I4-D11

Refinement of Principles of the Xcerpt Processors

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D11/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Dimitris Plexousakis and Jose Alferes
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	31 August 2006
Actual submission date:	10 April 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, so far it lacks an efficient implementation to supplement the convenient language features. AMAχOS is an abstract machine implementation for Xcerpt that aims at efficiency and ease of deployment. The first part of this deliverable summarizes the motivation and principles behind AMAχOS and discusses how its current architecture realizes these principles.

The second part of this deliverable describes the operational semantics of the Xcerpt prototype. The operational semantics consists of three parts: the description of the constraint solver, the definition of simulation unification, and the description of the (backward) chaining in the prototype. The simulation unification is the core concept of the Xcerpt prototype, unique among Web query and logic languages.

Keyword List

evaluation,principles,abstract machine,Xcerpt,XML,RDF

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

© REWERSE 2007.

Refinement of Principles of the Xcerpt Processors

Sebastian Schaffert[1], **François Bry**[1], **Tim Furche**[1], **Benedikt Linse**[1], **Andreas Schroeder**[1]

[1] Institute for Informatics, University of Munich, Germany
Email: ⟨*first-name*⟩.⟨*last-name*⟩@ifi.lmu.de

10 April 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, so far it lacks an efficient implementation to supplement the convenient language features. AMAχOS is an abstract machine implementation for Xcerpt that aims at efficiency and ease of deployment. The first part of this deliverable summarizes the motivation and principles behind AMAχOS and discusses how its current architecture realizes these principles.

The second part of this deliverable describes the operational semantics of the Xcerpt prototype. The operational semantics consists of three parts: the description of the constraint solver, the definition of simulation unification, and the description of the (backward) chaining in the prototype. The simulation unification is the core concept of the Xcerpt prototype, unique among Web query and logic languages.

Keyword List
evaluation,principles,abstract machine,Xcerpt,XML,RDF

Contents

1 Principles of the Abstract Machine: Xcerpt 2.0 **1**
1.1 Introduction . . . 1
1.2 A Brief History of Abstract Machines . . . 2
1.3 Xcerpt: A Versatile Web Query Language . . . 4
1.3.1 Data as Terms . . . 4
1.3.2 Queries as Enriched Terms . . . 4
1.3.3 Programs as Sets of Rules . . . 4
1.4 Architecture: Principles . . . 4
1.4.1 "Execute Anywhere"—Unified Query Execution Environment . . . 5
1.4.2 "Compile Once"—Separation of Compilation and Execution . . . 5
1.4.2.1 Extensive static optimization. . . . 5
1.4.3 "Compile, Classify, Execute"—Unified Evaluation Algorithm . . . 6
1.4.4 "Optimize All the Time"—Adaptive Code Optimization . . . 7
1.4.5 "Distribute Any Part"—Partial Query Evaluation . . . 7
1.5 Architecture: Overview . . . 8
1.5.1 AMAχOS Core . . . 10
1.5.2 Query Compiler . . . 11
1.6 Conclusion and Outlook . . . 12

2 Operational Semantics of Xcerpt 1.0 **13**
2.1 A Simple Constraint Solver . . . 14
2.1.1 Data Structures and Functions . . . 14
2.1.1.1 Constraints . . . 14
2.1.1.2 Functions . . . 16
2.1.2 Solution Set of a Constraint Store . . . 16
2.1.3 Constraint Simplification . . . 18
2.1.4 Consistency Verification Rules . . . 18
2.1.4.1 Rule 1: Consistency . . . 18
2.1.4.2 Rule 2: Transitivity . . . 19
2.1.5 Constraint Negation . . . 19
2.1.5.1 Rule 3: Consistency with Negation . . . 19
2.1.5.2 Rule 4: Transitivity with Negation . . . 20
2.1.5.3 Rule 5: Negation as Failure . . . 21
2.1.6 Program Evaluation . . . 21
2.2 Simulation Unification . . . 22

2.2.1 Simulation Unifiers 22
2.2.2 Decomposition Rules 24
2.2.2.1 Preliminaries 24
2.2.2.2 Root Elimination 25
2.2.2.3 $\rightsquigarrow$ Elimination 26
2.2.2.4 Descendant Elimination 26
2.2.2.5 Decomposition with `without` 27
2.2.2.6 Decomposition with `optional` in the query term 27
2.2.2.7 Incomplete Decomposition with grouping constructs, functions, aggregations, and optional subterms in construct terms 29
2.2.2.8 Term References: Memoing of Previous Computations 30
2.2.3 Soundness and Completeness 31
2.3 Backward Chaining 31
2.3.1 Dependency Constraint 32
2.3.2 Query Unfolding 33
2.3.3 Soundness and Completeness 34
2.3.3.1 Soundness 35
2.3.3.2 Completeness 35
2.3.3.3 Criteria for Termination 36

Chapter 1

Principles of the Abstract Machine: Xcerpt 2.0

1.1 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 [28]). 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 [3,1,6]) 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 1.4.2.

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 1.4.1.

3. "Optimize all the time"—not only are queries optimized predictively during query compilation, but also adaptively during execution. For details see Section 1.4.4.

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 1.4.3 and [10]).

Following a brief look at the history of abstract and virtual machines for program and query execution (Section 1.2) and an introduction into Xcerpt (Section 1.3), the versatile Web query language that is implemented by the AMAχOS abstract machine, we focus in the course of this article first (Section 1.4) on a discussion of the *principles* of this abstract machine that also serves as a further motivation of the setting. The second part (Section 1.5) of the paper discusses the proposed *architecture* of AMAχOS and how this architecture realizes the principles discussed in the first part.

1.2 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. [17] 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 [11] and trusted computing [16] 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 [31], 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, [32]), Ada [19], Prolog [42], and functional programming languages (G-machine, [21]) 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 [36] for XSB Prolog.

Recently, the field has seen a reinvigoration, cf. [35], 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$^\sharp$. 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 [24] and Microsoft's common language infrastructure [20] (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 [13] and [40] 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 [28] 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/

1.3 Xcerpt: A Versatile Web Query Language

Xcerpt is a query language designed after principles given in [7] 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`.

1.3.1 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).

1.3.2 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).

1.3.3 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 [37, 38].

1.4 Architecture: Principles

The abstract machine for Xcerpt, in the following always referred to as AMAχOS, and its architecture are organized around five guiding principles:

1.4.1 "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 1.4.5.

1.4.2 "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.

1.4.2.1 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 1.5.2 and Figure 1.5 present a more detailed view of the query compiler and optimizer employed in the AMAχOS virtual

[4] http://www.saxonica.com/

[5] http://xml.apache.org/xalan-j/

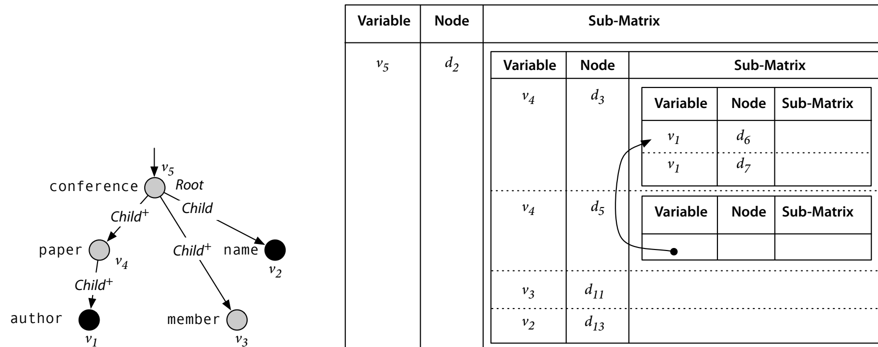

Figure 1.1: Sample Query and Memoization Matrix

machine. To be applicable to different scenarios, a control API for the query compilation stage allows the configuration 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. [29, 10]) 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.

1.4.3 "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 [37] and refined to guarantee polynomial size in [10]), 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 1.1: 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 1.1 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 [18]), 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 [10]. 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 [10], 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 1.5.1), the second must be controlled by the execution control API, cf. Section 1.5.

1.4.4 "Optimize All the Time"—Adaptive Code Optimization

As argued above in Section 1.4.2 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 1.4.3) 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. [2].

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.

1.4.5 "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 1.2 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

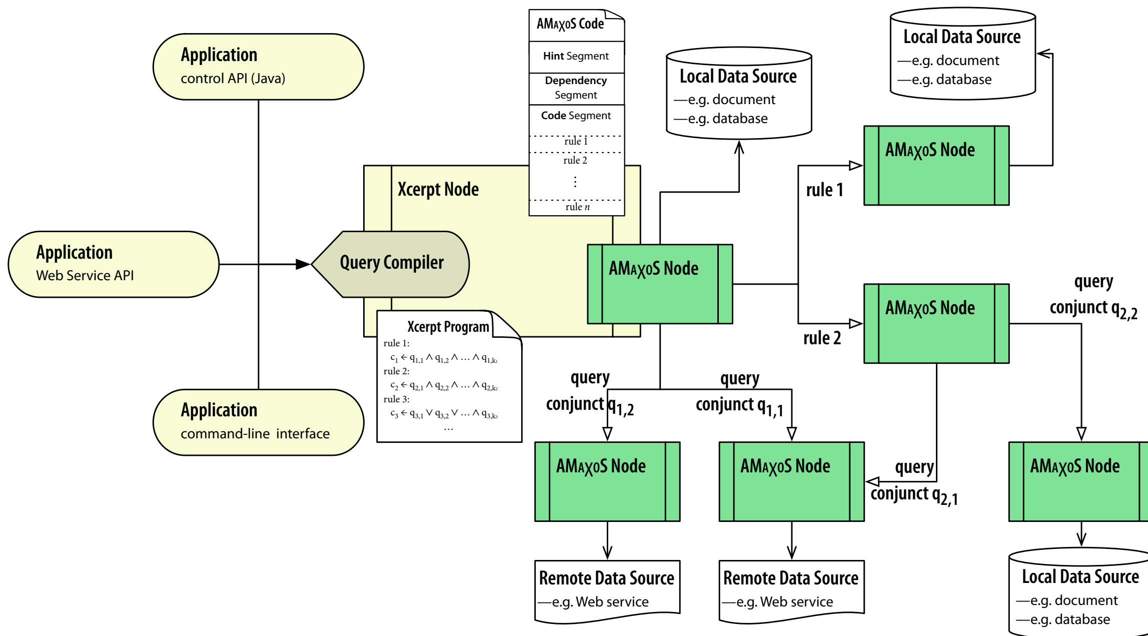

Figure 1.2: Query Node Network

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 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.

1.5 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 1.5.1 and 1.5.2.

Figure 1.3 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.

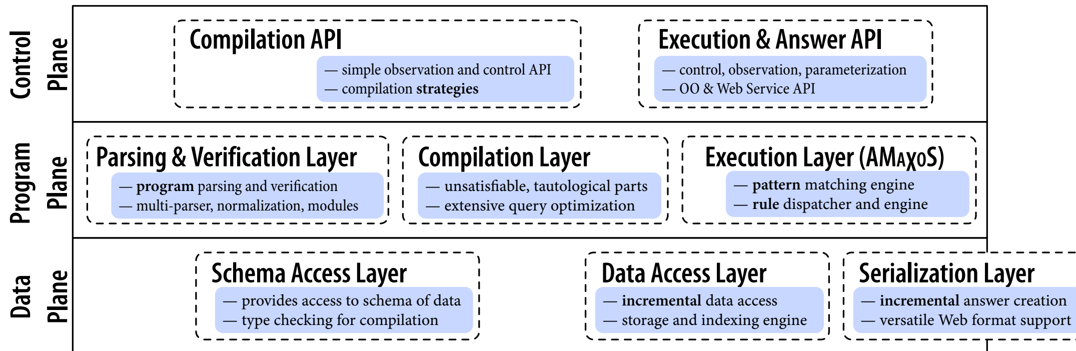

Figure 1.3: Overview of AMAχOS Components

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 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 1.5.2. 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 1.5.1.

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 [27]). 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 [22], 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.

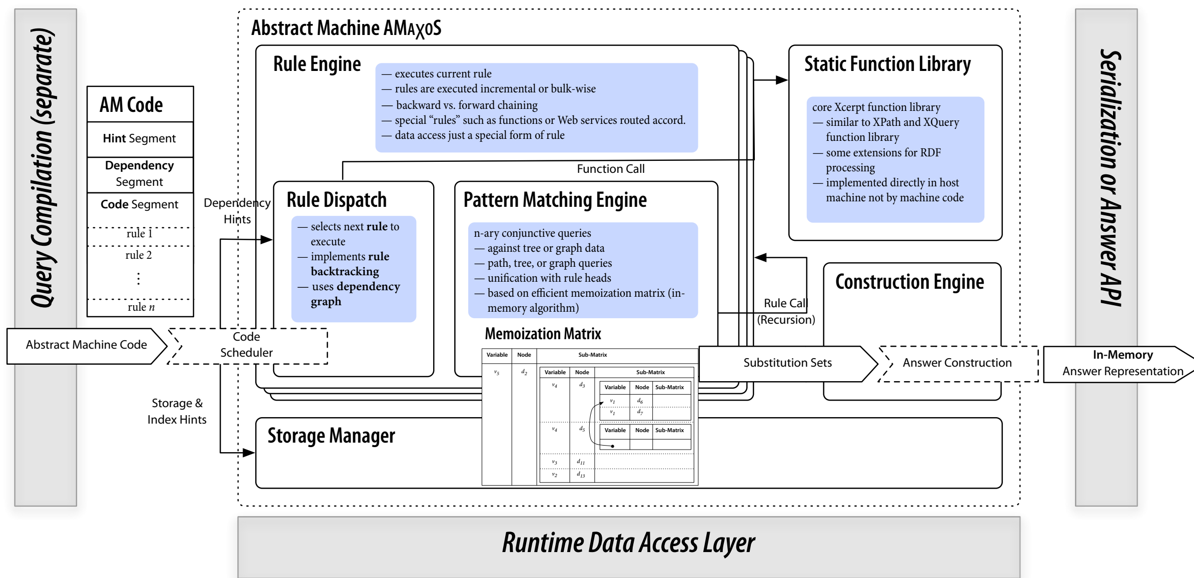

Figure 1.4: Architecture of Core Query Engine AMAχOS

1.5.1 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 1.5.2) is evaluated against data provided by the runtime data access layer resulting in answers that are serialized by the serialization API.

As shown in Figure 1.4, 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 [10] 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 [26] 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

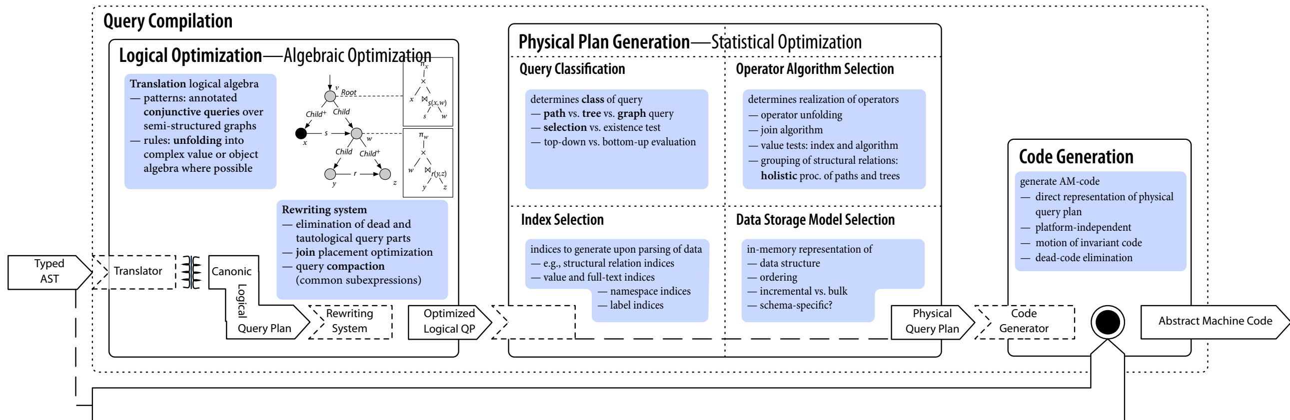

Figure 1.5: Architecture of Query Compiler for AMAχOS

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 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).

1.5.2 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 [10]. 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., [5, 30].

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 1.5.1.

Therefore, the *code generator* is rather simple, performing only basic serialization tasks and simple code optimizations such as motion of invariant code [23].

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.

1.6 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 [10] and in [4], demonstrating the promise of the discussed method and architecture.

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. [15]). 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. [37]. It has first been proposed in [8] and is further refined here. Based on simulation unification, a *backward chaining* algorithm is then described that eventually determines answer terms as defined in [37]. 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 [37] 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 thesis has the following form:

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

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* [14], 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 [44]. The approach taken in this thesis is rather simplistic, as it after each application of a simplification rule creates the disjunctive normal form (DNF) of the constraint store. Simplification rules are independently applied to the different conjuncts of the DNF. This approach is rather inefficient in implementations, and various optimisations can be considered. A straightforward optimisation would be to not create the DNF after *each* simplification step, but instead only if it is "necessary", because no other simplification rules apply. However, such optimisations are not further investigated in this thesis, as the focus is on Web query languages and not on constraint programming.

Furthermore, the constraint solver needs to be able to treat negation. As both negation constructs `not` and `without` describe negation as failure, the negation behaves differently to classic negation in some cases (cf. Example 4). 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 thesis, constraints are defined by the following grammar (defined in a variant of *Extended Backus-Naur Form*):

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

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

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

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

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

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

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

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

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

The justification for the dependency constraint are the requirements of the grouping constructs `all` and `some`, which require to consider all alternative solutions for the query part of a rule. If `all` or `some` appears in the head of a rule which is evaluated, the unification of a query with the head cannot be completed before the rule is fully evaluated.

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 $X \quad t$ 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
- $substitutions(False) \quad \{\}$, 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 [37] 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.

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

Depending on the constraint store, this solution set is restricted to substitutions fulfilling certain conditions. For example, the constraint $X \preceq_u f\{a\}$ requires that all substitutions in the solution set provide an assignment for the variable X that is compatible (i.e. *simulates*) with $f\{a\}$. Likewise, the constraint $f\{\{\}\} \preceq_u X$ requires that the solution set only contains substitutions that provide an assignment t for X such that $f\{\{\}\} \preceq t$.

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

Definition 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. [37])*
- $\Omega(False)\ \{\}$*, i.e. the empty set*
- $\Omega(X \preceq_u t)$ *is the set of all all-grounding substitutions* σ *such that* $\sigma(X) \sim \sigma(t)$
- $\Omega(t \preceq_u X)$ *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.

$$\frac{\begin{array}{l} X \preceq_u t_1 \\ X \preceq_u t_2 \end{array}}{X \preceq_u t_1 \wedge t_1 \preceq_u t_2 \wedge t_2 \preceq_u t_1}$$

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 1 (Consistency Rule)

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

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 [37] 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'".

$$\begin{array}{l}
t_1 \ \preceq_u \ t'_1 \text{ such that } t'_1 \text{ contains the variable } X \\
X \ \preceq_u \ t_2 \\
\hline
X \ \preceq_u \ t_2 \ \wedge \ t_1 \ \preceq_u \ t'_1[t_2/X]
\end{array}$$

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 2 (Transitivity Rule)

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

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

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 $\neg\neg C \ \not{C}$, 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:

$$\frac{\begin{array}{l} X \preceq_u t_1 \\ \neg(X \preceq_u t_2) \end{array}}{X \preceq_u t_1 \wedge \neg(t_1 \preceq_u t_2 \wedge t_2 \preceq_u t_1)}$$

Example 3 (Consistency Rule with Negation) *Consider the constraint store*

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

Applying the consistency rule with negation yields

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

the DNF of which is

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

and after further decomposition steps

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

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.

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

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.

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

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 4 (Negation as Failure Rule) *Consider the simulation constraint $\neg\neg(X \preceq_u t)$ such that X does not occur elsewhere in a non-negated simulation constraint. Applying Rule 5 to this constraint yields $\neg False$ $True$ (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(*$\langle Q\rangle_\emptyset$*)*
        print *apply(t,Subst)*

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

Algorithm 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 [8], 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. [34]), 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 $X \quad t$. 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 [37] and instead of a single substitution, substitution sets as in [37] 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 $\Sigma \quad \{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto b, Y \mapsto a\}$, a query term $t^q \quad f\{var\ X\}$ and a construct term $t^c \quad f\{var\ Y\}$. 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 [37] 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 5 (Simulation Unifiers) *1. Let* t^q $f\{\{var\ X, b\}\}$ *and let* t^c $f\{a, var\ Y, c\}$*. A simulation unifier of* t^q *in* t^c *is the (all-grounding) substitution set*

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

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

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

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 $t \not\perp$ 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* $t^1 \;\; l\{t^1_1,\dots,t^1_n\}$ *and* $t^2 \;\; l\{t^2_1,\dots,t^2_m\}$*, the following sets of functions* $\Pi_X : \langle t^1_1,\dots,t^1_n\rangle \rightarrow \langle t^2_1,\dots,t^2_m\rangle$ *are defined:*

- $SubT \subseteq \langle t^1_1,\dots,t^1_n\rangle$ *is the sequence of all non-negated subterms of* t^1 *and* $SubT^- \subseteq \langle t^1_1,\dots,t^1_n\rangle$ *is the sequence of all negated subterms of* t^1
- $SubT^! \subseteq \langle t^1_1,\dots,t^1_n\rangle$ *is the sequence of all non-optional subterms of* t^1 *and* $SubT^? \subseteq \langle t^1_1,\dots,t^1_n\rangle$ *is the sequence of all optional subterms of* t^1
- Π *is the set of partial, index injective functions* π *from* $\langle t^1_1,\dots,t^1_n\rangle$ *to* $\langle t^2_1,\dots,t^2_m\rangle$ *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 6 *Consider the terms* $t^1 \;\; f[[a, \text{without } b]]$ *and* $t^2 \;\; f[a,b,c]$*. The set of index monotonic mappings of the set of subterms of* t^1 *into the set of subterms of* t^2 *(*Π_{mon}*) is as follows (*`without` *b abbreviated as* $\neg b$*):*

$$\begin{array}{lll} \{a \mapsto a, \neg b \mapsto \perp\} & \{a \mapsto b, \neg b \mapsto \perp\} & \{a \mapsto c, \neg b \mapsto \perp\} \\ \{a \mapsto a, \neg b \mapsto b\} & \{a \mapsto b, \neg b \mapsto c\} & \\ \{a \mapsto a, \neg b \mapsto c\} & & \end{array}$$

Note that all these mappings can be generated in a rather straightforward manner by creating a table with the terms $t_1^1 \cdots t_n^1$ arranged top-down and the terms $t_1^2 \cdots t_m^2$ arranged left-right and then determining paths from the first line to the nth 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 equivalent 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***:

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

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

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

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 $l_1 \not= l_2$.
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***:

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

2.2.2.5 Decomposition with `without`

The declarative specification of `without` in the ground query term simulation of [37] 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 $\perp$.

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

*Decomposition Rule **without***:

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

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

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

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 $\pi(t) \not\perp$, 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. Con-

sider for example the term

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

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

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

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:

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

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

$$\Sigma\ \{\{X \mapsto a, Z \mapsto c\}\}$$

whereas the evaluation of the second set of terms would yield

$$\Sigma\ \{\{X \mapsto a, Z \mapsto c\}, \{X \mapsto a\}\}$$

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***:

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

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 [43, 12].

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

```
fib :: Int →Int
fib 0 = 1                                      2
fib 1 = 1
fib n = fib (n-1) + fib (n-2)                  4
```

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.

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

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 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 [25]. 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 algorithm Different approaches to backward chaining in Xcerpt have been considered in the course of this thesis [9, 39]. The approach presented here is a further refinement of the "all at once" approach presented in [9].

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:

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

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

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

X. The following refinement of the rule above uses the incomplete decomposition of `all` and `some` to add such information:

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

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:

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

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

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

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

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.

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

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

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".

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

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

In the following, let $\mathcal{P}_{grouping} \subseteq P$ be the set of program rules $t^c \leftarrow Q$ such that t^c contains grouping constructs, functions, aggregations, or optional subterms, and let $\mathcal{P}_{nongrouping} \subseteq P$ bet the set of program rules $t^c \leftarrow Q$ such that t^c does not contain grouping constructs, functions, aggregations, or optional subterms. Note that goals are not considered in either case, as they do not participate in chaining. Furthermore, $n \geq 0$ and $m \geq 0$.

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

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 [37], 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 $M_P \models \Sigma(Q)$.*

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 [37], 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 Σ $substitutions(\mathcal{CS})$, 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 Ψ $\Omega(CS)$ is a maximal substitution set with $M_{P'} \mid \Psi(Q)$.

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 $M_P \mid \Psi(Q)$, 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 Ψ $\Omega(CS)$. 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 [37]. This is easy to see on a small example. Consider the program

$$
\begin{array}{l}
f\{a\} \leftarrow f\{a\} \\
f\{a\}
\end{array}
$$

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 [25] 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

$$
\begin{array}{l}
g\{all\ var\ X\} \leftarrow f\{var\ X\} \\
f\{a\} \leftarrow f\{a\} \\
f\{a\}
\end{array}
$$

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 Ψ $\Omega(CS)$ is a maximal substitution with $M_P \,|\Psi(Q)$, 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 $\Sigma \ \mathit{substitutions}(CS) \ \Omega(CS)_{|V}$ where Σ yields the same ground instances of t^c as $\Omega(CS)$. Thus, Σ is also maximal wrt. $\Sigma(t^c) \subseteq M_P$. .

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 [37] didn't use recursion despite the rather complex task. Likewise, many of the other examples of [37] 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* [33]).

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 [12] 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 [41]. In particular, it avoids the problem described above.

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.

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

`http://rewerse.net`).

Bibliography

[1] 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.

[2] 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.

[3] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] F. Bry, A. Schroeder, T. Furche, and B. Linse. Efficient Evaluation of n-ary Queries over Trees and Graphs. Submitted for publication, 2006.

[11] 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.

[12] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[13] 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.

[14] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, Berlin, March 1995.

[15] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, Heidelberg, 2003.

[16] 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.

[17] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[18] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive Queries over Trees. *Journal of the ACM*, 53(2), 2006.

[19] 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.

[20] ISO/IEC. 23271, Common Language Infrastructure (CLI). International Standard 23271, ISO/IEC, 2003.

[21] T. Johnsson. Efficient Compilation of Lazy Evaluation. *SIGPLAN Notices*, 19(6), 1984.

[22] M. Kay, N. Walsh, H. Zongaro, S. Boag, and J. Tong. XSLT 2.0 and XQuery 1.0 Serialization. Working draft, W3C, 2005.

[23] 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.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, 2nd edition, 1999.

[25] J. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, second, extended edition, 1987.

[26] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Working draft, W3C, 2005.

[27] A. Marian and J. Siméon. Projecting XML Documents. In *Proc. Int. Conf. on Very Large Data Bases*, 2003.

[28] A. Novoselsky. The Oracle XSLT Virtual Machine. In *XTech 2005: XML, the Web and beyond*, 2005.

[29] 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.

[30] 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.

[31] D. L. Overheu. An Abstract Machine for Symbolic Computation. *Journal of the ACM*, 13(3):444–468, 1966.

[32] S. Pemberton and M. Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.

[33] 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.

[34] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM Journal*, 12(1):23–41, January 1965.

[35] M. Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2(5):34–40, 2004.

[36] 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.

[37] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.

[38] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.

[39] A. Schroeder. An Approach to Backward Chaining in Xcerpt (Projektarbeit). Master's thesis, Institute for Informatics, University of Munich, 2004.

[40] 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.

[41] 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.

[42] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[43] D. S. Warren. Memoing for Logic Programs. *Communications of the ACM*, March 1992.

[44] J. Würtz and T. Müller. Constructive disjunction revisited. In *KI - Künstliche Intelligenz*, pages 377–386, 1996.