called on to a value of a given (usually an other) given type.

The global type system information is stored in static attributes that can be accessed via getter methods. Those attributes are modified every time a new type is registered.

5.2.3 The `XcerptFunction` class

The `XcerptFunction` class is quite similar to the `XcerptType` class in turns of organisation. Instances of this class represent unique functions on priorly registered types. In order to define a function a subclass of `XcerptFunction` has to be instanciated, which is done most conveniently by taking advantage of anonymous classes. This subclass has to implement the abstract `apply` method which should contain the actual function implementation.

A function here is characterised by its identifier, its expected parameter types and its result. The identifier is not unique but the signature (identifier and parameter types) is. This information has to be supplied upon registration and cannot be modified. Integrated in the registration process is the identification of conversion functions by a naming convention. In our framework a function is a conversion function if and only if

- its identifier is equal to the identifier of a type A,
- its result type is A and
- it only takes one parameter.

Also there may not exist a function which fullfills the first two of the above requirements but violates the last. So for types A and B a conversion function always has the following form.

$$A : B \to A$$

After a function has been identified as a conversion function it is added to the function's parameter type as a conversion to the function's result type by using the `addConversion` method. In our example from above we would call it on the type B.

Now in order to retrieve instances of `XcerptFunction` globally, we offer two methods, namely `getAll`, which returns all functions which share a given identifier, and `get`, which returns a single function by supplying an exact signature. Global information (i.e. the list of functions) is handled the same way as global type system information is in the `XcerptType` class.

5.2.4 The `XcerptValue` class

The `XcerptValue` class serves no other purpose than representing an actual value in our type and function framework by supplying a Java object with type (as instance of `XcerptType`) information. Hence its attributes are `type` and `boxedValue`, the latter being a Java `Object`, which are both immutably set upon construction and can be accesed by getter methods.

For convenience sake there exists an additional constructor, which doesn't take an actual instance of `XcerptType` but a type identifier as one of its parameters, and a `convert` method, which simply forwards calls to the `convert` method of the `XcerptType` class.

5.2.5 Usage

For a programmer who wants to extend Xcerpt with new types and functions it is important to know how to interact with the provided interfaces. Hence this section contains a usage guide for our implementation briefly introduced above.

There are three basic elements: types, functions, and values. Types are used to define a set of values. Each type has either one ore no direct supertype. If a type does not have a direct supertype it is considered the default type and no other type in the type system must have no direct supertype. Functions may have zero or more parameters and a single result. Overloading is possible so that a function is uniquely identifiable by its signature which consists of its identifier and its parameter types. Values are used for function parameters and results. They always consist of a type and a boxed value (i.e. a Java `Object`) which can only be read.

Before a type system for Xcerpt can be built with the here provided framework, the following requirements and recommendations have to be considered:

1. Every type system needs a default type, which is a global supertype for all types in the type system. This means that there has to exist a transitive conversibility relationship (regarding implicit conversion) for every other type in the system.

2. Implicit conversion should always maintain information (i.e. implicit conversion may fail but not both succeed and lose information, at least not within a big tolerance) and conversions to a (transitive) supertype should never fail.

3. Static type checking will only be possible if information about type disjointness is given, dynamic type checking is always possible (e.g. via trial-and-error conversion).

What follows is a detailed explanation of the steps that have to be performed in order to create types and functions.

5.2.5.1 Adding types

To add a type to the type system an instance of `XcerptType` has to be created and registered afterwards. The first type to be added to the type system is the default type. Supposing this type should actually be named `default`, the following code snippet shows how to add it.

Listing 5.1: Adding the default type

```
XcerptType defaultType = new XcerptType() {};
defaultType.register("default");
```

[2]

By not providing any parameters but its identifier when the `register` method is called `defaultType` becomes the default type. The existence of additional parameters would mean that *default* has supertypes and thus couldn't be a default type. Together with the forced existence of at least one supertype for every non-default type this means, that in every type system, the first succesfully registerable type is the default type.

Something not yet discussed is the fact, that `XcerptType` is an abstract class (thus the curly braces). By using an abstract class we allow abstract customization method to be added later as the framework evolves (e.g. a method to parse literals of the specified type) like the `apply` method is used in the `XcerptType` class.

Usually a type system does not consist of only a single type but adding additional types is only slightly more complicated than adding the default type. The first new thing that has to be taken into consideration are supertypes. In the following example types *a*, *b* and *c* are added.

Listing 5.2: Adding additional types

```
XcerptType aType = new XcerptType() {};
aType.register("a", defaultType);
XcerptType bType = new XcerptType() {};
bType.register("b", defaultType);
XcerptType cType = new XcerptType() {};
cType.register("c", bType);
```

Here *a* and *b* are both direct subtypes of the default type, *c* is a subtype of *b* and thus an indirect subtype of the default type. Several supertypes can be given after the identifier as variable parameter length is used here.

We still do not now if any of those types are definitely not convertible into one another. So in order to perform static type checking additional information about the relation between types has to be provided. The following short example shows how to tell the type system which types are disjoint.

Listing 5.3: Setting disjointness

```
aType.setDisjointTypes(b);
```

Now *a* and *b* are considered disjoint, *c* is a subtype of *b* and "inherits" all properties form its supertype. Thus *a* and *c* are also disjoint. Like the `register` method does, the `setDisjointTypes` method can accept more than one parameter in order to set disjointness for multiple types at once.

5.2.5.2 Adding functions

In the steps above the basic type system structure has been defined. To complete this structure with information about general convertibility we first need to implement and register conversion functions. For each pair of subtype and its supertype there has to exist a registered conversion function from subtype to supertype and a registered conversion function from supertype to subtype. Concerning implementation and registration conversion functions are treated the same as normal functions and can only be distinguished if they adhere to the naming convention already described earlier.

Listing 5.4: Adding a conversion function

```
new XcerptFunction() {
```

```
      public XcerptValue apply(XcerptValue... parameters) {
            // Insert implementation here
      }
}.register("default", defaultType, bType);
```

Except for the fact that there is no real implementation, which would depend on the values of b and $default$ and is not discussed in detail here, the example shows how to create and register a conversion function from b to $default$. Of course in order to have a valid type system we would need more conversion functions as described above (in the requirements). Besides the required conversion functions we can register additional conversions for example from $default$ to c. These would be taken into consideration when transitive conversions are calculated.

The procedure to add "normal", non-conversion functions only differs in naming the function (and of course in implementation).

Listing 5.5: Adding a "normal" function

```
new XcerptFunction() {
      public XcerptValue apply(XcerptValue... parameters) {
            // Insert implementation here
      }
}.register("doSomething", aType, aType, cType);
```

Here registration also makes use of variable parameter length. First the identifier, then the result type and after that an arbitrary amount of parameter types. The example results in the following function.

$$doSomething : a \times c \rightarrow a$$

5.2.5.3 Finalization and resetting

After everything has been set up the type and function definitions have to be finalized. This is only two lines of code but itis necessary for infering conversions from the registered functions etc.

Listing 5.6: Finalization

```
XcerptFunction.finish();
XcerptType.finish();
```

Note: This would actually fail because of some missing conversion functions. But assuming we added those earlier finalization will suceed.

Although it probably will not occur very often, there still could be a reason for starting over again. For this case everything can be reset, which is as short as finalization.

Listing 5.7: Resetting

```
XcerptFunction.reset();
XcerptType.reset();
```

In Figure 3 the example type system created (or at least intended to be created) here is visualized. The Venn diagramm on the left side shows the types and their relationships to each other defined by registration and disjointness settings, the hierarchy in the middle matches the type system defined in this

section and the hierarchy on the right side shows the same type system minimally enhanced by conversion functions in order to make it finalizable.

5.3 A pragmatic set of base types and functions

To show that our type and function framework can actually be used to extend Xcerpt reasonably, a pragmatic set of base types and functions will be presented in this section. Of course a complete and maybe complex set as well as detailled function implementations are not a part of this documentation. Instead the focus of this document is to explain ideas and this section in particular only contains a basic prototypical type system with basic functions.

5.3.1 Base type system

Figure 4 shows our base type hierarchy, where the part enclosed in parantheses has not been implemented yet and is to be seen as a motivational idea for further extension of this type system. They are a direct copy off of a branch of the hierarchy presented in the W3C Recommendation "XML Schema Part 2: Datatypes Second Edition" (http://www.w3.org/TR/xmlschema-2/)

So in fact there are only five types present at the moment:

- *string*, simple strings (e.g. `Hello world!`, `Xcerpt`, `document`)
- *rational*, rational numbers (e.g. `3.6`, `2`, `-127.2`)
- *integer*, integer numbers (e.g. `4`, `-10`)
- *boolean*, boolean values (`true`, `false`)
- *IRI*, Internationalized Resource Identifiers (e.g. `http:\\127.0.0.1\index.htm`)

Still those are enough types for defining interesting functions that demonstrate our approach. At first it might appear strange to consider *rational* a subtype of *string* and *integer* a subtype of *rational*, but the former relationship can be explained as input from an XML file is, if not otherwise defined (for example with XMLSchema), text at first.

The following Venn diagram visualization is not surprising as in this type system there are hardly any non-empty type value intersections (maybe with the exception of *IRI* and *token*, but the latter is not really taken into consideration now anyway).

5.3.2 Conversion functions

string	:	*rational*	→ *string*	conversion from *rational* to *string*
string	:	*boolean*	→ *string*	conversion from *boolean* to *string*
string	:	*IRI*	→ *string*	conversion from *IRI* to *string*
rational	:	*string*	→ *rational*	conversion from *string* to *rational*
rational	:	*integer*	→ *rational*	conversion from *integer* to *rational*
integer	:	*rational*	→ *integer*	conversion from *rational* to *integer*
boolean	:	*string*	→ *boolean*	conversion from *string* to *boolean*
IRI	:	*string*	→ *IRI*	conversion from *string* to *IRI*

5.3.3 Base functions

5.3.3.1 *string* functions

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

5.3.3.2 *rational* functions

abs	:	*rational*	→ *rational*	calculates the absolute value of a given number
sign	:	*rational*	→ *rational*	calculates the sign of a given number
inv	:	*rational*	→ *rational*	calculates the (additive) inverse of a given number
add	:	*rational* × *rational*	→ *rational*	addition of two numbers
sub	:	*rational* × *rational*	→ *rational*	subtraction of two numbers
mul	:	*rational* × *rational*	→ *rational*	multiplication of two numbers
div	:	*rational* × *rational*	→ *rational*	(rational) divison of two numbers
pow	:	*rational* × *rational*	→ *rational*	exponentiation of number
sqrt	:	*rational*	→ *rational*	calculates the square root of a number
round	:	*rational*	→ *integer*	rounds a number (explicit, lossful conversion to *integer*)

5.3.3.3 *integer* functions

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

5.3.3.4 *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

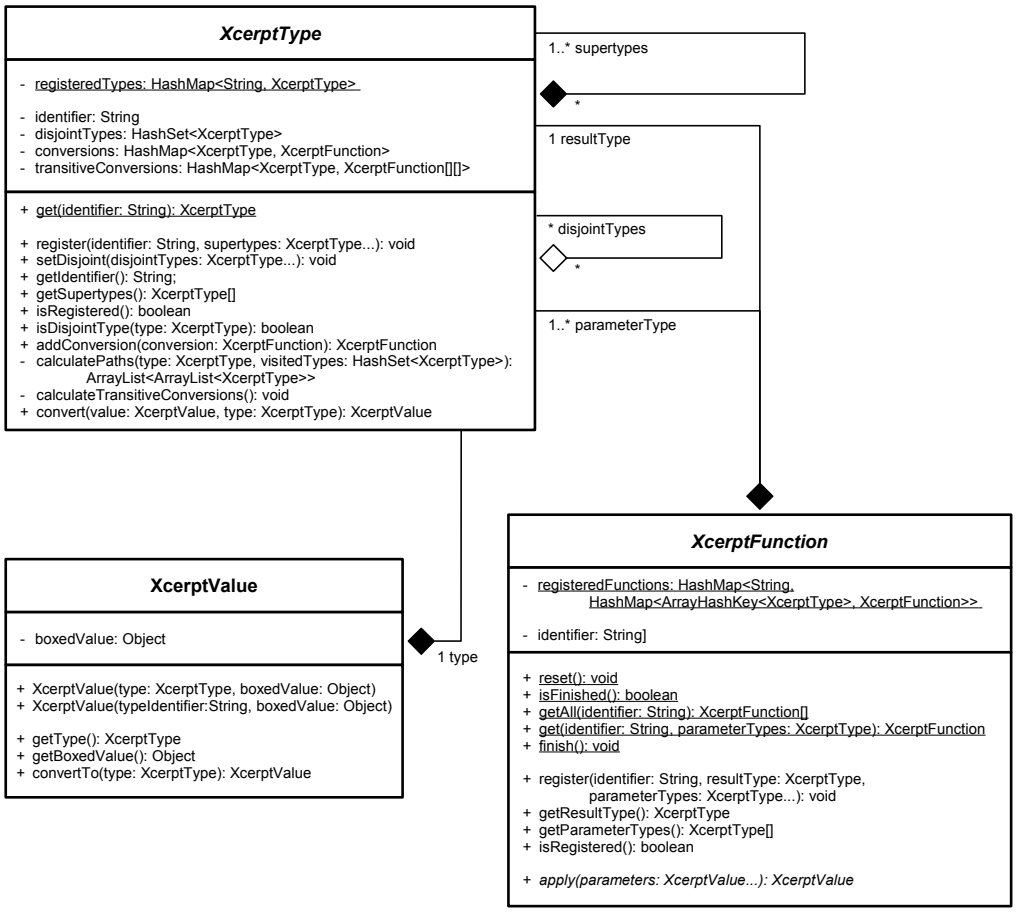

Figure 5.1: UML diagramm

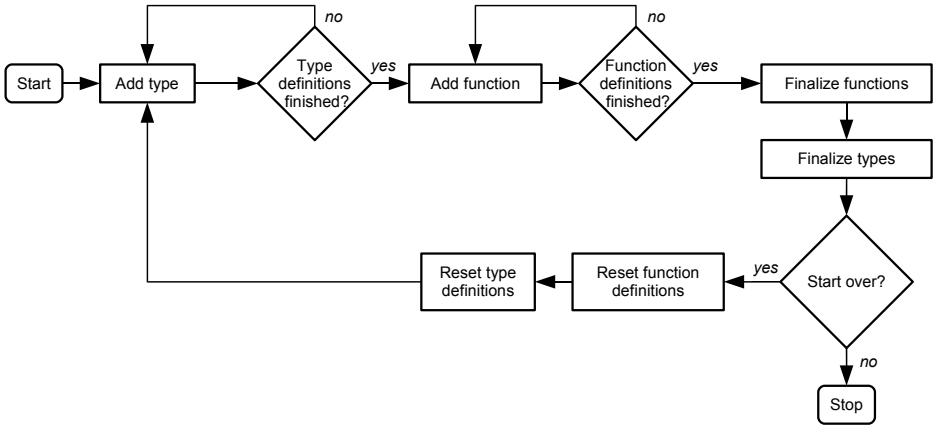

Figure 5.2: Usage flowchart

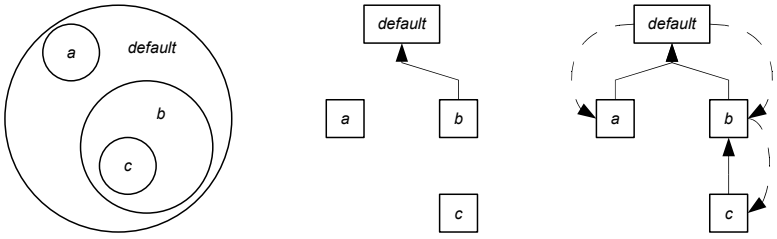

Figure 5.3: Example type system visualization

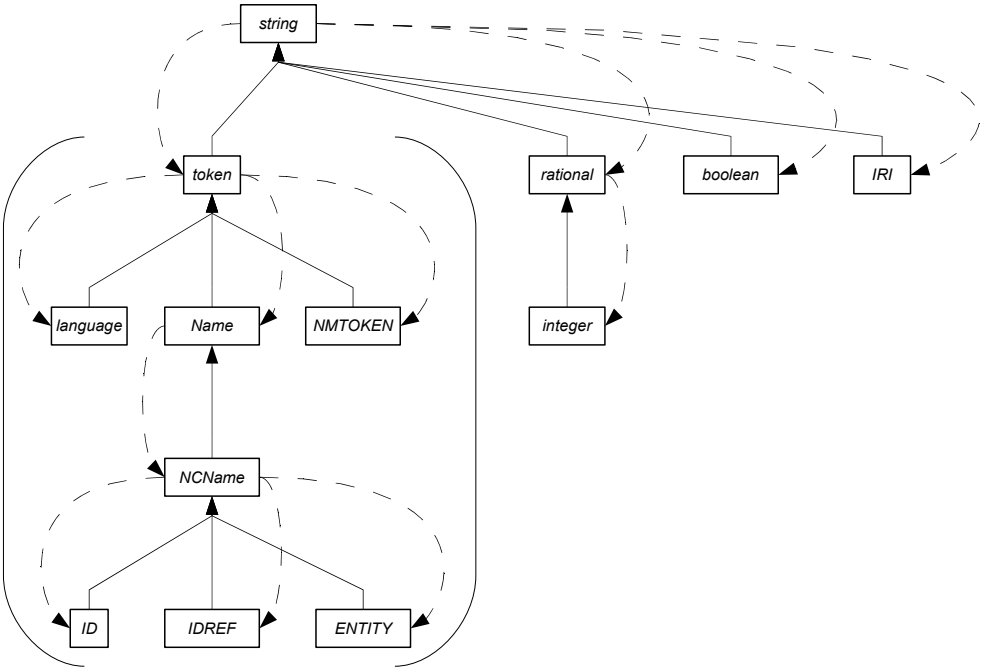

Figure 5.4: Base type system hierarchy

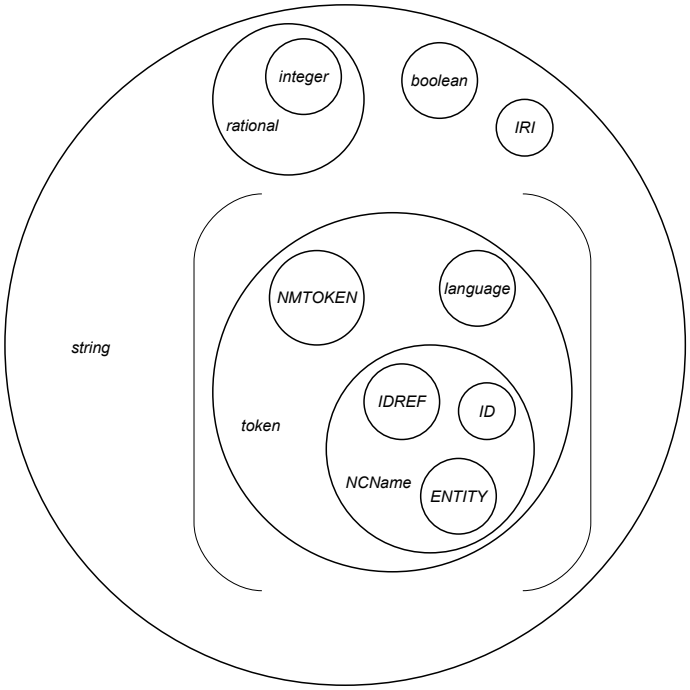

Figure 5.5: Base type system Venn diagramm

Chapter 6

Towards Xcerpt 2.0—Enabling Component-Reuse from Rules to Stores

6.1 Introduction

As the amount and diversity of data available on the Web is constantly increasing, querying this great abundance of information is becoming more and more important. In fact, it is becoming less important to possess certain knowledge, but more important to know how to acquire it—know how to formulate a precise *query* to find the desired information. Query languages for different purposes are emerging in multitude. [8] surveys some existing query and transformation languages for Web and Semantic Web data, identifying 14 textual XML query languages and 24 for RDF metadata.

Yet, most of these languages provide very little support to the user to cope with the dramatic increase in information size and diversity. Increasing information diversity results in increase of query size and complexity, which can weigh down even experienced query programmers. It must be easy for users to partition (both conceptually and from an evaluation point of view) query programs and to make such partitioning flexible enough to allow for reuse in different contexts. This is not the case unless the query language provides some means to separate large and complex queries into smaller, properly isolated, and reusable fragments—*modules*. Such modules allow to "localize" the effect of the introduction of additional data sources or query tasks in query programs.

Thus, modules allow a *separation of concern* not just on the basis of single rules but on the basis of larger conceptual units of a query program. For example, one part of a Web application is often concerned with extracting data from a set of sources, such as a set of Web pages. At the next step, the data might have to be syndicated into a common view and format. From this syndicated data, some new implicit data could possibly be derived. Finally, the resulting data set should be displayed in an appropriate human-readable form, for example, by being displayed in a well-structured Web page (see Section 6.3 for an example). These different steps taken by the application have to do with different concerns of the overall realization, such as data extraction, data management and data display. Furthermore, each of the concerns deals with different schemata, but the knowledge of the schemata can be hidden and encapsulated within each concern – within each module. In contrast, exposing all these concerns in one monolithic query program not only becomes very hard to understand, but is also impossible to manage as a change in some part may affect any other part.

This work is based on ideas from [41, 4] where we propose a flexible approach for augmenting arbitrary

languages with new levels of *abstractions*, and new constructs for authoring reusable entities. The only requirement we put on the newly introduced constructs is that their realization is already expressible in the original language, i.e., that they have a reduction semantics. In doing this we take advantage of existing software composition techniques[1] to realize the added reuse abstractions [41]. However, in this paper we do not focus on the details of composition systems, but show an application of the ideas to a concrete query language, viz. Xcerpt [68].

For that language, we propose a module system that (a) demonstrates how Web query languages can profit from modules by partitioning the query program as well as its execution; (b) provides an easy, yet powerful module extension for Xcerpt that shows how well-suited rule-based languages are for component-based reuse; (c) is based on a single new concept, viz. "stores"; and (d) uses a reduction semantics exploiting the power of a language with views. This semantics enables the reuse of the existing query engine making the design of the module system easier and its deployment less time consuming.

The rest of this paper is organized around these contributions: Following a brief introduction to Xcerpt we demonstrate the need for modules or similar reuse and partitioning mechanisms by a use case on integrating (Semantic and plain old) music data on the Web. Then we introduce the module extension for Xcerpt by implementing part of the aforementioned use case. We conclude with a discussion of the semantics and realization of the module extension.

6.2 Introducing Xcerpt

We choose to demonstrate our ideas using the rule-based, Web and Semantic Web query language Xcerpt [68], which has been co-developed by some of the authors and is particularly well-suited for reuse due to its rule-based nature. This chapter is not intended as a full introduction to Xcerpt but merely recalls some of its most relevant features for this article. For a proper introduction please see [68].

An Xcerpt program consists of a finite set of Xcerpt *rules*. The rules of a program are used to derive new, or transform existing, XML data from existing data (i.e. the data being queried). *Construct rules* are used to produce intermediate results while *goal rules* form the output of programs.

While Xcerpt works directly on XML or RDF data, it has its own data format for modeling XML documents or RDF graphs, viz. Xcerpt *data terms*. For example, the XML snippet `<book><title>White Mughals</title></book>` corresponds to the data term `book [ title [ "White Mughals" ] ]`. The data term syntax makes it easy to reference XML document structures in queries and extends XML slightly, most notably by also allowing unordered data.

For instance, in the following query the construct rule defines data about books and their authors which is then queried by the goal. Intuitively, the rules can be read as deductive rules (like in, say, Datalog): if the body (after **FROM**) holds, then the head (following **CONSTRUCT** or **GOAL**) holds. A rule with an empty body is interpreted as a fact, i.e., the head always holds.

```
GOAL
  authors [ var X ]
FROM
  book [[ author [ var X ] ]]
END

CONSTRUCT book [ title [ "White Mughals" ], author [ "William Dalrymple" ] ] END
```

[1] Developed within the Reuseware Composition Framework (`http://reuseware.org`).

Xcerpt *query terms* are used for querying data terms and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique[2] to match data terms. Query terms can be configured to take partiality and/or ordering of the underlying data terms into account during matching (indicated by different types of brackets).

Query terms may also contain logic variables. If so, successful matching with data terms results in variable bindings used by Xcerpt rules for deriving new data terms. Matching, for instance, the query term `book [ title [ var X ] ]` with the XML snippet above results in the variable binding `{X / "White Mughals" }`.

Construct terms are essentially data terms with variables. The variable binding produced via query terms in the body of a rule can be applied to the construct term in the head of the rule in order to derive new data terms. For the example above we obtain the data term `authors [ "William Dalrymple" ]` as result.

6.3 Use case: Music aggregation with the Web Music Library

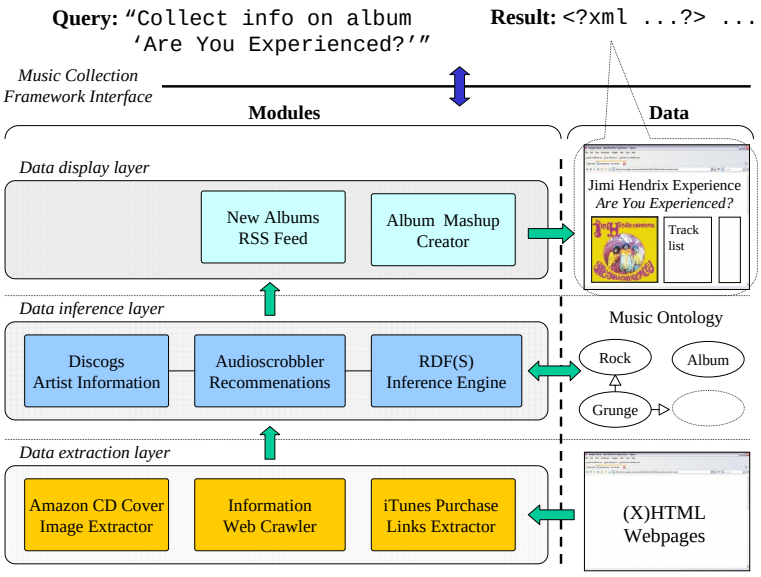

Figure 6.1: Many query languages only allow writing monolithic queries, while modular query development greatly increases reuse and ease of programming.

The use case illustrated in Figure 6.1 presents a library (called MusicLibrary) of functionality useful for coping with music and information about music found on the (Semantic) Web. At an (arguably) lower layer, information is extracted from various established Web sites like `amazon.com` or `discogs.org`. The extraction has to be handled differently for every web site, but is valuable for many users and applications. For example, many of the currently established desktop music players exploit the album or CD images of Amazon to display cover art while playing back music. Encapsulating reusable queries dealing with a particular information source allow for flexible maintenance and propagation to a larger user base. The legacy information as found on external Web sites is then converted to an internal representation

[2] Called *simulation unification*. For details of this technique, please refer to [67].

loosely based on the Music Ontology [36]. Music Ontology is an RDFS-based standard, hence knowledge inference and reasoning on—possibly incomplete—Music Ontology data can be achieved using an RDFS reasoner. Since such a reasoner is usable in many different fields of applications, it is implemented and provided as an Xcerpt module and included in the main library, hence allowing for its reuse. Perhaps more interesting to the end user, various modules providing pleasant visualizations of gathered information or predefined query skeletons can be provided in the library. Such modules can also be provided by third parties or, last but not least, as part of an application using the Web Music Library.

6.3.1 Realizing Musical Modules in Xcerpt

How can we today realize this application in Xcerpt? In the absence of modules we have to carefully craft a single query program with a considerable number of rules (well over three dozens if we follow the basic design presented below) at each step taking great care that the rules do not, by chance, interfere with each other. Furthermore, we have to update the whole query program as soon as any information source changes, since this information is hard-coded in the program.

In the presence of a module extension, the task becomes a lot less daunting: Let us start from the top with a user program that gathers information about Jimi Hendrix from all the sources described in Figure 6.1. For that, it relies on a module called MusicLibrary (discussed above). The library is not a mere database, it is an interface to various ways of reasoning about musical information available on the Web. To the user the complexity remains hidden. The user just poses his query to the module without caring whether the data is extensional or intensional and how it is obtained. The module system ensures that, regardless of the actual rules and their distribution between modules, there is no chance for interference by rules of different sub-modules used within MusicLibrary.

```
IMPORT "MusicLibrary"

GOAL
 html [ body [
  h1 [ "Records by Jimi Hendrix" ],
  table [ tr [ td [ "Record" ], td [ "Year" ] ],
         all tr [ td [ var R ], td [ var Y ] ] ]
 ] ]
FROM
 in "MusicLibrary" (
  desc record { artist [ "Jimi Hendrix" ],
                title [ var R ], year[ var Y ] } )
END
```

The MusicLibrary module itself is integrating data and knowledge of other modules the same way as the user program. It has to provide the information, and only the desired information, to the user of the module. Some rules may be necessary internally in the module to achieve the task, but should not be directly visible to the user of the module. The visible parts of the module are hence *public*, the others (implicitly) *private*.

Apart from using knowledge of other modules, modules may also receive data provided by importing modules. MusicLibrary accesses data extracted by a module gathering MusicBrainz metadata, feeds it to a module for converting that data to Music Ontology knowledge (Musicbrainz2MOFacts), and finally injects that knowledge to an RDFS reasoner (using the MO-Ontology-Reasoner module). It also accesses `discogs.org` directly and feeds the acquired data into another instance of the MO-Ontology-Reasoner. To distinguish

multiple instances of the reasoner, each instance is given an alias (using the `@` construct), which can be used the same way as the module identifier when querying, or sending data to, a module. In this way, modules also give rise to *scoped* reasoning where consequences only apply in a certain scope (or module), but are not (automatically) propagated outside of that scope. In particular, knowledge in different scopes may, if considered globally, be inconsistent, but within each scope be consistent.

```
MODULE "MusicLibrary"
IMPORT "MusicBrainz"
IMPORT "Musicbrainz2MOFacts"
IMPORT "MO-Ontology-Reasoner" @ "reasoner-for-musicBrains"
IMPORT "MO-Ontology-Reasoner" @ "reasoner-for-discogs"

CONSTRUCT public var KNOWLEDGE
FROM in "reasoner-for-musicBrains" ( var KNOWLEDGE ) END

CONSTRUCT public var KNOWLEDGE
FROM in "reasoner-for-discogs" ( var KNOWLEDGE ) END

CONSTRUCT to "reasoner-for-musicBrains" ( var FACTS )
FROM in "Musicbrainz2MOFacts" ( var FACTS ) END

CONSTRUCT to "Musicbrainz2MOFacts" ( var METADATA )
FROM in "MusicBrainz"( metadata [[ var METADATA ]] ) END
...
CONSTRUCT discogs-document-for-crawler[ all HREF ]
FROM in document(iri="http://www.discogs.org") ( desc a [[ href [ var HREF ] ]] ) END
```

Finally, let us glance at the MO-Ontology-Reasoner module which is one of the modules that not only extracts data but is injected with data to operate on (here: reason on). Hence, one of the queries is adorned with the **public** keyword, indicating that chaining is to be performed against the rules of the importing module that pass input data to the reasoner. Those facts, together with the ontology definition (and any domain dependent reasoning we would like to perform on the music ontology data) are sent to an RDFS reasoner module, whose consequences are then made publicly available.

```
MODULE "MO-Ontology-Reasoner"
IMPORT "RDFS-Reasoner"

CONSTRUCT public var KNOWLEDGE
FROM in "RDFS-Reasoner" ( var KNOWLEDGE ) END

CONSTRUCT to "RDFS-Reasoner" ( var FACTS )
FROM public var FACTS END

CONSTRUCT to "RDFS-Reasoner" ( var MO )
FROM in document(type="xmlrdf" iri="http://purl.org/ontology/mo/") ( var MO ) END
```

6.4 Modular Xcerpt—Requirements and Constructs

We have seen that modules can greatly ease the development of complex Web queries (as observed increasingly) and how to apply them in examples. Before we discuss the principles of the semantics in Section 6.5, let us first summarize the core concepts and constructs introduced. We divide the presentation of the concepts in two parts: from the perspective of the module programmer and of the module user.

Module programmers need constructs for defining sets of rules and ways of declaring appropriate access to the module—interfaces for proper encapsulation. To allow module authors to encapsulate modules, *visibility constructs* are employed. For each rule of the module, the construct term and the query term (if present) is associated with a visibility concept: *public* or *private*. Only public visibility is specifically specified, otherwise the default visibility *private* is used to encourage encapsulation.

Module declaration: We can group sets of rules into modules and give such a set an identifier. This module can than be imported into other modules or programs.

⟨*module*⟩ ::= 'MODULE' ⟨*module-id*⟩ ⟨*import*⟩* ⟨*rules*⟩*

Module interfaces: We can declare allowed access points to a module to facilitate encapsulation and proper interfaces. Any construct term can be annotated with **`public`** to indicate that it can be queried by importing modules (see below).

⟨*interface-out*⟩ ::= 'public' ⟨*construct-term*⟩

Conversely, importing modules may provision data to an imported module (see 'module provision' below). This data is exclusively queried by query terms marked with **`public`** in the imported module.

⟨*interface-in*⟩ ::= 'public' ⟨*query-term*⟩

In other words, a module programmer defines the name and the in- and output interfaces of a module. The input of a module is accessed or queried by public query terms, the output of a module is formed by public construct terms. A module should also be complemented by documentation for the user describing its task and interfaces.

Module users need to be able to (a) declare which modules they want to use in a program, to (b) query the public interfaces of such modules, and to (c) provide data to such modules.

Module importation: We can import modules into other modules or programs. The only effect of a module is that the module identifier (or its alias, if an alias is used) becomes available for use in module querying or provision statements. In practice, module identifiers are often rather long and complex URIs which makes the use of (short and easy to read) aliases advisable in most cases.

⟨*import*⟩ ::= 'IMPORT' ⟨*module-id*⟩ ('@' ⟨*alias-id*⟩)?

Module querying: We can query the consequences of the public construct terms of a module. The given query term is matched only against the results from *public* rules of the given module but neither against those from that module's *private* rules nor against other rules from the current module.

⟨*module-access*⟩ ::= 'in' ⟨*module-id*⟩ '(' ⟨*query-term*⟩ ')'

Module provision: We can feed or provision data to the public query terms of a module. The result of a rule with such a construct term is only considered for *public* query terms in the given module, not for query terms in the current module or for query terms from the given module that are not marked *public*.

⟨*module-provision*⟩ ::= ‘`to`’ ⟨*module-id*⟩ ‘`(`’ ⟨*construct-term*⟩ ‘`)`’

With only these three operations, a module user can flexibly compose modules (even multiple instances of the same module) while all the encapsulation is taken care of by the module system without further user intervention.

So far, all module access is always explicitly scoped with the module identifier. In a language with views such as Xcerpt, this suffices as we always can add a bridging rule (such as the first rule in the MusicLibrary module from Section 6.3) that makes all data obtained from the public interface of an imported module available to other rules in the importing module (without need for qualification). We provide two additional variants of module import for convenience that cover this case. They only differ in the way they affect module cascading: ‘`import public`’ ⟨*module-id*⟩ makes all data provided by the public interface of module *module-id* available to all unqualified rules in the importing module and also adds it to the public interface of that module whereas ‘`import private`’ ⟨*module-id*⟩ only makes it available to the unqualified rules.

6.5 Reducing Xcerpt Modules—Stores

The dual objectives of our approach are to (a) keep the module system simple and easy to use and to (b) allow the reuse of existing language tools and engines without modification. These two objectives actually go hand in hand, as a reduction semantics for modules (i.e., a semantics that is based on the semantics of the module-free language) proves to be elegant and easy to understand and naturally fulfills the second objective.

To allow users to truly think in terms of modules and make use of this abstraction, it is important to ensure proper and valid module interactivity *statically* before applying the module-unaware query engine to the involved rules. Thus, only the intended rule dependencies must be present in the merged rules—we have no way of enforcing rule separations during rule execution.

For the Xcerpt module system we ensure proper rule dependencies using the notion of **stores**. Intuitively, a store is a designated data area where data and queries are appropriately redirected to adhere to the proper access of rules as specified by the module programmer. A store is associated with an identifier and consists of a *private*, *in* and *out* part. Intuitively, the *private* part is intended for data access internal to the module only and the *in* and *out* parts for input and output data of that module. That is, data to be processed by the module will be injected into the *in* part of the store and data constructed by the module—upon request from another module—will reside in the *out* part of the store and can be queried by an importing module.

Stores can already be simulated using the existing Xcerpt mechanisms. Let us first assume that for each module we have one associated store that is identified by the same (unique) identifier. The construct terms and query terms of each rule in an imported module as well as rules using **`in`** or **`to`** for module access or provision in an importing module are modified such that the appropriate store is referenced:

```
in <module-id> ( <query>→)  store [ id [ <module-id> ], access [ "out"], <query> ]
to <module-id> ( <construct>)store [ id [ <module-id> ], access [ "in"], <construct> ]
CONSTRUCT <c> FROM <q> END  CONSTRUCT store [ id [<module-id>], access["private"], <c>]
                            FROM store [ id [<module-id>], access["private"], <q> ] END
```

Some rules in the imported module are exempted from this transformation, viz. construct terms in goals (producing results for the end user), query terms specifically referencing an external resource (such as an XML document or other module) rather than the internal module store. Also, if the query term

is a complex query it might be necessary to propagate the store specification inside the query (e.g., over disjunctions, negation, etc.). However, these details are omitted here for space reasons.[3]

6.5.1 Refining *Stores:* Instance Stores

The store concept described above ensures basic encapsulation capabilities for Xcerpt modules and is attractive for its simplicity. However, there are certain situations where associating one store per module is not sufficient. Consider the situation where two modules (`A`,`B`) imports a third one (`C`) and both `A` and `B` injects data into the store associated with `C`. In such a case, after module `C` has processed the data, module `A` *may* receive data initially injected by module `B`. As such, modules `A` and `B` are not kept separate violating one of the core premises of our desire for modules. This is not a limit of the store approach, but due to the assumption of the existence of one store per module.

To address this problem, we associate stores not with a module but with a module *import*. This can be seen as instantiating a store for each module import with the identifier of the importing module. We thus end up with two stores `C<A>` and `C<B>`, due to two import operators. A similar case where this is needed is when we use the same module but with different "feeds" using aliases. This is the case in the Music Library module presented in Section 6.3 where aliases (using `@`) were used to force such separations.

Implementation.

Not only is it an advantage to reuse the query engine in executing the transformed and merged rules, it is also beneficial if existing technology can be used to realize the above-described transformations. To achieve this, we realize the module system via composition in the Reuseware Composition Framework [41]. The composition framework allows for the development of a light-weight composition system responsible for handling the augmented constructs related to modules. The composition framework allows both to extend the Xcerpt language with the additional syntactic constructs and to handle the transformation and merging of the involved rules in the manner described above to enforce encapsulation. The details of this implementation are left out for space reasons, but are available at `http://www.reuseware.org/modularxcerptexample`.

6.6 Related work

Practical Web *query* languages need to provide support for some form of reuse and modules as evidenced by (though somewhat limited) module support in languages such as XSLT and XQuery. *Rule* languages for the Web, on the other hand, show an apparent lack of module support, despite considerable research on module extensions for classical logic programming. One of the reasons that modules are still not in the "standard repertoire" of rule languages may be the complexity of many previous approaches.

Representative and, arguably, the most comprehensive treatment of modules in logic programming is presented in [13]. It is far more expressive than our approach but at the price of a complex semantics and several operations with, in our opinion, little practical use (such as module intersection or renaming). We believe that a single well-designed union (or combination) operation with well-defined interfaces together with a strong reliance on views as an established and well-understood mechanism in rule languages is not only easier to grasp but also easier to realize.

Though many rule languages for the Web fail to provide modules, this is not true for the two preeminent Web query languages, XSLT and XQuery. XSLT [25] can be considered a rule language, however

[3] But available with examples at `http://www.reuseware.org/modularxcerptexample`.

using precedence rather than union semantics for multiple applicable rules. Rule precedence is also the dominating issue for XSLT's module system which provides intricate mechanisms for determining the precedence of rules from different modules. Nevertheless, the resulting module system is considerably less powerful (no scoped import, limited parameterization: apply-imports) yet needs a more complex semantics than module-free XSLT, quite in contrast to our approach.

It is worth mentioning that XQuery [12] also provides a module system, however without parameterization, but as a function programming language requires explicit flow control in all cases. Thus, issues such as private or public import (or the difference between import and include in XSLT) do not apply for XQuery. SPARQL [60], finally, the recently proposed RDF query language, has no concept of user defined program units (such as rules, functions, procedures, etc.) and thus no use for a module concept in the sense of our approach. However, rule-based extensions for SPARQL (in the spirit of Datalog) could certainly profit from the module system illustrated here using Xcerpt.

6.7 Conclusions and Outlook

We argue that one ingredient to cope with size and diversity of information on the (Semantic) Web is *modular* query authoring and execution. We show advantages along a concrete use case dealing with music information aggregation on the Web. Furthermore, we demonstrate how it is possible to augment existing query languages—here focused on the language Xcerpt—with new constructs while reusing already developed semantics and query engines thanks to a reduction semantics approach. The proposed module system is simple to use (in contrast to many approaches from logic programming) yet provides better encapsulation and more advanced features (such as scoping and paramterization) than module systems for XSLT or XQuery.

The proposed module system has been formalized [4] and implemented using the Reuseware Composition Framework. Integration with upcoming revisions of Xcerpt is planned. Furthermore, we would like to exploit existing techniques and tools such as Xcerpt's type system [83] for improving module composition. We are also investigating how similar techniques can be applied to add or improve module systems for other (non-rule based) query languages (for example, the module system of XSLT).

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] Java 2 platform. website. `http://java.sun.com`.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[3] Apache Software Foundation. *Apache Tomcat 6.0*, 2006. Apache Tomcat 6.0 Homepage. Retrieved on May 2007 `http://tomcat.apache.org/tomcat-6.0-doc/index.html`.

[4] 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. Submitted for publication, 2007.

[5] K. Avedal, D. Ayers, T. Briggs, C. Burnham, A. Halberstadt, R. Haynes, P. Henderson, M. Holden, S. Li, D. Malks, T. Myers, A. Nakhimovsky, S. Osmont, G. Palmer, J. Timney, S. Tyagi, G. V. Damme, M. Wilcox, S. Wilkinson, S. Zeiger, and J. Zukowski. *Professional JSP : Using JavaServer Pages, Servlets, EJB, JNDI, JDBC, XML, XSLT, and WML*. Wrox Publishing, 1 edition, January 2000.

[6] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 261–272, New York, NY, USA, 2000. ACM Press.

[7] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.

[8] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In J. Maluszinsky and N. Eisinger, editors, *Reasoning Web Summer School 2005*, number 3564 in LNCS. Springer-Verlag, 2005.

[9] S. Berger, F. Bry, T. Furche, B. Linse, and A. Schroeder. Beyond XML and RDF: The Versatile Web Query Language Xcerpt. In *Proc. Int. Conf. on World Wide Web*, 2006.

[10] W. Bibel. *Automated Theorem Proving*. Vieweg Verlag, Wiesbaden, second edition, 1987.

[11] J. Bloch. *Effective Java Programming Language Guide*. Prentice Hall, 1 edition, June 2001.

[12] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Working draft, W3C, 2005.

[13] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Trans. Program. Lang. Syst.*, 16(4):1361–1398, 1994.

[14] J. Brown. Getting Up To Date with JDBC API, August 2001.

[15] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 310–321, New York, NY, USA, 2002. ACM Press.

[16] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML Stream Query Processor SPEX. In *Proc. Intl. Conf. on Data Engineering*. IEEE, 2005.

[17] F. Bry, T. Furche, L. Badea, C. Koch, S. Schaffert, and S. Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems*, 1(2), 2005.

[18] F. Bry, T. Furche, B. Linse, and A. Schroeder. Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs. In *Proc. ACM Int'l. Workshop on Web Information and Data Management (WIDM)*. ACM Press, 2006.

[19] F. Bry and S. Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML'02)*, Sardinia, Italy, June 2002. (invited article).

[20] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of the International Conference on Logic Programming (ICLP'02)*, LNCS 2401, Copenhagen, Denmark, July 2002. Springer-Verlag.

[21] F. Bry, S. Schaffert, and A. Schroeder. A contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Proceedings of the 18th Workshop Logische Programmierung (WLP04)*, LNCS, Potsdam, Germany, 2004. Springer-Verlag.

[22] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[23] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. *XML Query Use Cases W3C Working Group Note 23 March 2007*. W3C, March 2007. W3C Working Group Note. Retrieved on June 2007 `http://www.w3.org/TR/xquery-use-cases`.

[24] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[25] J. Clark. XSL Transformations, Version 1.0. Recommendation, W3C, 1999.

[26] J. Clark, A. Slominski, S. Haustein, and J. Strachan. *JSR 173: Streaming API for XML*, January 2007. Maintenance Draft Review. Retrieved on February 2007 `http://jcp.org/en/jsr/detail?id=173`.

[27] CollabNet. *Java Compiler Compiler (JavaCC) - The Java Parser Generator*, 2007. JavaCC Homepage. Retrieved on May 2007 `https://javacc.dev.java.net`.

[28] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The Case for Virtual Register Machines. In *Proc. Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49, New York, NY, USA, 2003. ACM Press.

[29] C. Dornan, I. Jones, and S. Marlow. *Alex User Guide*. `http://www.haskell.org/alex/doc/html/alex.html`.

[30] J. Farley, W. Crawford, P. Malani, J. Norman, and J. Gehtland. *Java Enterprise in a Nutshell*. O'Reilly Media, Inc., 3 edition, November 2005.

[31] François Bry and Tim Furche and Benedikt Linse. AMaXoS Abstract Machine for Xcerpt: Architecture and Principles. In *4th Workshop on Principles and Practice of Semantics Web Reasoning*, Budva, Montenegro, June 2006. Rewerse.

[32] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, Berlin, March 1995.

[33] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, Heidelberg, 2003.

[34] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a Virtual Machine-based Platform for Trusted Computing. In *Proc. of ACM Symposium on Operating Systems Principles*, pages 193–206, New York, NY, USA, 2003. ACM Press.

[35] J. J. Garrett. Ajax: A New Approach to Web Applications. *Adaptive Path LLC*, February 2005.

[36] F. Giasson and Y. Raimond. Music ontology specification. Specification, Zitgist LLC, 2007.

[37] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[38] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive Queries over Trees. *Journal of the ACM*, 53(2), 2006.

[39] L. J. Groves and W. J. Rogers. The Design of a Virtual Machine for Ada. In *Proc. ACM Symposium on Ada Programming Language*, pages 223–234, New York, NY, USA, 1980. ACM Press.

[40] M. Hall and L. Brown. *Core Servlets and JavaServer Pages, Vol. 1: Core Technologies*. Prentice Hall, 2 edition, August 2003.

[41] J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware – adding modularity to your language of choice. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.

[42] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. *Document Object Model (DOM) Level 2 Core Specification Version 1.0*. W3C, November 2000. W3C Recommendation. Retrieved on February 2007 `http://www.w3.org/TR/DOM-Level-2-Core/`.

[43] ISO/IEC. 23271, Common Language Infrastructure (CLI). International Standard 23271, ISO/IEC, 2003.

[44] *JavaScript/DOM*, 2007. SelfHTML JavaScript Website. Retrieved on June 2007 `http://de.selfhtml.org/intro/technologien/javascript.htm`.

[45] M. V. C. Jim Melton. *XQuery API for Java (XQJ) 1.0 Specification*. Sun Microsystems, July 2006. Early Draft Review. Retrieved on February 2007 `http://jcp.org/en/jsr/detail?id=225`.

[46] T. Johnsson. Efficient Compilation of Lazy Evaluation. *SIGPLAN Notices*, 19(6), 1984.

[47] M. Kay, N. Walsh, H. Zongaro, S. Boag, and J. Tong. XSLT 2.0 and XQuery 1.0 Serialization. Working draft, W3C, 2005.

[48] J. Knoop, O. Rüthing, and B. Steffen. Optimal Code Motion: Theory and Practice. *ACM Tranactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[49] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, 2nd edition, 1999.

[50] J. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, second, extended edition, 1987.

[51] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Working draft, W3C, 2005.

[52] A. Marian and J. Siméon. Projecting XML Documents. In *Proc. Int. Conf. on Very Large Data Bases*, 2003.

[53] S. Marlow and A. Gill. *Happy User Guide*. `http://www.haskell.org/happy/doc/html/happy.html`.

[54] D. Megginson. *About SAX*, 2004. Official Website for SAX. Retrieved on February 2007 `http://www.saxproject.org`.

[55] A. Novoselsky. The Oracle XSLT Virtual Machine. In *XTech 2005: XML, the Web and beyond*, 2005.

[56] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Management*, volume 2490 of *LNCS*. Springer-Verlag, 3 2002.

[57] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML Node Labels. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 903–908. ACM Press, 2004.

[58] D. L. Overheu. An Abstract Machine for Symbolic Computation. *Journal of the ACM*, 13(3):444–468, 1966.

[59] S. Pemberton and M. Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.

[60] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Candidate recommendation, W3C, 2007.

[61] T. Przymusinsik. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann, 1988.

[62] G. Reese. *Database Programming with JDBC and Java*. O'Reilly Media, Inc., 2 edition, January 2000.

[63] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM Journal*, 12(1):23–41, January 1965.

[64] M. Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2(5):34–40, 2004.

[65] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[66] S. Schaffert. *Xcerpt Prototype*. http://demo.xcerpt.org, 2002.

[67] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.

[68] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages (Int'l. Conf. on Markup Theory & Practice)*, 2004.

[69] A. Schroeder. An Approach to Backward Chaining in Xcerpt (Projektarbeit). Master's thesis, Institute for Informatics, University of Munich, 2004.

[70] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual Machine Showdown: Stack versus Registers. In *Proc. ACM/USENIX Int. Conf. on Virtual Execution Environments*, pages 153–163, New York, NY, USA, 2005. ACM Press.

[71] Sun Microsystems. *Trail: JavaBeans(TM)*, 1995. Sun JavaBeans Tutorial Website. Retrieved on June 2007 `http://java.sun.com/docs/books/tutorial/javabeans`.

[72] Sun Microsystems. *Java SE - Java Database Connectivity (JDBC)*, 2002. Sun JDBC Homepage. Retrieved on February 2007 `http://java.sun.com/javase/technologies/database`.

[73] Sun Microsystems. *JavaServer Pages Technology - Documentation*, 2005. Sun JSP Documentation. Retrieved on July 2007 `http://java.sun.com/products/jsp/docs.html`.

[74] S. Verbaeten, K. Sagonas, and D. de Schreye. Termination Proofs for Logic Programs with Tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, January 2001.

[75] W3 Consortium. *Extensible Stylesheet Language Transformations (XSLT)*, November 1999. W3C Recommendation, `http://www.w3.org/TR/xslt`.

[76] W3 Consortium. *XML Path Language (XPath)*, November 1999. `http://www.w3.org/TR/xpath`.

[77] W3 Consortium. *XQuery: A Query Language for XML*, February 2001. W3C Recommendation, `http://www.w3.org/TR/xquery/`.

[78] W3 Consortium. *Extensible Markup Language (XML) 1.1*, February 2004. W3C Recommendation, `http://www.w3.org/TR/2004/REC-xml11-20040204/`.

[79] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proceedings of the International Conference on Functional Programming*, Paris, September 1999. `http://www.cs.york.ac.uk/fp/HaXml/icfp99.html`.

[80] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[81] D. S. Warren. Memoing for Logic Programs. *Communications of the ACM*, March 1992.

[82] Information retrieval query language, February 2007. Retrieved on July 2007 `http://en.wikipedia.org/wiki/Information_retrieval_query_language`.

[83] A. Wilk and W. Drabent. A Prototype of a Descriptive Type System for Xcerpt. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning, Budva, Montenegro (10th–11th June 2006)*, volume 4187 of *LNCS*, pages 262–275. REWERSE, 2006.

[84] J. Würtz and T. Müller. Constructive disjunction revisited. In *KI - Künstliche Intelligenz*, pages 377–386, 1996.