# Completing Queries: Rewriting of Incomplete Web Queries under Schema Constraints

Sacha Berger, François Bry, Tim Furche, and Andreas J. Häusler

Institute for Informatics, University of Munich,
Oettingenstraße 67, D-80538 München, Germany
http://www.pms.ifi.lmu.de/

**Abstract.** Web queries have been and will remain an essential tool for accessing, processing, and, ultimately, reasoning with data on the Web. With the vast data size on the Web and Semantic Web, reducing costs of data transfer and query evaluation for Web queries is crucial. To reduce costs, it is necessary to narrow the data candidates to query, simplify complex queries and reduce intermediate results.

This article describes a static approach to optimization of web queries. We introduce a set of rules which achieves the desired optimization by schema and type based query rewriting. The approach consists in using schema information for removing incompleteness (as expressed by 'descendant' constructs and disjunctions) from queries. The approach is presented on the query language Xcerpt, though applicable to other query languages like XQuery. The approach is an application of rules in many aspects—query rules are optimized using rewriting rules based on schema or type information specified in grammar rules.

## 1 Introduction

Web queries have been and, by all accounts, will remain an essential tool for accessing, processing, and, ultimately, reason with data on the Web. They are an essential component of many Web rule languages for expressing conditions on the information available on the Web. Web queries occur in so diverse rule languages as XSLT, CSS, Xcerpt, WebSQL, RVL, and dlvhex. The perceived strength and main contribution of Web queries and their underlying semi-structured data model is the ability to model data with little or no (a priori) information on the data's schema. In this spirit, all semi-structured query languages are distinguished from traditional relational query languages in providing core constructs for expressing *incomplete* queries, i.e., queries where only some of the sought-for data is specified but that are not affected by the presence of additional data. Examples of such constructs are regular path expressions in Lorel, the **`descendant`** and **`following`** closure axis in XPath and XQuery, descendant and adjacency selectors in CSS, or Xcerpt's **`desc`** and partial query patterns. Incompleteness constructs often, in particular if concerned with navigation in the graph, resemble to reachability or transitive closure constructs.

Incomplete query constructs have proved to be both essential tools for expressing Web queries and a great convenience for query authors able to focus better on the parts of the query he or she is most interested in. Though some evaluation approaches, e.g.,

[5] (usually limited to tree-shaped data) can handle certain incomplete queries (viz., those involving **`descendant`** or **`following`**) efficiently, most approaches suffer from lower performance for evaluating incomplete queries than for evaluating queries without incompleteness. The latter is particularly true for query processors with limited or no index support (a typical case in a Web context where query processors are often used in scenarios where data is transient rather than persistent).

In this paper, we propose a set of equivalences for removing (or introducing) queries with incomplete constructs. Our main contributions are as follows:

*First,* we discuss the types of incompleteness that occur in Web query languages and how they can be rewritten in Section 1.1. In this and the following parts, we have chosen our own query language Xcerpt for providing examples, mostly as it is able to express all forms of incompleteness that we consider in this article conveniently. We do not, however, rely on any specialized evaluation engine. The query rewriting equivalences are purely static and can be applied separately in a pre-processing step or during logical query optimization. It is worth noting that, where XQuery allows the distinction between complete and incomplete queries as in the case of the descendant axis, our equivalences can as well be used for rewriting XQuery expressions.
*Second,* in Section 2, we introduce the query language Xcerpt and its types, based on a graph schema language and a convenient automaton model for specifying and checking schema constraints on graph-shaped semi-structured data. The automaton model is exploited to be able to specify the equivalences introduced in the second part of the article concisely.
*Third,* we introduce a collection of equivalences for removing all forms of incompleteness discussed in the first part. These equivalences (Section 3) can actually be used in both directions, i.e., they could also be used to introduce incompleteness into a complete query. In contrast to previous work on minimization and containment under schema constraints, these equivalences operate on graph schemata and graph queries instead of tree schemata and queries.
*Fourth,* we discuss briefly how these equivalences can be exploited for query optimization (Section 4), both in a context where incompleteness is undesirable from the point of evaluation cost and in a context where at least certain incomplete queries can be evaluated as fast as equivalent complete queries, e.g., [5].

### 1.1 Three Forms of Incompleteness

In Web queries, incompleteness occurs in three forms: breadth, depth, and order. In this article, we focus mostly on breadth and depth though we briefly consider also order incompleteness.

1. Incompleteness in depth allows a query to express restrictions of the form “there is a path between the paper and the author” without specifying the path’s exact shape. The most common construct for expressing depth incompleteness is XPath’s **`descendant`** or Xcerpt’s **`desc`**, an unqualified, arbitrary-length path between two nodes. Regular path expressions and Xcerpt’s qualified **`desc`** allow more specific restrictions, e.g., “there is a path between paper and author and it contains no institutions”.

2. Incompleteness in breadth allows a query to express restrictions on some children of a node without restricting others ("there is an author child of the paper but there may be other unspecified children"). Breadth incompleteness is an essential ability of all query languages. Indeed, in many languages breadth completeness is much harder to express than incompleteness. Nevertheless, breadth completeness allows e.g. indexed access to a node's children (often preferable to a "search-always" model).
3. Incompleteness in order allows a query to express that the children order of a node is irrelevant ("there is an author child of the paper and a title child of the same paper but don't care about their order").

In Section 3, we discuss how the first two forms of incompleteness can be rewritten and briefly mention how the last form could be treated as well.

## 2 Preliminaries—Brief Introduction to Xcerpt and $R_2G_2$ Types

The query and transformation language Xcerpt [15], is a declarative, logic based Web query language. Its salient features are **pattern based query** and **construction** of graph-shaped semi-structured data, possibly **incomplete query patterns** reflecting the *heterogeneity* and the *semi-structured* nature of Web data, **rules** relating query and construction, and **rule chaining** enabling simple inference and query modularization.

As we focus on query rewriting and optimization in this article, Xcerpt queries will be introduced, construct terms and rules are omitted. For more details about the Xcerpt query language refer, e.g., to [15].

An Xcerpt term (query- construct- or data term) represents a tree- or graph-like structure, it consists of a label and a sequence of child terms enclosed in braces or brackets. Square brackets (i.e., [ ]) denote *ordered term specification* (as in standard XML), curly braces (i.e., { }) denote *unordered term specification* (as is common in databases). Double braces (i.e., [[ ]] and {{ }}) are used to denote that a term's content is just partially specified—this concept only applies to query terms. This so called *incompleteness in breadth* denotes, that additional child terms may be interspersed in data matching this incomplete query term, among the ones specified in the query.

Graph structure can be expressed using a reference mechanism, but is not further introduced as not considered in the current rewriting rules.

*Queries* are connections of zero or more query terms using the n-ary connectives *and* and *or*. A query is always (implicitly or explicitly) associated with a *resource*.

Query terms are similar to (possibly) *non-ground* functional programming expressions and logical atoms. Query terms are Xcerpt terms, Xcerpt terms prefixed by the **desc**-Keyword, query variables, or ★. A query term `l[ desc a[] ]` may match a data term with label `l` and child term `a[]` or with any child term that contains `a[]` at arbitrary depth as descendant or child term. A variable, e.g. *var* `X`, may match any term, multiple occurrences of the same variable have to match equivalent data terms. Variables can be restricted, e.g. *var* `X` $\rightarrow q$ denotes, that the variable may only match with terms matching the query term $q$. The term ★ denotes the most general query

matching any data term. It can also be seen as an anonymous variable. While not formally part of Xcerpt, it is under current investigation for further versions of Xcerpt and it is a short hand for a query term of constant length.

Query terms are *unified* with database or construct terms using a non-standard unification called *simulation unification*, which has been investigated in [7]. Simulation unification is based on *graph simulation* [1] which is similar to graph homomorphisms.

*Typed Xcerpt* is the basis for (static and dynamic) type checking of Xcerpt and for the optimization presented in this article. In a (fully) typed Xcerpt program, every term $t$ is annotated with a disjunction of types $\tau_1, \ldots, \tau_n$, denoted as $t : (\tau_1 | \ldots | \tau_n)$. Types are defined using a so called "regular rooted graph grammar", short $R_2G_2$. Internally, types are represented as automata, type automata are used for query rewriting. A type represents a set of data terms valid w.r.t. the given term. A well typed query w.r.t. a given type is a query that may match some data terms that are valid w.r.t. the given type.

$R_2G_2$ is a slight extension of regular tree grammars [6], the extensions cope with typed references (not introduced here) used to model graph shaped data and unordered child lists (neither introduced) as defined in Xcerpt.

While $R_2G_2$ grammars are convenient for the user, $R_2G_2$ is translated into an instance of a tree automaton model appropriate for processing. Tree automata for ranked trees are well established [10]. As Xcerpt and XML is based on an unranked tree model, a new automaton model specially tailored for unranked trees has been proposed [2]. A non-deterministic regular tree automaton $M$ is a 5-tuple $(Q, \Delta, F, R, \Sigma)$ with label alphabet $\Sigma$, states $Q$, final states $F$ where $F \subseteq Q$, transitions $\Delta$ where $\Delta \subseteq (Q \times \Sigma \times Q \times Q)$ and a set of root transitions $R$ with $R \subseteq \Delta$. [1] The unorthodox about these automata are the edges—they are hyper edges of arity 3. An edge $(s, l, c, e)$ represents a transition from state $s$ to state $e$ consuming (in the sense of automata acceptance) a data term with label $l$ and a sequence of child terms accepted by a part of the automaton with start state $c$. Figure 1 shows an example automaton used through out the reminder of this article as example type.

In practice, various atomic data types like string, integer, and boolean also exist, but as they are arguably not relevant for structure based optimization they are omitted here.

In this article, in a typed term of the shape $t : \tau$ we will consider $\tau$ to correspond to an edge in the automaton. The example data term in the caption of figure 2 is hence type annotated under $M$ as

```
z[ a[ c[]:(6,c,8,9),d[]:(9,d,10,11)]:(2,a,6,7)]:(1,z,2,3).
```

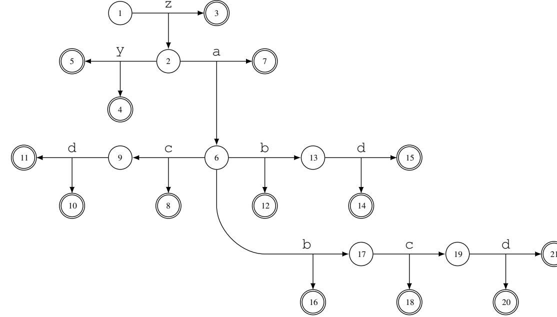
## 3 Rules for Completing Queries under a Schema

The previous sections have established the formal aspects of the query and schema language employed and intuitively established the aim of rewriting incomplete queries under a given schema. The following section defines in a precise and formal manner a set of rules for this task.

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

**Fig. 1** This automaton represents the type used in the following rewriting rule example. An example data term valid w.r.t. this type is e.g. `z[ a[ d[],c[] ] ]`

$$
\begin{aligned}
M = (\ & \{\ 1,\dots,20\},\\
& \{\ (1,z,2,3),\\
& \quad (2,y,4,5),\\
& \quad (2,a,6,7),\\
& \quad (6,c,8,9),\\
& \quad (9,d,10,11),\\
& \quad (6,b,12,13),\\
& \quad (13,d,14,15),\\
& \quad (6,b,16,17),\\
& \quad (17,c,18,19),\\
& \quad (19,d,20,21)\},\\
& \{\ 3,4,7,8,10,12,14,16,18,20\},\\
& \quad (1,z,2,3),\\
& \{\ a,b,c,d,y,z\})
\end{aligned}
$$




Recall, that these are merely equivalence rules and not a full optimization algorithm. Note also, that for simplicity these rules apply only to non-recursive schemata (no recursion in either depth or breadth). However, this is only needed due to the naive application of the rules until no further expansion of rules is possible. This limitation is not needed if these rules are part of an optimization algorithm that chooses when to further expand and when to stop (e.g., because the size increase offsets the gain from the reduction of incompleteness—cf. Section 5). Furthermore, no order of application for the rules is given—see also the outlook for a brief discussion on possible optimization strategies incorporating our rules. For the examples in this article, we have chosen to apply the rules in the most convenient way.

### 3.1 Prerequisites

There are a number of convenience functions that allow more concise rule definitions:

(1) HORIZONTALPATHSTOENDSTATES($s$) (*hptes*): The *hptes* function takes as argument a state $s$ and returns a set $\{\tau_1 = [t_{1_1},\dots,t_{1_{m_1}}],\dots,\tau_n = [t_{n_1},\dots,t_{n_{m_n}}]\}$ containing lists of paths $\tau_i$ from $s$ to all end states reachable from $s$.

(2) HORIZONTALPATHSTOSTATES($s_1$, $s_2$) (*hpts*): For a given state $s_1$ of an $R_2G_2$ graph, the *hpts* function returns a set of all paths $\tau_i$ through the graph which begin with $s_1$ and end with state $s_2$.

(3) MAP(T, E): Given an Xcerpt term $t$ and a sequence of transitions (edges) $E = [e_1,\dots,e_n]$ in a given $R_2G_2$ graph, *map* returns the sequence $[t : e_1,\dots,t : e_n]$.

### 3.2 The set of rules

Now the rules for rewriting queries are defined. Afterwards, and before describing how the rules actually work, we give an example demonstrating the effects of rule application to an example query (Figure 3.2).

$$
\frac{(\textbf{desc}\ t : \tau) : (s, \mathtt{l}, c, e)}{\mathtt{l[[}(\textbf{desc}\ t : \tau) : (s', \mathtt{l}', c', e')\mathtt{]]} : (s, \mathtt{l}, c, e)}\ (\text{DESC1}) \qquad \frac{\mathit{var}\ \mathtt{X} : \tau}{\mathit{var}\ \mathtt{X} \rightarrow \bigstar : \tau}\ (\text{VAR})
$$

$$\frac{(\texttt{desc}\ t:\tau):(s,\texttt{l},c,e)}{t:(s,\texttt{l},c,e)}\quad(\textsc{Desc2})\qquad\qquad \frac{\bigstar:(s,\texttt{l},c,e)}{\texttt{l [[ ]]}:(s,\texttt{l},c,e)}\quad(\textsc{Star})$$

$$\frac{\texttt{l[[}t_1,t_2,\ldots,t_q\texttt{]]}:(s,\texttt{l},c,e)}{\texttt{or}\begin{bmatrix}\ldots\\ \texttt{l[}map(\bigstar,z_1),t_1,map(\bigstar,z_2),t_2,\ldots,map(\bigstar,z_q),t_q,map(\bigstar,z_{q+1})\texttt{]}\\ \ldots\end{bmatrix}}$$
$$z_1,\ldots,z_{q+1}\in hpts(c,s_{t_1})\times hpts(e_{t_1},s_{t_2})\times\ldots\times hpts(e_{t_{q-1}},s_{t_q})\times hptes(e_{t_q})$$
(PARTIAL)

**Fig. 2** Application of our rules to an example query. The applied rules are stated at each line to illustrate the way the query changed from line to line.

| | Query | Rule |
|---|---|---|
| **1**: | `desc (a[[`*var* X → `c]] : (2, a, 6, 7)): (1, z, 2, 3)` | |
| | | DESC1 |
| **2**: | `z[[ (`**desc** `(a[[`*var* X → `c]] : (2, a, 6, 7)): (2, a, 6, 7))]] : (1, z, 2, 3)` | |
| | | DESC2 |
| **3**: | `z[[ (a[[`*var* X → `c]] : (2, a, 6, 7))]] : (1, z, 2, 3)` | |
| | | PARTIAL |
| **4**: | `z[ (a[[`*var* X → `c]] : (2, a, 6, 7))] : (1, z, 2, 3)` | |
| | | PARTIAL |
| **5**: | `z[` **or**`[ a[`*var* X → `c, d], a[b,` *var* X → `c, d] ]` | |

The example query **1** binds elements of type `c` occurring immediately beneath any `a` to the variable `X`. The `c`s may occur at any position within the list of `a`'s children. Therefore, the query contains incompleteness in depth (the **`desc`** construct) and in breath (the double brackets).

The first step in our example applies the DESC1 rule (responsible for descendent expansion) in order to remove the depth incompleteness. This rule expands a **`desc`** *t* which is queried against a node with label `l` by replacing it with a partial subterm specification for the node with label `l`. Please note that because of the subterm specification of the `l` labelled node being partial after applying this rule, the $R_2G_2$ graph node $s'$ might be any of the states connected (horizontally) to $c$. Applying the rule results in an "inwards moved" **`desc`**, illustrated in line **2** of Figure 3.2.

Type checking now reveals that a second iteration of this rule is not necessary, because elements with label `a` are only allowed to occur immediately beneath `z` labelled nodes. This means that the **`desc`** in step **2** is not needed anymore and could be removed. This is achieved by applying rule DESC2 and results in query **3**.

Now the depth incompleteness has completely been removed, but the query still contains incompleteness in breath. In Xcerpt incompleteness in breath means partial subterm specification. Using the list of "subpaths" provided by the *hptes* function, a partial content model can be expanded to become total by the help of the most general node ★. (The ★s themselves can then in turn be specialized in a possible follow-up step.) However, in general the double brackets or braces may contain a list of the subterms the regarding node must possess, but do not preclude additional subterms in

the data (as double brackets or braces indicate partial query terms). The PARTIAL rule ("partial specification expansion") covers this (using $s_t$ as possible start states of the term $t$ and $e_t$ as its possible end states). Intuitively, it states that any partial query term $t$ with sub-terms $t_1, \dots, t_q$ and content model start state $c$ can be complete to a disjunction of total query terms in the following way: for each path through the content model of $t$ that touches also the given sub-terms $t_1, \dots, t_q$ (in that order), we generate one disjunct where that path is explicitly unfolded. More precisely, the $z_1, \dots, z_{q+1}$ represent one such path through the content model of $t$ that touches, in order, each of the $t_1, \dots, t_q$: the path is partitioned at the $t_1, \dots, t_q$ with $z_1$ being any possible path from the content model start state $c$ to $s_{t_1}$ (the start state of $t_1$), $z_i$ for $1 < i <= q$ the path from the end state of $t_{i-1}$ to the start state of $t_i$, and $z_n$ the path from the end state of $t_q$ to an end state of the schema automaton. Each of the $z_i$s is a sequence of types representing its segment. For each combinations of $z_i$'s a disjunct is generated where the actual $z_i$ is unfolded into the missing siblings (using, as above, the ★ notation for elements restricted only by their type).

In the example query **3** this rule can be applied twice: once for the outermost partial term and once for the inner (with `a` label). The outermost can be simply dropped as there is just one possible path containing an `a` label in the $R_2G_2$ graph of our schema *on the "child level" of* `z` (omitting the superfluous **`or`** which could be rewritten by general normalization rule, cf. Section 5). This results in step **4**, of which the inner partial term can be addressed. Here, however, we have a choice of several paths through the $R_2G_2$ graph containing the required `c` labelled edge. The final result of the rule application to our example query is thus step **5**, which the expanded list of all `a` elements possible under the query's constraints..

The rules VAR and STAR are added merely for convenience in handling ★ in the rewriting process. They have been implicitly applied in steps **4** and **5** and will therefore not be illustrated with examples themselves. With VAR (variable specialization), a variable `X` of type $\tau$ (as might be used in a typed Xcerpt query) can be transformed to a variable binding, where `X` is bound to a (concrete) node ★ of type $\tau$. Here one can also recognize the flexibility of ★: it can represent a node of any type and nevertheless be handled like any other concretely given node. With the rule of star specialization, STAR, we can transform the general ★ to an explicitly labelled term.

To conclude the discussion of the rewriting rules, please note that though we have given only rewriting rules concerning *ordered* subterm specifications, this is no limitation to the approach. On the level of the rewriting rules, the difference between ordered and unordered subterms is just notationally. In each of the above rules, the brackets may be replaced by braces. Therefore, the details of handling unordered subterm specifications are left out.

## 4 Related Work

Rewriting and minimization of queries is as central to Web queries as it has been to queries on relational databases. Often some form of normalization to rewrite undesirable language features into equivalent expressions is employed, e.g., the removal of reverse axis in XPath optimization [14].

On the remaining language, previous work has mostly concentrated on removing depth incompleteness (in form of regular path expressions (short RPEs) or XPath's descendant axis).

For Web queries using regular path expressions (short RPEs), [11] gives a practical algorithm for rewriting RPEs containing wild cards and a closure axis like XPath's **`descendant`**. They employ, as we do in this work, graph schemata and automata for processing such schemata. However, as the queries they consider are only regular path expressions, they can also use an automaton for (each of) the regular path expressions to be rewritten. Our approach is at the same time broader and more focused: Due to the limitation to RPEs they can only consider rewriting of depth incompleteness, whereas we consider also breadth (and briefly order) incompleteness. However, their approach can obtain rewritings in cases where our approach fails or produces undesirably large results.

On XPath containment and minimization, the essential results for our work are positive (polynomial time algorithms exist) if only tree pattern queries (understood as XPath queries with only child and descendant axis and no wild card labels) are considered, for details see [16].

Our approach differs from these works in rewriting not only vertical path expressions (involving only child and descendant) axis, but in considering also breadth (and briefly order) incompleteness. In this aspect, it is more closely related to approaches from area (3) such as [9], where a heuristic optimization technique for XQuery is proposed: Based on the PAT algebra, a number of normalizations, simplification, reordering, and access path equivalences are specified and a deterministic algorithm developed. Though the algorithm does not necessarily return an optimal query plan it is expected and experimentally verified to return a reasonably good one. Our approach could be employed in such a framework assuming a cost model where depth, breadth or order incompleteness is considered more expensive than complete, but under a given schema equivalent queries .

Whereas none of the above discussed approaches considers breadth and order incompleteness in the way we do in this work, some relation regarding such incompleteness to works on using schema information for pruning query processing against XML streams is noticeable. [17] proposes such a use of schema information.

Again, our proposed techniques for removing breadth incompleteness can be exploited in such a scenario. In case the schemata are rather regular, our techniques might even give rise to fixed memory constraints for the streamed processing, cf. [13]. However, the details of such an exploitation are still open.

## 5 Outlook and Conclusion

The previous section concludes the discussion of the equivalences for reducing or introducing (depending on the reading direction) incompleteness in Web queries. These equivalences, however, are only the first step to an automatic optimization of Web queries w.r.t. incompleteness. To be of practical use, they need to be integrated into an (necessarily heuristic, cf. Section 4) optimization algorithm such as [9].

It is worth noting, that elimination of any kind of incompleteness leads to no practically useful heuristic: Eliminating all breadth incompleteness, i.e., rewriting all partial subterm lists in total subterm lists. This is clearly infeasible, if types may occur in many different combinations as siblings of a node, as, e.g., in HTML where most element types may be combined with most other element types. In many cases it is even impossible, as repetition in breadth (i.e., any content model with kleene-stars involved) of a schema gives rise to infinite disjunctive query completions. A practical heuristic needs to implement some cut-off point where this expansion is no longer useful. Similar arguments apply for the elimination of all depth and order incompleteness. Infinite query completions arise with recursive schemata in depth, though completion of depth incompleteness is often more promising, as most practical Web documents and schemata have rather limited nesting depths. Despite these remarks, simple heuristics may be applicable if certain assumptions on the schemata are made such as upper limits on the number of possible parent and sibling types a given schema type may combine with.

The proposed equivalences are a small, though, in our opinion, important part of the optimization rules applicable for Web queries. Combination and integration with other forms of query optimization and rewriting for Web queries has not yet been considered.

If the proposed equivalences are to be employed in an XPath or XQuery context, the rewriting of reverse axes such as **`ancestor`**, cf. [14], in XPath is required as a precondition, since the discussed rules assume forward-only expressions (since these have mostly the same expressiveness as expressions allowing also reverse axes).

We have not yet integrated the discussed equivalences into an optimization algorithm, and thus experimental results on their practical use are still open.

## Conclusion

In this paper, we present a novel look on incompleteness in Web queries expressed, e.g., in Xcerpt or XQuery. Incompleteness is one of the distinguishing features of Web query languages compared to languages such as SQL. However, incomplete queries are often considerably more expensive to evaluate than complete queries. Moreover, manually eliminating incompleteness robs Web query languages of one of the most used and most convenient features, the ability to specify data of interest without considering the context. Therefore, we propose to exploit schema information for *automatic* rewriting of Web queries containing incompleteness where applicable.

We propose a set of equivalences for rewriting graph-shaped Web queries on graph-shaped semi-structure data that allows the introduction or removal of all three forms of incompleteness (though order incompleteness is only briefly discussed). These equivalences form the foundation of a flexible treatment of incomplete Web queries beyond just the treatment of depth incompleteness as in previous work.

Ongoing work is on the development of an heuristic optimization algorithm that chooses when to apply these equivalences and to experimentally verify the practical improvement to query evaluation that can be obtained through these equivalences.

*Acknowledgments.* This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

## References


1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
2. S. Berger. An Automaton Model for Xcerpt Type Checking and XML Schema Validation. REWERSE-TR-2007-01, Inst. for Computer Science, Univ. of Munich, Germany, 2007.
3. S. Berger and F. Bry. Towards Static Type Checking of Web Query Language. In *Proc. Workshop über Grundlagen von Datenbanken (GvD)*. 2005.
4. S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive Typing Rules for Xcerpt. In *PPSWR*, LNCS 3703. Springer, 2005.
5. P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/X-Query: a fast XQuery Processor powered by a Relational Engine. In *SIGMOD*, 2006.
6. A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. HKUST-TCSC-2001-0, Hongkong Univ. of Science & Tech., 2001.
7. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *ICLP*, LNCS 2401. 2002.
8. D. Chamberlin, P. Frankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. Working draft, W3C, 2005.
9. D. Che, K. Aberer, and T. Özsu. Query Optimization in XML Structured-document Databases. *The VLDB Journal*, 15(3):263–289, 2006.
10. H. Common, M. Dauchet, R. Gilleron, , F. J. D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata/`, 1999.
11. M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, 1998.
12. C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. In *Proc. ACM Symp. on Principles of Database Sys. (PODS)*, 2005.
13. D. Olteanu. SPEX: Streamed and Progressive Evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering*, 2007.
14. 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, 2002.
15. S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.
16. T. Schwentick. XPath Query Containment. *SIGMOD Record*, 33(1):101–109, 2004.
17. H. Su, E. A. Rundensteiner, and M. Mani. Semantic Query Optimization for XQuery over XML Streams. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 2005.