Schemas for safe and efficient XML processing

Dario Colazzo

Schemas for safe and efficient XML processing

Habilitation à Diriger des Recherches

(Spécialité Informatique)

Université Paris-Sud 11

présentée et soutenue publiquement le 8 septembre 2011

par

Dario Colazzo

Composition du jury

Rapporteurs :	Angela BONIFATI	Université des Sciences et Technologies de Lille
	Denis LUGIEZ	Université de Provence
	Victor VIANU	UC San Diego
	ooo	
Examinateurs :	Nicole BIDOIT-TOLLU	Université de Paris-Sud 11
	Alain DENISE	Université de Paris-Sud 11
	Philippe RIGAUX	Conservatoire National des Arts et Métiers

Laboratoire de Recherche en Informatique – CNRS UMR 8623
Équipe associée INRIA Saclay – Île-de-France – , Université Paris Sud 11,
bât. 650, F-91405, Orsay Cedex France
http://www.lri.fr

Mis en page avec la classe thloria.

INDEX

ABSTRACT

This Habilitation Thesis manuscript presents main results obtained during my research activities carried out as Assistant Professor at Université Paris-Sud since 2005. At the beginning of this period the *eXtensible Markup Language* (XML) was already recognized as the *de facto* standard for representing semi-structured data. Also, XML acquired an important role in data exchange and data integration systems. During this period my research interests were in the intersection of database and programming languages, and focused on the use of type-based static analysis to ensure safe and efficient XML processing. In more detail, I focused on three research directions: i) type-projection for efficient main-memory XML processing, ii) checking correctness of schema-to-schema XML mappings in the context of data integration systems, and iii) efficient algorithms to check XML schema inclusion (a crucial property to type-check XML queries and updates). This Habilitation Thesis presents motivations, techniques and results obtained along these lines of research.

RÉSUMÉ

Ce manuscrit d'Habilitation à Diriger des Recherches présente des résultats que j'ai obtenus dans le cadre d'activités de recherche menées depuis 2005 en tant que Maître de Conférences à l'Université Paris-Sud XI. Au début de cette période XML (*eXtensible Markup Language*) était déjà reconnus comme le standard pour la représentation de données semi structurées. En même temps, XML c'est aussi affirmé comme format de representation dans le contexte de l'intégration et l'échange de données. Pendant cette période mes intérêts de recherche se sont situés à la confluence des langages des bases de données et langages de programmation, et se sont focalisé sur l'utilisation des systèmes de types pour assurer la sureté et optimisation des programmes manipulant les données XML. Plus en détails, je me suis principalement intéressé à trois axes de recherche: i) optimisation de requêtes et mise à jours XML via la projection de données, ii) vérification de la correction des *mappings* entre deux schémas XML, iii) algorithmes efficaces pour la vérification d'inclusion entre schémas XML (une propriété qui est à la base des systèmes de types pour requêtes et mises à jour XML). Ce manuscrit d'Habilitation à Diriger des Recherches est consacré à ces trois axes de recherche, et présente le contexte, les motivations et résultats obtenus pour chacun des axes.

Sommaire

CHAPTER 1
INTRODUCTION

The last decade has seen the rapid expansion of the *eXtensible Markup Language* in many application fields. Born as the successor of SGML, XML soon became the natural way of representing data with loose structure; furthermore, its great flexibility made it a universal data representation format, and allowed the use of XML as a convenient medium for exchanging data between different applications. Finally, XML also acquired an important role in data exchange and data integration systems.

To support the diffusion of XML, several tools for transforming, querying, manipulating, and modeling XML data have been defined. In particular, the *World Wide Web Consortium* (W3C) introduced XQuery [XQub] as the standard query language for XML data.

Since its introduction, XQuery has attracted a big deal of attention, from both industrial and academic worlds. The research community has been involved in many research directions such as complexity and expressivity analysis of XQuery fragments, techniques to optimize query evaluation, security, static analysis, and data integration.

Many of the techniques devised in these research lines involve or rely on the use of schema information. The W3C defined two schema languages for XML data: DTD [BPC+06] and XML Schema [TBMM04, BM04]. While the first one comes from the document community as a language to constrain the format of SGML documents, the second one is closer to the spirit of relational/object DBMSs and general programming languages type systems. In terms of regular tree grammars, DTDs correspond to local tree grammars, while XML Schemas correspond to the more expressive class of single type tree grammars [MLMK05].

Schemas allow one to represent both the structure and the constraints of the data being processed. Schemas, hence, play a key role in many XML database tasks, such as query optimization [ABS99, FYL+09, FS98, BCF+02, BC09, BCCN06], data integration and exchange [BZH11, FB08, BCF+02, CS09], and development of safe database applications [Che08a, CGMS06, DZLM04, PV, SV02, AMN+01, PV00].

This thesis reports on main results I obtained during my research activity in the last years, after I obtained my PhD degree. During this period I focused on three research directions, respectively related to the three above mentioned tasks, and all relying on schema-based analysis, namely: i) efficient in-memory XML processing, ii) checking correctness of schema-to-schema XML mappings, and ii) efficient algorithms to check XML schema inclusion (a crucial operation to type-check XML queries and updates).

The following three sections describe the context and results concerning these three research lines; each section indicates the chapter presenting described results. A fourth section contains

details about other works. This introductory chapter ends with a list of publications described in each chapter of the thesis.

1 Main-memory XML processing

XQuery has been originally defined has a language to query XML databases, but quickly became the main tool to process XML data simply stored in files or generated by a stream, as happens in many kinds of applications (e.g., streaming, information integration, full-text search). Most of these applications do not necessarily need complex functionalities of traditional DBMSs, like for instance those for the management of transactions, and secondary storage indexes. To meet this kind of needs, many light-weight XQuery processors have been devised in recent years, like Galax [gal], Saxon [saxb], QizX [qiz], and eXist [exi], just to mention some of the main ones (a richer list can be found in [xqua]). These systems usually provide full compliance with respect to the W3C specifications, and process data in a main-memory fashion: data to be manipulated are entirely loaded in main memory before being processed.

Main-memory XQuery engines, often rely on smart internal XML representation and indexes, built at loading time and still kept in main-memory, in order to ensure efficient XML navigation, the basic operation behind any kind of XML processing. However, main-memory engines suffer of main-memory size limitations, making impossible the processing of large documents.

One of the main techniques that have been devised to overcome such limitation is *XML projection*. This consists of pruning out at loading time parts of the document that are not needed by a query; the resulting document is called a *projection* of the original document, and if used for query evaluation, instead of the original document, it preserves query result. Since in many contexts queries are likely to involve a small percentage of the original document, XML projection enables main-memory systems to query very large documents even in the presence of a limited amount of main memory.

It is worth mentioning that main-memory size limitations pose problems also for systems that can not be classified as main-memory. This is the case for Monet-DB, one of the fastest XML query engines available today. Its efficiency is due, in particular, to the stair-case operation [GvKT03] it adopts in order to minimize the amount of intermediate results while evaluating XPath expressions. MonetDB is rather a disk-based system, since it uses the disk as secondary storage system, thus being able to process very large documents. However it uses as much main-memory as possible to answer a query efficiently and performs its own page management by mapping memory pages to the disk and reading them back when needed. Therefore for such a query engine, speed is directly proportional to the amount of available memory: the more memory is available, the less swapping occurs between pages on disk and pages in main memory. Experiments have shown that XML projection can entail sensible improvements in terms of execution time [BCCN11].

In joint works with Veronique Benzaken, Nicole Bidoit, and Giuseppe Castagna, we investigated two schema-based techniques to project XML documents:

- The first one is based on a key observation coming from my previous research activities [Col04, CGMS04] in the context of type-checking systems for XQuery: during type-checking, besides type information about the query result, type information about nodes that are needed to compute the query result is inferred as well. Starting from this observation, we worked on the definition of new type analysis techniques for XQuery allowing to infer, from a query and an input schema, a *type-projector*, consisting of a quite precise over approximation of the set of types of nodes that are strictly needed for query execution. This information is then used at loading time to prune out nodes whose type is not in the type-projector.

- The second XML projection technique has been devised in order to optimize XML updates expressed by means of the XQuery Update Facility language [xup]. Dealing with updates

required to face two main problems: i) devising a technique to propagate to the original document all the updates performed on the projected document, ii) devising a new notion of type-projector enabling an efficient propagation process.

Results about XML projection for queries have been published in [BCCN06]. Results about XML projection for updates have been published in [BCMS09b, BCMS09a, BBC+11b], and have been obtained in the context of the two PhD projects of Amine Bahazizi and Marina Sahakian, both co-supervised with Nicole Bidoit. Also, alternative techniques for type-projector have been investigated in Federico Ulliana's Master Thesis [Ull], that I supervised.

The above results are presented in Chapter 2.

2 Detection of corrupted XML mappings

As already outlined, XML is an universal data format that can be used to represent the vast majority of data sources, from strongly structured data (e.g., relational data) to semistructured or even unstructured data. This property made XML a natural medium for integrating heterogeneous data sources.

One of the most important problems in data integration systems (both centralized and decentralized) is the *maintenance* of *mappings*. Mappings are dependencies among schemas, that are used during query answering for reformulating queries or, as in data exchange systems [AL05a], for generating canonical solutions. Since a mapping m from $\mathscr{S}_i$ to $\mathscr{S}_j$ exploits the structural properties of both schemas $\mathscr{S}_i$ and $\mathscr{S}_j$, a sudden change in one of the schemas, let's say $\mathscr{S}_j$, may corrupt the mapping m, so that its mapping rules are no longer true. Mapping corruption has a deep impact on query answering, and essentially prevents the system from generating (useful) query results.

In a joint work with Carlo Sartiani, I dedicated a substantial part of my recent research activities to the problem of XML mapping maintenance. Interestingly enough, obtained results involved a binary relation over types based on XML projection, and according to which two types are related if each instance of the first one is a projection of an instance of the second one.

Main results obtained in this line of research are described below.

- A mapping maintenance technique based on the following observation: unlike schema-to-schema XML functions or transformations, an XML mapping does not output instances of the target schema, but, rather, *projections* of target schema instances [HIMT03]. This is because, in the data integration setting, it is often the case that some target elements do not have corresponding source elements. As a consequence, in order to check correctness of a mapping, the projection relation among a type, inferred for the mapping output, and the target schema is checked.

- A characterization of the type projection relation in terms of a notion of *type simulation*, allowing for a better understanding of the properties of type projection itself, and, in particular, of its relationship with subtyping.

- A type inference technique able to infer, quite precisely, the result type of an XQuery mapping specification. This type system is an extension of the one proposed in my PhD Thesis [Col04]. The inferred type for the mapping is used in a type-projection comparison against the target schema, as described in the first point.

- The proof of NP-completeness of the problem of checking the type projection relation, and the definition of an algorithm which is polynomial in most practical cases. Exhaustive experimental evaluation of an implementation of the proposed framework (type inference system plus a checker for the type projection relation) confirmed its effectiveness.

The above results have been published in [CS05b, CS06, CS09], in particular the journal paper [CS09] collects all of them, and are presented in Chapter 3.

3 Efficient XML subtype checking

XML schemas are an essential tool for the robustness of applications that involve XML data manipulation and transformation. To solve any static analysis problem that involves such types one must first be able to reason about their inclusion and equivalence.

XML schema languages are designed to describe ordered data, but they usually offer some support to deal with cases where the order among some elements is not constrained. Also, XML schema languages usually offer counting operators, enabling the specification of the the minimum and maximum number of times a value of given type can repeat in a sequence.

As shown in in [GMN07], the addition of interleaving (also called *shuffle*) and counting operators to standard regular expressions raises the complexity of inclusion checking from PSPACE (or EXPTIME, for Extended DTDs) to EXPSPACE. These are completeness results.

In a joint work with Giorgio Ghelli, Luca Pardini, and Carlo Sartiani, we worked towards the characterization and study of classes of regular expressions (REs in the following) with interleaving and counting, and for which the inclusion relation can be checked in polynomial time. Such subclasses can be used either to design a new schema language, or to design adaptive algorithms, that use the PTIME algorithm whenever is possible, and resort to the full algorithm when needed. To this aim, it is important that (i) the subclass covers large classes of XML types used in practice, (ii) it is easy to verify whether a schema belongs to the subclass.

This research activity involved several steps:

- A first step was dedicated to the characterization of a class of REs based on the following two restrictions. Each expression is *conflict-free* (or *single occurrence*) meaning that no symbol appears twice, and counting is only applied to symbols or to disjunctions of symbols. These restrictions are severe, but, as shown in [BNdB04] and [BNST06], are actually met by the vast majority of the schemas that are used in practice.[1] We designed a class of logical constraints and proved that the semantics of conflict-free REs can be exactly captured by these constraints. This allowed to rephrase sub-typing as constraint implication, and paved the way to a quadratic algorithm for checking inclusion over conflict-free regulars expressions.

- A second step concerned the relaxation of symmetry of the previous technique, by allowing the subtype to be any type. For this case we provided a quadratic algorithm for inclusion checking. This technique is still based on constraints.

- A third step originated from the observation that in many cases types compared for inclusion checking share a similar structure. We provided a new algorithm that is linear-time for types featuring some similarity properties that can be detected in constant time. When these properties are not met the algorithm reverts to the above quadratic approach for those subparts of the initial types for which similarity is not detected.

Results described in the above first step were first presented in [GCS07] and then in a journal paper [CGS09b]. Concerning the second step, results were presented in [CGS09a] but an extended version is available [CGS11a]. Finally results in the third step were presented in [CGPS09].

Results obtained in the second and third steps are presented in Chapter 4; the technique described in second step includes the symmetric case described in the first step [GCS07, CGS09b].

[1] "More that 99% of the REs occurring in practical schemas", according to [BNST06]

4 Other works

Some the results previously described have been also presented in a recent tutorial given at ICDE'11 [CGS11b], in collaboration with Giorgio Ghelli and Carlo Sartian. An overview on the problem of updating XML data and schemas has been proposed in a book chapter [CGM+10], in collaboration with Giovanna Guerrini, Marco Mesiti, Barbara Oliboni, and Emmanuel Waller.

In collaboration with Nicole Bidoit, I worked on techniques to express DTD-like schemas with references by means of the hybrid modal logic [Bla00]. Based on this coding, we then devised a tableaux system to check that integrity constraints expressed by means of the hybrid modal logic are consistent wrt a schema [BC07].

The constraint-based encoding of conflict-free XML schemas with interleaving and counting, that are presented in Chapter 4, has been at the basis of an *almost* linear method to check membership of XML trees into a class of Extended DTDs with interleaving and counting. The resulting algorithm has time complexity which is linear in the product of the input size with the maximal depth of all the content models in the schema. This works has been presented in [GCS08], and has been carried out in collaboration with Giorgio Ghelli, Luca Pardini and Carlo Sartiani.

In a joint work with Michele Bugliesi, Silvia Crafa and Damiano Macedonio [BCCM09] we studied a type-based theory of DCA (Discretionary Access Control) models for a process calculus that extends the pi-calculus with groups [CGG00].

List of publications described in this thesis, grouped by chapter

Chapter 1, works described in the above Other Works section

[CGS11b] D. Colazzo, G. Ghelli and C. Sartiani. *Schemas for Safe and Efficient XML Processing*. IEEE International Conference on Data Engineering (ICDE), 2011.

[CGM+10] D. Colazzo, G. Guerrini, M. Mesiti, B. Oliboni, and E. Waller. *Document and Schema XML Updates*. In Changqing Li and Tok Wang Ling, editors. *Advanced Applications and Structures in XML Processing: Label Stream, Semantics Utilization and Data Query Technololgies*, IDEA Group, 2010.

[BC07] N. Bidoit and D. Colazzo. *Testing XML constraint satisfiability.* Electronic Notes in Theoretical Computer Science. Volume 174(6) : 45-61, 2007

[BCCM09] M. Bugliesi, D. Colazzo, S. Crafa and D. Macedonio. *A Type System for Discretionary Access Control*. Mathematical Structures in Computer Science (MSCS). Volume 19(4) : 839-875, 2009.

[GCS08] G. Ghelli, D. Colazzo and C. Sartiani. *Linear Time Membership for a Class of XML Types with Interleaving and Counting.* ACM Conference on Information and Knowledge Management (CIKM), pages 389-398, 2008.

Chapter 2: Type-based projection for efficient XML processing.

[BBC+11b] A. Baazizi, N. Bidoit, D. Colazzo, N. Malla and M. Sahakyan. *Projection for XML Update Optimization*. 14th International Conference on Extending Database Technology (EDBT), 2011.

[BCMS09b] N. Bidoit, D. Colazzo, N. Malla and M. Sahakyan. *Projection-based optimization for XML updates.* International Workshop on Schema Languages for XML (X-Schemas), 2009. Also appeared in BDA'09.

[BCCN06] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. *Type-Based XML Projection.* International Conference on Very Large Data Bases (VLDB), 2006.

Chapter 3: Projection-based detection of corrupted XML schema mappings.

[CS09] D. Colazzo and C. Sartiani. *Detection of Corrupted Schema Mappings in XML Data Integration Systems.* ACM Transaction on Internet Technology (TOIT). Volume 9(4), paper 14, 53 pages, 2009.

[CS06] D. Colazzo and C. Sartiani. *An efficient algorithm for XML type projection.* ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), 2006.

[CS05b] D. Colazzo and C. Sartiani. *Mapping Maintenance in XML P2P Databases.* 10th International Symposium on Database Programming Languages (DBPL). LNCS 3774, pages 74-89, 2005.

Chapter 4: Efficient XML subtype checking.

[CGS09a] D. Colazzo, G. Ghelli, L. Pardini and C. Sartiani. *Linear Inclusion for XML Regular Expression Types.* ACM Conference on Information and Knowledge Management (CIKM), pages 137-146, 2009.

[CGS09b] D. Colazzo, G. Ghelli and C. Sartiani. *Efficient Inclusion for a Class of XML Types with Interleaving and Counting.* Information Systems. Volume 34(7) : 643-656, 2009. Also appeared in International Symposium on Database Programming Languages (DBPL), LNCS 4797, 2007.

[CGPS09] D. Colazzo, G. Ghelli, L. Pardini and C. Sartiani. *Linear Inclusion for XML Regular Expression Types.* ACM Conference on Information and Knowledge Management (CIKM), pages 137-146, 2009.

Chapter 5: Work in progress and perspectives.

[CS11] D. Colazzo and C. Sartiani. *Precision and Complexity of XQuery Type Inference.* ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), 2011.

[BBC11a] M.A. Baazizi, N. Bidoit, and D. Colazzo. *Efficient Encoding of Temporal XML Documents.* International Symposium on Temporal Representation and Reasoning (TIME), 2011.

[BCU10a] N. Bidoit, D.Colazzo and F. Ulliana. *Detecting XML Query-Update Independence.* Bases de Donnés Avancées, 2010.

Sommaire

CHAPTER 2

TYPE-BASED PROJECTION FOR EFFICIENT XML PROCESSING

This chapter presents results about type-based projection techniques, both for XML queries and updates. The chapter is composed of four main sections. Section 1 introduces some basic notation used in the presentation of both techniques, respectively presented in Section 2 and 3. Finally, Section 4 is dedicated to conclusions.

1 Preliminaries

This section is dedicated to basic notations, which will be used in the whole chapter. We will focus on notation for the data model, schemas and related notions, used in following examples and formalizations. We assume the reader to be familiar with the XQuery query and update languages.

Note that for the sake of uniform presentation, in this chapter we use a notation which may sometimes differ from that of presented articles [BCCN06] and [BCMS09b, BCMS09a, BBC+11b].

Data Model

An instance of the XQuery data model is a forest f generated by the following grammar:

$$\begin{array}{lllll} \textbf{Forests} & f & ::= & () \mid f,f \mid t \\ \textbf{Trees} & t & ::= & s_{\mathbf{i}} \mid l_{\mathbf{i}}[f] \end{array}$$

A forest f is an ordered sequence of labelled ordered *trees* (ranged over by t); $()$ denotes the empty forest.

An XML document is represented by a tree t. Nodes are labelled by *element tags* (ranged over by l) while, without loss of generality, we consider only leaves that are text nodes (that is, strings, ranged over by s) or empty trees (that is, elements that label the empty forest).

Each node in a forest f or tree t has a unique *identifier* (ranged over by $\mathbf{i}$). Node identifiers are needed to define several basic notions such as validation and query semantics (see [BCCN06] for full definitions).

Henceforth we will consider only well-formed forests:

Définition 2.1 (Good formation) A forest is *well formed* if every identifier $\mathbf{i}$ occurs in it at most once. Given a well-formed forest f and an identifier $\mathbf{i}$ occurring in it, we denote by $f@\mathbf{i}$ the unique subtree t of f such that $t = s_{\mathbf{i}}$ or $t = l_{\mathbf{i}}[f']$. The set of identifiers of a forest f is then defined as $Ids(f) = \{\mathbf{i} \mid \exists\, t.\ f@\mathbf{i} = t\}$.

We define a complete partial order $\preceq$ on forests (and thus on trees) by relating a forest with the forests obtained either by adding or by deleting subforests:

Définition 2.2 (Projection ($\preceq$)) Given two forests f and f' we say that f' is a *projection* of f, noted as $f' \preceq f$, if f' is obtained from f by deleting some of its subtrees.

DTDs and validation

Following [LMM00] we define a DTD as a *local tree grammar*, namely a pair (r,E) where r is a distinguished *tag name* and E is a set of productions (or *edges*) of the form $\{a_1 = R_1, \ldots, a_n = R_n\}$ such that

1. r is in $\{a_1, \ldots, a_n\}$ (it denotes the root element type).
2. tag names a_i's are pairwise distinct;
3. each R_i is a regular expression over symbols $\{a_1, \ldots, a_n\} \cup \{String\}$, where *String* is a special symbol denoting the type of string values.

We write $Sym(R)$ for the set of all symbols used in R ($Sym(R)$ may include *String*) and $Sym(E)$ for the set of tag names defined in E (that is, $\{a_1 \ldots a_n\}$). We also say that R is a regular expression over (r,E), if R is a regular expression over $Sym(E) \cup \{String\}$. We will use a, b, c, d to indicate tag names. We use Greek letters to range over sets of tag names (in particular we use π to stress that it is a *type projector* (Def. 2.6) and κ and τ to stress that the set is used as a context or as a type, respectively. When speaking of DTDs we will often identify them with their set of edges E, leaving the root r as implicit.

Définition 2.3 (Root id) Given a tree t, if $t = s_{\mathbf{i}}$ or $t = l_{\mathbf{i}}[f]$ then we define $RootId(t) = \mathbf{i}$.

Below we consider the root type of a tree defines as follows: if $t = a_{\mathbf{i}}[f]$ then $r\text{-}type(t) = a$ else $r\text{-}type(t) = String$.

Définition 2.4 (Valid Trees) A tree t is *valid* with respect to a DTD (r,E), noted as $t \in (r,\ E)$ if the following conditions hold:

1. $t = r_{\mathbf{i}}[f]$ (the root element is typed by the root tag in the DTD);
2. for each $\mathbf{i}$ in $Ids(t)$, if $t@\mathbf{i} = b_{\mathbf{i}}[t_1,\ldots,t_n]$, then $b = R \in E$ and the word $r\text{-}type(t_1),\ldots,r\text{-}type(t_n))$ is generated by R.

2 Type-based projection for XML query optimization

XML data projection (or pruning) is an optimization technique adopted in the context of main-memory XML query-engines. The main idea underlying XML projection is quite simple and productive at once: given a query Q over a document t, the subtrees of t not necessary to evaluate Q are pruned out, thus obtaining a smaller document t'. Then Q is executed over t', hence avoiding to allocate and process nodes that will never be reached by navigational specifications in Q.

As shown in [MS03, BCL+05], XML navigation specifications expressed in queries tend to be very selective, especially in terms of document structure. Therefore, projection is likely to yield significant improvements both in terms of execution time and in terms of memory usage (for main-memory XML query engines, very large documents can not be queried without pruning).

In this chapter we present a projection technique based on the use of the schemas of queried documents. In order to better highlight its features we first overview main schema-less approaches.

Related works: untyped apporaches

Marian and Siméon [MS03] propose that the actual data-needs of a query Q (that is, the part of data that is necessary to the execution of the query) is determined by statically extracting all paths in Q. These paths are then applied to t at load time, in a SAX-event based fashion, in order to prune unneeded parts of data. The technique is powerful since: (i) it applies to most of XQuery *core*, (ii) it can be applied to a set of queries over the same document, and (iii) it does not require any *a priori* knowledge of the structure of t. However, this technique suffers some limitations. First, the document loader-pruner is not able to manage *backward axes* nor path expressions with predicates which, especially the latter, can contain precious information to optimise pruning. Also, as a consequence of (iii), the technique does not behave efficiently in terms of loading time and pruning precision (hence, memory allocation) when `//` occurs in paths. Indeed, when `//` is present in a projection path, the pruning process requires to visit all descendants of a node in order to decide whether the node contains a useful descendant. What is worst is that pruning time tends to be quite high and it drastically increases (together with memory consumption) when the number of `//` augments in the pruning path-set. As a matter of facts, in this technique pruning corresponds to computing a further query, whose time and memory usage may be comparable to those required to compute the original query. In particular, in this technique every occurrence of `//` may yield a full exploration of the tree. Therefore, pruning execution overhead and its high memory footprint may jeopardize the gains obtained by using the pruned document. Finally, the precision of pruning drastically degrades (even vanishes) for queries containing the XPath expressions `descendant::node`$[cond]$, which are very useful and used in practice.

Bressan *et al.* [BCL+05] introduce a different and quite precise XML pruning technique for a subset of XQuery FLWR expressions. The technique is based on the *a priori* knowledge of a data-guide for t. The document t is first matched against an abstract representation of Q. Pruning is then performed at run time, it is very precise, and, thanks to the use of some indexes over the data-guide, it ensures good improvements in terms of query execution time. However, the technique is one-query oriented, in the sense that it cannot be applied to multiple queries, it does not handle XPath predicates, and cannot handle backward axes (recall that the encodings of [OMFB02] are defined for XPath, and no extension to XQuery-like languages is known). Also, the approach requires the construction and management of the data-guide and of adequate indexes.

Our type-based approach

Our schema-based technique combines the advantages of the previously mentioned works while relaxing their limitations. Unlike [MS03, BCL+05], our approach accounts for backward axes, performs a fine-grained analysis of predicates, allows (unlike [BCL+05]) for dealing with bunches of queries, and (unlike [MS03]) cannot be jeopardised by pruning overhead. Our solution provides comparable or greater precision than the other approaches, while it requires always negligible or no pruning overhead. Moreover, contrary to [MS03, BCL+05], our approach is formally proved to be *sound* (projection preserves query semantics) and, furthermore, we can also prove it to be *complete* (it produces the best possible type-driven projection) for a substantial class of queries and DTDs.

We have devised our framework in three main steps:

1. In the first step, we consider a simplified version of XPath, we dub XPath$^\ell$, which includes only upward/downward axes and unnested disjunctive predicates. We define for XPath$^\ell$ a static analysis that determines a set of type names, a *type projector*, that is then used to prune the document(s). One of the particular features of this approach is that our pruning algorithm is characterised by a constant (and low) memory consumption and by an execution time linear in the size of the document to prune. More precisely, a pruning based on type projectors is equivalent to a single bufferless one-pass traversal of the parsed document (it simply discards elements not generated by any of the names in the projector). So if embedded in query processors, pruning can be executed during parsing and/or validation and brings no overhead, while if used as an external tool it requires a time always smaller than or equal to the time used to parse the queried document. Soundness and (partial) completeness results for the static analysis are stated.

2. The second step consists of extending the analysis to the whole XPath (more precisely, to XPath 1.0). This is done by associating to each XPath query Q a XPath$^\ell$ query P which soundly approximates Q, in the sense that the projector inferred for P is also a sound projector for Q.

3. The final step is to extend the approach to XQuery (hence, to XPath 2.0). This is obtained by defining a path extraction algorithm as done in [MS03]. Our path extraction algorithm improves in several aspects the one of [MS03]. It also computes the XPath$^\ell$ approximation of the extracted paths so that the static analysis of the first step can be directly applied to them.

Type projectors

Given a tree t valid with respect to a DTD (r,E), we can use subsets of $Sym(E)$ to project that tree. Essentially, only nodes that are associated with names in the projecting subset of $Sym(E)$ are kept in the projection. Of course not every subset of $Sym(E)$ can be used to project a tree, since we want to delete whole subtrees (not nodes in the middle of a tree), thus if we discard some tag symbol, we must also discard all the tags it generates. In order to define formally this notion we need to define the reachability relation $\Rightarrow_E$, that we introduce below together with several other definitions that we use later on in this chapter.

Définition 2.5 (Forward Reachability) Given a DTD (r,E) and $a \in Sym(E)$, we write $a \Rightarrow_E b \iff a \to r \in E$ and $b \in Sym(r)$. We use $\Rightarrow_E^+$ and $\Rightarrow_E^*$ to denote respectively the transitive closure, and the transitive and reflexive closure of $\Rightarrow_E$.

Strings of names are called *chains* denoted by c, c_i, c',... In particular we use $Chains_{(r,E)}(a)$ to denote the set of all chains rooted at a, defined as $\{a\,a_1 \dots a_n \mid a \Rightarrow_E a_1 \Rightarrow_E \dots \Rightarrow_E a_n, n \geq 0\}$. With a little abuse of notation, use $Sym(c)$ to denote the set of all tag symbols occurring in a chain c.

Définition 2.6 (Type-Projectors) Given a DTD (r,E), a (possibly empty) set $\pi \subseteq Sym(E)$ is a *type projector* for (r,E) if and only if there exists $C \subseteq Chains_{(r,E)}(r)$ such that

$$\pi = \bigcup_{c \in C} Names(c)$$

A type projector is thus a union of sets of names, where each of these sets is generated (i.e. reached) by a sequence of productions starting from the root of the DTD. A type projector can be used to prune a valid tree as follows:

Définition 2.7 (Type Driven Projections) Let π be a type projector for (r,E) and t a tree such that $t \in (r,E)$. The π-projection of t, noted as $t\backslash\pi$, is defined according to the following rules:

$$\begin{array}{lcll} l_i[f]\backslash\pi & = & l_i[f\backslash\pi] & \text{if } l \in \pi \\ l_i[f]\backslash\pi & = & () & \text{if } l \notin \pi \\ s_i\backslash\pi & = & s_i & \text{if } String \in \pi \\ s_i\backslash\pi & = & () & \text{if } String \notin \pi \\ (f,f')\backslash\pi & = & (f\backslash\pi),(f'\backslash\pi) & \end{array}$$

In words, projection erases (by replacing it by an empty forest) every node that corresponds to a name not in π.

Lemma 2.1 *Let π be a type projector for (r,E). Then for every tree $t \in (r,E)$ it holds $(t\backslash\pi) \preceq t$.*

Type-projectors for XPath$^\ell$, overview

This section has a twofold purpose: first it provides examples illustrating how a type-projector is associated to an XPath$^\ell$ query, then it present the formal inference system together with main properties.

XPath$^\ell$ is defined by the following grammar:

$$\begin{array}{lcl} \mathit{Path} & ::= & \mathit{Step} \mid \mathit{Step}[\mathit{Cond}] \mid \mathit{Path}/\mathit{Path} \\ \mathit{Step} & ::= & \mathit{Axis}::\mathit{Test} \\ \mathit{Axis} & ::= & \texttt{self} \mid \texttt{child} \mid \texttt{descendant} \\ & \mid & \texttt{parent} \mid \texttt{ancestor} \mid \texttt{ancestor-or-self} \\ & \mid & \texttt{descendant-or-self} \\ \mathit{Test} & ::= & \mathit{tag} \mid \texttt{node} \mid \texttt{text} \end{array}$$

where *tag* is a meta-variable ranging over element tags, and

$$\begin{array}{lcl} \mathit{Cond} & ::= & \mathit{SPath} \mid \mathit{Cond}\ \texttt{or}\ \mathit{Cond} \\ \mathit{SPath} & ::= & \mathit{Step} \mid \mathit{SPath}/\mathit{SPath} \mid /\mathit{SPath} \end{array}$$

Consider the following DTD $(report, E)$ describing medical reports, borrowed from [CFF+07], and where E consists of the following productions:

```
report = (section*)
section = (title, content)
title =  String
content =  (String | anesthesia | prep | incision | action | observation)*
anesthesia =   String
prep =  ( String | action)*
```

```
incision = ( String  | geography | instrument)*
action = ( String | instrument )*
observation = String
geography = String
instrument = String
```

and the following queries XPath$^{\ell}$ queries:

$$\begin{array}{lcl} Q_1 & = & \texttt{/report/section/title} \\ Q_2 & = & \texttt{/descendant::node/title} \\ Q_3 & = & \texttt{/descendant::node/action/instrument/ancestor::node} \\ Q_4 & = & \texttt{/descendant::node[geography or instrument]} \end{array}$$

Given a valid document D, a projection of D which is sound for Q_1 must contain report, section, and title nodes (of course, if these nodes are in D). Each title node that may appear in D belongs to the query result, as a consequence each descendant of such node must belong to the projection as well. So, a type-projector entailing a sound projection for Q_1 is $\tau = \{\texttt{report}, \texttt{section}, \texttt{title}, \texttt{String}\}$. Note that any τ' including τ still is a sound type-projector for Q_1.

For simple queries like Q_1, a sound type-projector can be inferred by using standard static type inference techniques [Col04, CGMS04], as follows. We recall that static type inference for XPath allows to statically determine a superset of the types of nodes *resulted* by a query. So in this case, it would be sufficient to perform type inference for all the prefixes of Q_1 plus the query Q_1/`descendant::node` (needed to include type of nodes for the final result):

$$\begin{array}{lcl} \texttt{/report} & : & \{\texttt{report}\} \\ \texttt{/report/section} & : & \{\texttt{section}\} \\ \texttt{/report/section/title} & : & \{\texttt{title}\} \\ \texttt{/report/section/title/descendant::node} & : & \{\texttt{String}\} \end{array}$$

and consider the union of these inferred types.

This simple and direct approach also works for Q_2, in the sense that it produces a sound projector for this query. Unfortunately, the resulting projector would be useless as it coincides with the set of all DTD types, due to the presence of the prefix `/descendant::node`.

These two examples highlight a strict connection between type-projector inference and traditional type-inference [CGMS06, DFF+10]. At the same time, by means of Q_2, we see that a simple type-inference based approach has serious limitations, as useless type-projectors can be inferred. However, type-inference still remains the basic tool: in order to infer a precise type-projector for a query `/descendant::node`/*Path*, we can proceed as follows. First the type of the prefix `/descendant::node` is inferred, then this type is filtered/refined by retaining only element types that are *productive* for the subsequent path *Path*. The obtained set is a first component of the final projector, which of course includes the projector inferred for *Path* as well.

Deciding whether a type a in the DTD is productive for a relative path *Path* can be done by first inferring the type of *Path*, by assuming that its navigation starts from nodes of type a, and then by checking non-emptiness of the inferred type. For instance, for the previous DTD, the path *action*/*instrument* is productive for *prep*, while it is not for the type *section*.

So, for the query Q_2, the first step selects all the DTD types, and a second step selects only *report* and *section* types, which are productive for the step /*title*. This allows to infer a type-projector coinciding to that of the equivalent query Q_1.

Now, consider the query Q_3. A projector for the *forward* prefix

`/descendant::node/action/instrument`

is inferred according to what just described. This type is {`instrument`}. To complete the inference the last *backward* step has to be taken into account. A naive approach would proceed as follows: a

type for the las step is inferred starting from the type {`instrument`}, which is inferred for the prefix. That is all types which are ancestors of this type in the DTD are considered as the remaining portion for the projector. The problem with this approach is its low precision: here also the type `incision` would be considered in the projector, while a sound projection for Q_3 does not require `incision` nodes. This unneeded type can be filtered out by using the following ingredient: during the type-projector inference for the prefix `/descendant::node/action/instrument` we also keep track of types that are traversed. This contextual information in this case consists of the set

$$\kappa = \{\texttt{report}, \texttt{section}, \texttt{content}, \texttt{action}, \texttt{instrument}\}$$

and during type inference is collected into a set called *context*. This set is then intersected with the type for `ancestor::node`, inferred as previously described, so that the unwanted `incision` type is ruled out.

Again, by means of intersection of intermediate inferred types, a precise projection can be inferred for queries containing conditional steps *Step*[*Cond*]. Consider the query Q_4. If we consider the type inferred for the first step as part of the final projector, then this degenerates to the set of all DTD types. On the other hand, by simply restricting to only types of the first step that are productive for at least one of the condition in [`geography or instrument`], the more precise projector

$$\tau = \{\texttt{report}, \texttt{section}, \texttt{content}, \texttt{action}, \texttt{instrument}, \texttt{incision}\}$$

is obtained.

Type-projector inference for XPath$^\ell$.

In this section we present type-projector inference rules implementing the above described approaches. According to what we have seen, type-projector rules rely on type inference rules, so we first focus on the presentation of these last ones.

Type inference rules for XPath$^\ell$.

Type inference rules for XPath$^\ell$ are collected in Figure 2.1. These rules prove judgements of the form

$$(\tau_c, \kappa_c) \vdash_E \mathit{Path} : (\tau_r, \kappa_r)$$

meaning that given a DTD E, starting from the names in τ_c and the current context κ_c, the path *Path* generates the names τ_r in an updated context κ_r.

Type inference rules make use of two functions $\mathbf{A}_E(\tau, \mathit{Axis})$ and $\mathbf{T}_E(\tau, \mathit{Test})$, mimic *Axis* navigation and *Test* filtering on a set τ of E types. For instance, for the previously used DTD we have :

$$\begin{array}{lcl}
\mathbf{A}_E(\{\mathit{prep}\}, \texttt{child}) & = & \{\texttt{String}, \texttt{iaction}\} \\
\mathbf{A}_E(\{\mathit{prep}\}, \texttt{descendant}) & = & \{\texttt{String}, \texttt{action}, \texttt{instrument}\} \\
\mathbf{A}_E(\{\mathit{instrument}\}, \texttt{parent}) & = & \{\texttt{incision}, \texttt{action}\} \\
\mathbf{A}_E(\{\mathit{instrument}\}, \texttt{ancestor}) & = & \{\texttt{incision}, \texttt{action}, \texttt{prep}, \texttt{content}, \texttt{section}, \texttt{report}\} \\
 & & \\
\mathbf{T}_E(\{\mathit{String}, \mathit{action}\}, \mathit{action}) & = & \{\texttt{action}\} \\
\mathbf{T}_E(\{\mathit{String}, \mathit{action}\}, \texttt{text}) & = & \{\texttt{String}\} \\
\mathbf{T}_E(\{\mathit{String}, \mathit{action}\}, \texttt{node}) & = & \{\texttt{String}, \texttt{action}\}
\end{array}$$

The two functions can be defined in straightforward way (see [BCCN06] for details) and can be composed in the obvious way for typing a single XPath step *Axis* :: *Test*.

Primitive Single Step

$$\frac{}{\Sigma \vdash_E Axis :: \texttt{node} :\ (\mathbf{A}_E(\Sigma_\tau, Axis)\ ,\ \Sigma_\kappa \cup \mathbf{A}_E(\Sigma_\tau, Axis))}\ Axis \in \{\texttt{self}, \texttt{child}, \texttt{descendant}\}$$

$$\frac{}{\Sigma \vdash_E Axis :: \texttt{node} :\ (\mathbf{A}_E(\Sigma_\tau, Axis)) \cap \Sigma_\kappa\ ,\ \mathbf{A}_E(\Sigma_\kappa, Axis) \cap \Sigma_\kappa)}\ Axis \in \{\texttt{parent}, \texttt{ancestor}\}$$

$$\frac{}{\Sigma \vdash_E \texttt{self} :: Test :\ (\mathbf{T}_E(\Sigma_\tau,\ Test)\ ,\ (\Sigma_\kappa \cap \mathbf{A}_E(\mathbf{T}_E(\Sigma_\tau,\ Test), \texttt{ancestor})) \cup \mathbf{T}_E(\Sigma_\tau,\ Test))}\ Test \neq \texttt{node}$$

$$\frac{\forall a_i \in \Sigma_\tau, P_j \in Cond\ ,\quad (\{a_i\}, \Sigma_\kappa) \vdash_E P_j :\ \Sigma^{ij}}{\Sigma \vdash_E \texttt{self} :: \texttt{node}[Cond] :\ (\tau\ ,\ (\Sigma_\kappa \cap \mathbf{A}_E(\tau, \texttt{ancestor})) \cup \tau)}\ \tau = \{a_i \mid \exists j. \Sigma^{ij}_\tau \neq \varnothing\}$$

Encoded Single Step

$$\frac{\Sigma \vdash_E Axis :: \texttt{node}/\texttt{self} :: Test :\ \Sigma'}{\Sigma \vdash_E Axis :: Test :\ \Sigma'}\ \begin{array}{c} Test \neq \texttt{node} \\ \wedge \\ Axis \neq \texttt{self} \end{array}$$

$$\frac{\Sigma \vdash_E Axis :: Test/\texttt{self} :: \texttt{node}[Cond] :\ \Sigma'}{\Sigma \vdash_E Axis :: Test[Cond] :\ \Sigma'}\ \begin{array}{c} Test \neq \texttt{node} \\ \vee \\ Axis \neq \texttt{self} \end{array}$$

Composed paths

$$\frac{\Sigma \vdash_E Step :\ \Sigma''\quad \Sigma'' \vdash_E Path :\ \Sigma'}{\Sigma \vdash_E Step/Path :\ \Sigma'}$$

FIGURE 2.1: Inference rules for single step queries

The rules inf Fig. 2.1 are relatively simple to understand. The first two rules implement our main idea: when we follow an axis *Axis*, we compute the type by $\mathbf{A}_E(\Sigma_\tau, Axis)$; if the axis is a downward one, then we add this type to the current context, otherwise if the axis is an upward one, then we intersect it with the current context (both for the type part and for the context part). The rule for `self` :: *Test* is slightly more difficult since it discards from the current set of nodes those that do not satisfy the test: the type is computed by $\mathbf{T}_E(\Sigma_\tau, Test)$, while the context is obtained by erasing all the names that were in there just because they generated one of the discarded nodes; to do it it generates (the type of) all ancestors of the nodes satisfying the test, and intersects them with the current context. These first three rules are enough to type all the paths of the form *Axis* :: *Test* since, as stated by the fifth typing rule, all remaining cases are encoded as *Axis* :: `node`/`self` :: *Test*.

The fourth rule is the most difficult one: recall that *Cond* is a disjunction of *simple* paths; the type τ is obtained by discarding from Σ_τ all (names of) nodes for which *Cond* never holds; thus for each X_i in Σ_τ we compute the type of all the paths in *Cond*, and keep in τ only names for which at least one path may yield a non-empty result; the context then is computed as in the third rule, by discarding from the context all names that generated only names discarded from Σ_τ.

Once more, all the remaining cases of conditional steps are encoded by this one, as stated by the sixth rule. Finally, step composition is dealt as a logical cut.

This system has bee proved to be sound (details can be found in [BCCN11]). In the following $[\![P]\!](t)$ denotes the set of nodes resulted by the evaluation of P on t. Also, as expected, we assume that the type on an element node is the element tag (of course the type of a string node is *String*).

Theorem 2.1 (Soundness of Type Inference) *Let (r,E) be a* DTD*, t a tree such that $t \in E$, and P a path such that $(\{r\},\{r\}) \vdash_E P : (\tau,\kappa)$. If a node is in $[\![P]\!](t)$ then the node type is in τ.*

Besides soundness, the proposed system also enjoys type-completeness for a wide class of cases, described next. This has been proved to hold for $*$-guarded, non-recursive, and parent-unambiguous DTDs):

- $*$-guarded DTD are those using regular expressions where union $|$ can only be used inside a $*$-type;
- parent-unambiguous DTDs are such that for each pair of label types $l1$ and $l2$, if the $l2$ is used in the content model definition of $l1$, then $l2$ can not be used in the content model of an $l3$ which can be reached by $l1$ by following DTD productions.

Non-recursivity and $*$-guardedness are properties enjoyed by a large number of commonly used DTDs. As an example, the reader can consider the DTDs of the XML Query Use Cases [use]: among the ten DTDs defined in the Use Cases, seven are both non-recursive and $*$-guarded, one is only $*$-guarded, one is only non-recursive, and just one does not satisfy either property. Furthermore, other studies [Cho02, BNdB04] provide a detailed classification of real world DTDs showing that non-$*$-guarded unions are quite infrequent.

Concerning the parent-unambiguous property, although DTDs satisfying this property are less frequent (five on the ten DTDs in [use]), its absence is in practice not very problematic since, only the presence of the `parent` axis may hinder completeness.

Theorem 2.2 (Completeness of Type Inference) *Under the same assumption of the previous Theorem (Soundness), if (r,E) is $*$-guarded, non-recursive, and parent-unambiguous, then for each $\alpha \in \tau$ there exists $t \in E$ such that a node in $[\![P]\!](t)$ has type α.*

To see why completeness does not hold in general consider the following DTD rooted at s and which is recursive and not $*$-guarded

$$\{c = a \mid b;\ a = a*, String;\ b = String\}$$

and the following two queries $\texttt{self} :: c[\texttt{child} :: a]/\texttt{child} :: b$ and $\texttt{self} :: c/\texttt{child} :: a/\texttt{parent} :: node$. The type inferred for the first query contains a, while the query is always empty. This is due to the non $*$-guarded union $a \mid b$: if we had $(a \mid b)*$ instead, then the query might yield a non-empty result, therefore a must correctly (and completely) be in the query type. The second query shows the reason why completeness does not hold in presence of recursion and backward axes (recursion with only forward axes does not pose any problem for completeness). The type of the second query should be $\{c\}$, but instead the type $\{c,a\}$ is inferred. This is due to the recursion $a = a*,\ldots$: since $a \Rightarrow_E a$, once a is reached it is kept in the inferred type for every backward step.

The techniques developed in my PhD Thesis [CGMS04, Col04] can be adapted to recover completeness for cases like the first query, while a more sophisticated type analysis could solve the problem with the second. It is worth observing, however, that if we relax the $*$-guardedness constraint, and if we keep child and parent axes, plus the `node` test condition, then the problem of inferring an exact set of types is NP-complete. This can be easily shown by using results on XPath satisfiability, widely studied in recent years (see for instance [Hid03, LRWZ04, Mar04, MSV03, BFG08]). The proof follows by considering that:

- The problem of determining whether a path P produce a non empty result for at least one instance of a DTD, is NP-complete for the above described XPath fragment and non-recursive DTDs with unguarded union [BFG08].
- A sound and complete type inference system infers a non empty type if and only if the path produces a non empty result on a schema instance.

The type inference technique we have proposed is polynomial. What motivated us in the research of large fragments for which type-completeness hold was to provide formal evidence of the high precision of the inference systems we devised. If completeness hold for large classes of cases, then for remaining cases the inference is likely to remain highly precise. From a practical point of view, precision of type inference is crucial since it implies precision of type-projector inference (next section), which in turn implies high reduction of memory needed for query evaluation.

Type-projector inference rules for XPath$^{\ell}$.

As already said, type-projector rules strongly relies on type rules reported in the previous section. We have seen that for simple paths $Step_1/\ldots/Step_n$, we may consider as type projector with respect to (r,E) the set $\bigcup_{i=1\ldots n}\tau_i \cup \{r\}$, where for $i = 1\ldots n$:

$$(\{r\},\{r\}) \vdash_E Step_1/\ldots/Step_i : (\tau_i, -)$$

(we use "$-$" as a placeholder for uninteresting parameters). and that this approach is sound but not precise at all: for an expression `descendant` :: `node`/*Path*: the use of the above union yields a set containing τ_1 defined as

$$(\{r\},\{r\}) \vdash_E \texttt{descendant} :: \texttt{node} : (\tau_1, -)$$

that is, all descendants of the root X (no pruning is performed). For a precise type-projector, we have to discard, at least, all names that are descendants of X but that are not ancestors of a node matching *Path*. These are the names $b \in \mathbf{T}_E(\mathbf{A}_E(\{s\},\texttt{descendant}),\texttt{node})$ such that

$$(\{b\},\kappa) \vdash_E \texttt{descendant} :: \texttt{node}/Path : (\varnothing, -)$$

for some appropriate context κ. A similar reasoning applies to `ancestor`.

Type-projector rules performing such a selection are reported in Figure 2.2. These rules prove prove judgements of the form

$$(\tau',\kappa') \Vdash_E Path : \pi$$

meaning that that given a DTD E, starting from the names in τ_c and the current context κ_c, the type-projector π is inferred for *Path*.

In order to infer a precise type-projector for paths formed by a single step, if the step has no condition (first rule), then the type inference of the previous section is enough; otherwise (second rule) the step is transformed into a complex path (a simple trick to avoid the definition of several rules). Thanks to the third rule the type inference can work on just one node at a time, and thanks to the fourth and fifth rules, it just analyses paths whose components have one of the following three forms: (i) `self`::*Test*, (ii) `self`::`node`[*Cond*], or (iii) *Axis*::`node`. These three cases are handled by the "Primitive Rules" of Figure 2.2: The first rule handles the case (i) simply by collecting the current context. The second rule handles the case (ii), by collecting besides the context also all the parts that are necessary to compute the condition (which in the rule is expanded in its more general form); the case (iii) is handled by the last three rules which are nothing but slight variations of the same rule according to the particular axis taken into account: each rule infers the type τ obtained by discarding from the type $\{a_1,\ldots,a_n\}$ of the step, all names that are useless for the rest of the path, and then uses this τ to continue the inference of the projector.

Base and induction

$$\frac{\Sigma \vdash_E \mathit{Step} : (\tau, \kappa)}{\Sigma \Vdash_E \mathit{Step} : \tau \cup \kappa} \qquad \frac{\Sigma \Vdash_E \mathit{Step}[\mathit{Cond}]/\texttt{self} :: \texttt{node} : \pi}{\Sigma \Vdash_E \mathit{Step}[\mathit{Cond}] : \pi}$$

$$\frac{(\{a_1\}, \kappa) \Vdash_E P : \pi_1 \quad \cdots \quad (\{a_n\}, \kappa) \Vdash_E P : \pi_n}{(\{a_1, \ldots, a_n\}, \kappa) \Vdash_E P : \bigcup_{i=1..n} \pi_i} \text{ if no other rule applies}$$

Encoded Rules

$$\frac{\Sigma \Vdash_E \mathit{Axis} :: \texttt{node}/\texttt{self} :: \mathit{Test}/P : \pi}{\Sigma \Vdash_E \mathit{Axis} :: \mathit{Test}/P : \pi} \; \begin{array}{c}\mathit{Test} \neq \texttt{node} \\ \wedge \\ \mathit{Axis} \neq \texttt{self}\end{array}$$

$$\frac{\Sigma \Vdash_E \mathit{Axis} :: \mathit{Test}/\texttt{self} :: \texttt{node}[\mathit{Cond}]/P : \pi}{\Sigma \Vdash_E \mathit{Axis} :: \mathit{Test}[\mathit{Cond}]/P : \pi} \; \begin{array}{c}\mathit{Test} \neq \texttt{node} \\ \vee \\ \mathit{Axis} \neq \texttt{self}\end{array}$$

Primitive Rules

$$\frac{(\{b\}, \kappa) \vdash_E \texttt{self} :: \mathit{Test} : \Sigma \quad \Sigma \Vdash_E P : \pi}{(\{b\}, \kappa) \Vdash_E \texttt{self} :: \mathit{Test}/P : \{b\} \cup \pi}$$

$$\frac{(\{b\}, \kappa) \vdash_E \texttt{self} :: \texttt{node}[P_1 \texttt{or} \ldots \texttt{or} P_n] : \Sigma \quad \Sigma \Vdash_E P : \pi \quad \Sigma \Vdash_E P_i : \pi_i}{(\{b\}, \kappa) \Vdash_E \texttt{self} :: \texttt{node}[P_1 \texttt{or} \ldots \texttt{or} P_n]/P : \{b\} \cup \pi \cup \pi_1 \cup \cdots \cup \pi_n} \; n \geq 1$$

$$\frac{\begin{array}{l}(\{b\}, \kappa) \vdash_E \mathit{Axis} :: \texttt{node} : (\{a_1, \ldots, a_n\}, \kappa') \\ (\{a_i\}, \kappa') \vdash_E P : \Sigma^i \quad (\tau, \kappa') \Vdash_E P : \pi'\end{array}}{(\{b\}, \kappa) \Vdash_E \mathit{Axis} :: \texttt{node}/P : \tau \cup \pi'} \; \begin{array}{l}\mathit{Axis} \in \{\texttt{parent}, \texttt{child}\} \\ \tau = \{a_i \mid \Sigma^i_\tau \neq \varnothing\} \cup \{b\}\end{array}$$

$$\frac{\begin{array}{l}(\{b\}, \kappa) \vdash_E \texttt{desc} :: \texttt{node} : (\{a_1, \ldots, a_n\}, \kappa') \\ (\{a_i\}, \kappa') \vdash_E \texttt{desc} :: \texttt{node}/P : \Sigma^i \quad (\tau, \kappa') \Vdash_E \texttt{child} :: \texttt{node}/P : \pi'\end{array}}{(\{b\}, \kappa) \Vdash_E \texttt{desc} :: \texttt{node}/P : \tau \cup \pi'} \; \tau = \{a_i \mid \Sigma^i_\tau \neq \varnothing\} \cup \{b\}$$

$$\frac{\begin{array}{l}(\{b\}, \kappa) \vdash_E \texttt{ancs} :: \texttt{node} : (\{a_1, \ldots, a_n\}, \kappa') \\ (\{a_i\}, \kappa') \vdash_E \texttt{ancs} :: \texttt{node}/P : \Sigma^i \quad (\tau, \kappa') \Vdash_E \texttt{parent} :: \texttt{node}/P : \pi'\end{array}}{(\{b\}, \kappa) \Vdash_E \texttt{ancs} :: \texttt{node}/P : \tau \cup \pi'} \; \tau = \{a_i \mid \Sigma^i_\tau \neq \varnothing\} \cup \{b\}$$

FIGURE 2.2: Projectors inference rules (where `ancs` and `desc` are shorthands for `ancestor` and `descendant`)

The main theorem enjoyed by type-projector inference is soundness.

Theorem 2.3 (Soundness of projector inference) *Let (r,E) be a* DTD *and P a path. If $(\{s\},\{s\}) \Vdash_E P : \pi$, then π is a type projector for (r,E) and for every $t \in E$:*

$$[\![P]\!](t) = [\![P]\!](t\backslash\pi)$$

Besides soundness, the proposed system also enjoys type-completeness for a wide class of cases we describe next. By type-completeness for an inferred type-projector τ we mean that if discard a type from τ, then the resulting projection does not preserve query semantics.

Type-completeness of projector inference requires completeness of the type inference (Theorem 2.4), and the following properties: an XPath query Q is *strongly-specified* if (i) its predicates do not use backward axes, (ii) along Q and along each path in the predicates of Q there are no two consecutive (possibly conditional) steps whose *Test* part is `node`, and (iii) each predicate in Q contains at most one path and this does not terminate by a step whose *Test* is `node`. For instance, among the following queries, only the first two are strongly-specified.

1. `descendant` :: `node`/`self` :: a/`ancestor` :: `node`
2. `descendant` :: `node`[`child` :: b]/`self` :: a/`parent` :: `node`
3. `descendant` :: `node`/`ancestor` :: `node`/`self` :: a
4. `descendant` :: `node`[`child` :: b/`child` :: `node`]/`self` :: a
4. `child` :: a[`descendant` :: `node`/`parent` :: b]/`child` :: c

Once more, we are in presence of a very common class of queries: for instance, almost all paths in the XMark and XPathMark benchmarks are strongly specified.

Theorem 2.4 (Completeness of projector inference) *Let (r,E) be a $*$-guarded, non-recursive, and parent-unambiguous* DTD*, and P a strongly-specified path. If $(\{s\},\{s\}) \Vdash_E P : \pi$, then there exists $t \in E$ such that for each $a \in \pi$, if $\pi' = \pi \setminus (\{a\} \cup \boldsymbol{A}_E(\{a\},\texttt{descendant}))$, then*

$$[\![P]\!](t\backslash\pi) \neq [\![P]\!](t\backslash\pi')$$

Handling full XPath 1.0 and FLWR expressions

XPath 1.0 has many features not considered in the XPath$^\ell$ fragment, notably:

- Horizontal axes (e.g., following-sibling, following).
- Predicates making use of complex conditions (involving conjunction, negation, functions, etc.)

We could deal with the missing XPath features by adding specific inference rules. Instead we opt to use an approximation of missing mechanisms in terms of mechanisms featured by XPath$^\ell$. As shown by experiments we conducted, this results in a good compromise between simplicity and effectiveness.

Concerning missing axes, the approximation is performed by two logical rewriting passes. In the first pass we rewrite preceding and following axes as specified in the W3C specifications [xquc]. Namely, we substitute each step *Axis* :: *Test* with *Axis* $\in$ {`preceding`, `following`} by the following equivalent path `ancestor-or-self` :: `node`/(*Axis*`-sibling`) :: `node`/`descendant-or-self` :: *Test*

The second pass is the one which introduces the approximation since it replaces all steps of the form *Axis* :: *Test* with *Axis* $\in$ {`preceding-sibling`, `following-sibling`} by the path expression `parent`::`node`/`child`::*Test*.

Clearly, the static analysis of the approximation yields a less precise projection than the one we could obtain by working directly on the original query. However, we still achieve good precision of pruning in practice as we will next, when discussion about experiments. For instance, by applying the above rewriting to XPathMark queries Q9 and Q11, we were able to prune a document down to 7.5% of its original size.

Concerning missing conditions in predicates, we opt to rewrite every predicate *Exp* expressible in XPath to a simple condition *Cond* such that *Cond* is a sound approximation of *Exp* with respect to data needs: the projection determined for *Cond* preserves the semantics for *Exp*. In other words, if we take a generic XPath query *Q* and approximate all its predicates to infer a projector π, then the execution of (the original) *Q* on a given document or on the document pruned by π yield the same result. This rewriting, together with the treatment of missing axes, allows us to deal with a large subset of XQuery and XPath queries, covering those in XPathMark [Fra05] and XMark [SWK+02] benchmarks.

Let us outline the rewriting by an example. Consider the predicate

`[position()>1 and parent::node/`*book/author*`="`*Dante*`" and `*year*`>1313]`

In our system this predicate is approximated by

`[ self::node or parent::node/`*book/author*`or`*year*` ]`

Essentially, given a predicate *Exp* we obtain a condition *Cond* that soundly approximates it by retaining the disjunction of all structural conditions (like `parent::node/`*book/author* and *year* in the previous example), plus either `descendant-or-self::node` or `self::node` if some non-structural condition is present (for instance, `position()>1`). The choice between `self::node` and `descendant-or-self::node` depends on the functions and operators used in the condition: for instance functions like `position` or `count` require `self::node` since their execution requires only the root nodes; instead a function such as `string` needs the whole tree. Formal definitions can be found in [BCCN06].

Type-projector inference for generic XQuery FLWR XQuery expression *q* is performed by a two-steps process:

1. A set of paths XPath$^{\ell}$ $\{Path_1, \ldots Path_n\}$ is extracted from the query *q*, according to a path-extraction mechanisms resembling to that of [MS03]. During the extraction, conditions are approximated as previously explained.

2. For each paths $Path_i$ a projector τ_i is inferred according to rules for XPath$^{\ell}$. For he whole *q*, the projector is $\cup_i \tau_i$ (projectors are closed under union) is inferred.

Experiments

We gauged and validated our approach by testing it both on the XPathMark [Fra05] and on the XMark [SWK+02] benchmarks. Extensive test results can be found in [BCCN06], where we used Galax as a query engine. Other tests were performed in a full version [BCCN11] by using Saxon and Monet-DB as well.

Conducted tests confirmed expected results: thanks to the handling of backward axes and of predicates the precision of our pruning is in general noticeably higher than for current approaches; the pruning time is linear in the size of the queried document and has a very low memory footprint; the time of the static analysis is always negligible (lower than half a second) even for complex queries and DTDs. But benchmarks also brought unexpected (and pleasant) results. In particular, they showed that type-based pruning brings benefits that go beyond those of the reduced size of the pruned document: by excluding a whole set of data structures (those whose type names are not included in the type projector), the pruning may drastically reduce the resources that must be allocated at run-time by the query processor. For instance, our benchmarks show that for several

XMark and XPathMark queries our pruning yields a document whose size is two thirds of the size of the original document, but the query can then be processed using three times less memory than when processed on the original document.

As an aside, it is worth observing that the presented technique relies on the definition of a new type system for XPath able to handle backward axes, which constitutes a contribution on its own.

For what concerns the overhead of the optimisation, tests confirmed that it is always negligible, both in memory and time consumption: the only noticeable overhead is pruning time, which is linear in the size of the pruned document, but can be embedded in document parsing and/or validation (e.g., for 60MB documents computing the projector took around 0.5s while pruning and saving the pruned document to disk was always below 10s). These results were confirmed by further experiments on large DTDs (e.g. XHTML) and long XPath expressions (twenty steps or so).

3 Projection for XML updates

XML projection, as described in the previous section, cannot be applied directly for updating XML documents, simply because given an update u over a document t, and a strict projection t' of t, we have $u(t) \neq u(t')$; in particular $u(t')$ lacks $u(t)$ subtrees pruned out during the projection of t.

This chapter presents a type-based projection technique for XQuery Update Facility [xup] which overcomes the above problem in the following way. First, new techniques to infer a type-projector π from an update u and an input DTD are provided; the type-projector is used to project a valid input document t so that the resulting projection t' is used for update execution. Second, a streaming algorithm called *Merge* is presented; , the algorithm performs a parallel and synchronous parsing of t and $u(t')$ in order to produce the final result $u(t)$.

For the sake of efficiency, the *Merge* step is designed so that (a) only child position of nodes and the projector π are checked in order to decide whether to output elements of t or of $u(t')$ and (b) no further changes are made on elements after the partial updated document $u(t')$ has been computed: output elements are either elements of the original document t or elements of $u(t')$.

Related Work

The approach here presented introduces substantial novelties wrt the type based approach for queries presented in the previous chapter. As it will be explained next, for updates a three-level projector is adopted, while the projector proposed in Chapter 2 is one level. A three level projector, allows to optimize (minimize) the size of projections. In particular, it allows to avoid keeping in the projection useless text nodes that would be kept with the technique proposed in Chapter 2: this can result into substantial improvements since in many cases large parts of documents consist of textual content.

Other works propose techniques to optimize update execution time by using static analysis in order to detect independence between several update operations, so that query rewriting techniques can be used for logical optimization [GRS07, GRS08, BBFV05, BC09]. The work here presented is definitely orthogonal wrt that line of research, and indeed, the two techniques can be combined in order to increase the efficiency in terms of time.

Some recent works [FCB07, Feg10] addressed the problem of translating an XQuery update expression u into a pure query expression Q_u, with the aim of executing the update u via the query Q_u. The advantages of these approaches are that updates can be executed even if the XQuery engine only deals with queries, and well established query-optimization techniques can be adopted to optimize update execution. A peculiar characteristic of these approaches [FCB07, Feg10] is that the query Q_u needs to select and return all nodes that are not updated, while those which are updated are selected and processed to compute new nodes. As a consequence, using standard projection techniques [BCCN06, MS03] for the query Q_u would lead to no improvement, since the *whole* document would be projected.

It is worth observing that, although not directly, existing projection techniques [BCCN06, MS03] could be used for a single update, provided that the projected document is used only to compute the update pending list, so that this last one can be then propagated to the input document in a streaming fashion. Such approach would require some techniques similar to those here developed in order to: opportunely determine the projection, and make node identity persistent in order to propagate, in the second phase, the calculated update pending list. This approach has two drawbacks. Firstly, it does not allow to use XML querying engines in a straight manner as we propose to do: controlling the two phase evaluation of XML updates would become necessary. Secondly, this approach would perform very inefficiently in the quite frequent case where a bunch of n updates has to be executed, according to a given order, because each update would need to be fully processed one after the other entailing the document to be processed/parsed n times. The approach here presented is different in that it allows to evaluate the n updates by processing the proposed method just once: a global projector can be easily inferred (it is sufficient to consider the union of each update projector); the n updates are evaluated on the global projection wrt the specified order; finally, the updates are propagated on the original document in a single pass, using the *Merge* function. As testified by performed tests (that will be commented next), this results in a much more efficient processing.

Overview

This section is devoted to introducing and illustrating, through examples, the main features of our method. We first focus on the merge process, and then switch to a new kind of type-projector which is required by the merge process itself in order to ensure a safe and efficient propagation of updates to the original document.

Merge, a first example. Let us consider the example in Fig. 2.3 based on the update

`for $x in /doc/a where $x/d return delete $x/b`

Assume that the partial updated document $u(t')$ has been produced by updating the document t', a projection of the original document t leading to t'. In order to produce the final result $u(t)$, we parse and merge the initial document t and the partial updated document $u(t')$.

Before commenting the examples, let us spend some words about the adopted notation. In the figure, each node of the initial document t is adorned with its label (a, b, ...) and with an identifier **i** inside square brackets (1, 1.1, ...). A node in t whose identifier is **i** is next denoted by t@**i**. We assume that the identifier of a node in t carries on information about the node position in t, according to document order.

In the projection t' of t, the identifier of a projected node is preserved, therefore it may not reflect the new position of the node in t' (it is the case, for instance, of the node t'@1.4 in Fig. 2.3.4). In the partial updated document $u(t')$, new identifiers are assigned to inserted or replaced nodes (see next examples).

Let us focus on the merge process. While merging t and $u(t')$, nothing special happens until the a labeled nodes t@1 and $u(t')$@1 are met. Just after, the two nodes examined by *Merge* are: the first child node t@1.1 labelled b of t@1, and the first child node $u(t')$@1.4 labelled d of $u(t')$@1 .

Here, child rank 4 of $u(t')$@1.4 is strictly greater than the child rank 1 of t@1.1. Also, the label b belongs to the projector π, indicating that the node t@1.1 has been projected in t'. Thus, the node t@1.1 is not output (it has been deleted by the update u), the original document t is further parsed.

The next two nodes examined are: t@1.2 labelled c and $u(t')$@1.4 labelled d. Once again, the child rank 4 of $u(t')$@1.4 is strictly greater than the child rank 2 of t@1.2, however this time, the label c does not belong to the projector π (the node t@1.2 was not needed for the partial update and thus not projected in t') and thus the node t@1.2 is output in the final result, the original document t is further parsed. The process will continue parsing t and $u(t')$ until both documents are fully

scanned. Note that, positions of nodes (more precisely child rank) in the initial document play a crucial role in the *Merge* process.

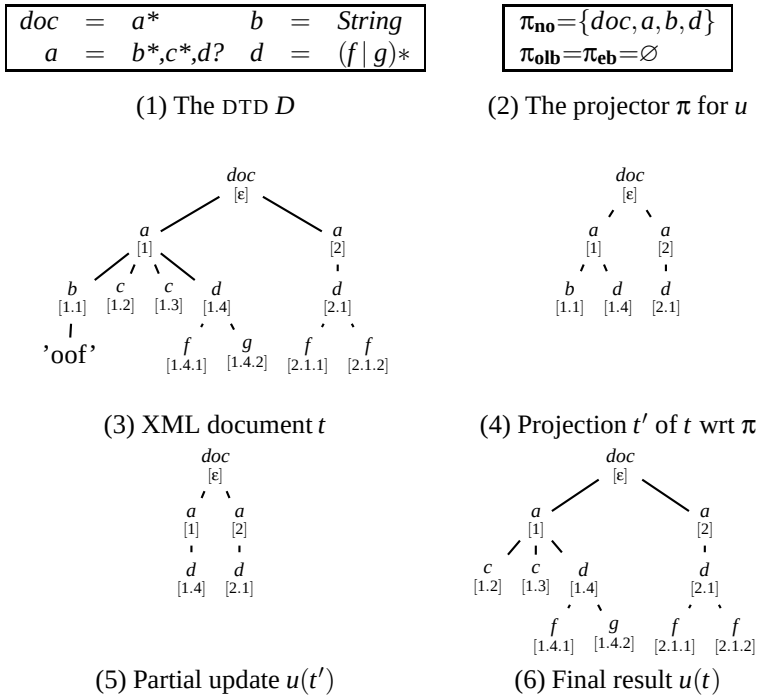

FIGURE 2.3: A motivating example of the Update Scenario

Dealing with insertion Consider the update u_1 specified by

`for $x in /doc/a return insert as last <e>'new'</e> into $x` and the same DTD

D and document t of Fig.2.3.1 and 2.3.4. Intuitively, the path corresponding to data relevant for the update u_1 is $/doc/a$ and the types of nodes traversed by this path are π_1=$\{doc, a\}$. The projection $\pi_1(t)$ of t as well as the partial update $u_1(\pi_1(t))$are illustrated in Figure 2.4. Recall that node identifiers in $\pi_1(t)$ correspond to node identifiers in t, the same holds for unchanged nodes in $u_1(\pi_1(t))$, and that new (inserted or replaced) nodes in $u_1(\pi_1(t))$ are given new identifiers. In Figure 2.4, **i** and **i**′ are new identifiers.

We see now how the merging of the initial document t and the partial result $u_1(\pi_1(t))$ is done in order to produce the final result $u_1(t)$. Once the root nodes of the two documents have been visited, the two nodes examined by *Merge* are: t@1.1 labelled b and the new node $u_1(\pi_1(t))$@**i** labelled e. Here, the new identifier **i** conveys no information about child rank of the new node and even if the projector tells us that the node t@1.1 has been projected out, there is no way to decide whether it has to be output before the inserted node or vice-versa. Recall here the assumption made for *Merge*: information about the update u_1 is not available.

In order to solve this problem, related to insertion, we opt for a new notion of projector, taking into account that for the update u_1 the path $/doc/a$ is the target of an insertion. The projector π_{u_1} will have 2 components: the type doc of category 'node only' and the type a of category 'one level below'. Applying this new projector to a document proceeds as follows: the nodes labelled by types of category 'node only' are projected; the nodes labelled by types of category 'one level below' are projected together with each of their *children*. Descendants of these children are not projected, unless other components of the projector require this projection.

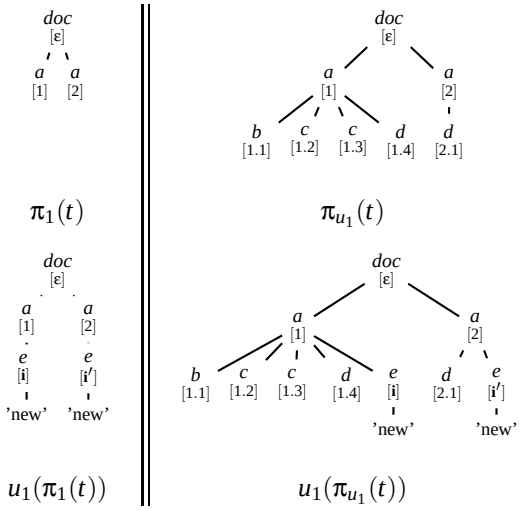

FIGURE 2.4: Dealing with insertion

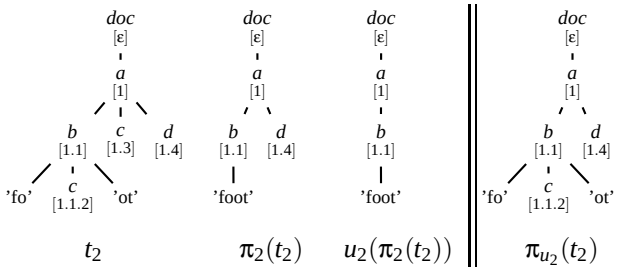

FIGURE 2.5: Dealing with mixed-contents

For our example, applying the projector $\pi_{u_1}=(\pi_{\mathbf{no}},\pi_{\mathbf{olb}})$ with $\pi_{\mathbf{no}}=\{doc\}$ and $\pi_{\mathbf{olb}}=\{a\}$ to the document t leads to the document $\pi_{u_1}(t)$ depicted in the table above together with the partial update $u_1(\pi_{u_1}(t))$. Since now the new nodes are inserted in a projection containing all their siblings, it is easy to check that the documents t and $u_1(\pi_{u_1}(t))$ can be merged in a valid, simple and efficient way.

It is worth observing that our projector avoids unnecessary node projection: the projection of all children of a 'one level below' node is forced, but labels of these children do not take part of the projector.

Dealing with String and mixed-content We now modify the DTD D by redefining the rule for b as $b\rightarrow(String\mid c)*$ and consider the update u_2 specified by `for $x in /doc/a where $x/b/text()='foot' return delete $x/d`. Intuitively, $/doc/a/d$ and $/doc/a/b/text()$ are the paths corresponding to data relevant for the update u_2. The associated types are $\pi_2=\{doc,a,b,String,d\}$. Let us consider the document t_2 and its projection $\pi_2(t_2)$ both given in Figure 2.5. Notice that projecting t_2 wrt π_2 has the side effect to concatenate the two *Strings* 'fo' and 'ot' and consequently, the node $u_2(\pi_2(t_2))@1.4$ labelled d is deleted when the update u_2 is applied on the projected document $\pi_2(t_2)$. Recall the assumption that *Merge* is not supposed to change the elements parsed in t_2 and $u_2(\pi_2(t_2))$ and has only access to the projector. Thus, we cannot expect that merging the initial document t_2 and the partial updated result $u_2(\pi_2(t_2))$ will produce the final updated document.
The problem here is due to mixed-content nodes and solved by modifying the projector in the same way as for insertion. The new projector π_{u_2} generated for the example will have 2 components:

$\pi_{\mathbf{no}}=\{doc,a,d\}$ of category 'node only' and $\pi_{\mathbf{olb}}=\{b\}$ of category 'one level below'.

Dealing with element extraction Consider the DTD D and the update u_3 `for $x in /doc/a return replace $x/b with $x/d`. First, it is clear that `replace` updates have to be treated like insert wrt to the target path `$x/b`: replace is a delete followed by an insert. Second, because the path $/doc/a/d$ is meant to return the element copied at the target node computed by $/doc/b$, the complete subtrees rooted at nodes of type d have to be completely projected. For this update, we propose to generate a projector π_{u_3} composed of three sets of types: $\pi_{\mathbf{no}}=\{doc\}$ of category 'node only', $\pi_{\mathbf{olb}}=\{a\}$ of category 'one level below', and $\pi_{\mathbf{eb}}=\{d\}$ of category 'everything below' (abbreviated '$\forall$ *below*').

Let us explain the behavior of the 3-level type projector wrt the category 'everything below': a node labelled by a type of this category is projected together with its sub-forest. Indeed, applying the projector π_{u_3} on the document t of Fig. 2.3.4 produces almost the whole document with the exception of the String 'oof' which is pruned out.

As already outlined, this third component of the projector ensures higher precision and efficiency wrt [BCCN06]. In particular, it allows avoiding to include in the projector the types of the nodes in the subtree of a '$\forall$ below' node and accelerates the projection process it-self.

Type-projector Inference and then Merge Algorithm

As we have seen, given an update u over an XML document t valid wrt the DTD, our optimization technique relies on 3 steps:

- An update type projector π is inferred from u and the DTD $(r,\ E)$, and t is projected wrt π.
- The update u is evaluated over the projected document $\pi(t)$ producing a partial result $u(\pi(t))$;
- The fully updated document $u(t)$ is built by merging the initial document t and $u(\pi(t))$.

We see in more details each of these steps. As expected, in the following part we assume that the identifier $\mathbf{i}$ in the root node of a tree $a_{\mathbf{i}}[f]$ actually is a position identifier, as depicted in previous examples. Also, differently form the previous chapter, we assume here that textual nodes have no (position) index.

Update type-projector and its inference Our 3-level type projectors are defined as follows.

Définition 2.8 (Type Projector) Given a DTD $(r,\ E)$ over the alphabet Σ, a type projector π is a triple $(\pi_{\mathbf{no}},\pi_{\mathbf{olb}},\pi_{\mathbf{eb}})$ such that (π also denotes $\pi_{\mathbf{no}}\cup\pi_{\mathbf{olb}}\cup\pi_{\mathbf{eb}}$):

1. $\pi\subseteq\Sigma$
2. $\pi_{\mathbf{no}}$, $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$ are pairwise disjoint, and
3. $r\in\pi$ and for each $b\in\pi$ there exists $a\in\pi$ such that $E(a)=R$ and b occurs in R.

The $\pi_{\mathbf{no}}$ (resp. $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$) component of π contains 'node only' types (resp. 'one level below' and '$\forall$ below' types). Notice that condition 3) ensures some closure property wrt the DTD E: label $a\in\pi$ cannot be deconnected from the root label r although it does not need to be connected in all possible manners (see projector π_4 below). Notice that the *String* type itself never belongs to a type projector π: as already explained, a string is projected "indirectly" when its parent node type is of category 'olb' or 'eb'.

1	$Merge(f_i \mid f_u) =$	f_u	if $f_i{=}()$
		otherwise assume $f_i{=}t_i, f'_i$	
2	$t_i \circ Merge(f'_i \mid f_u)$	if $t_i = s$,	
		otherwise assume $label(t_i){=}a$,	
3	$Merge(f'_i \mid f_u)$	if $a{\in}\pi$ and either $f_u{=}()$ or $f_u{=}t_u, f'_u$ with $pos(t_u){>}pos(t_i)$	
4	$TreeMerge(t_i \mid t_u) \circ Merge(f'_i \mid f_u)$	if $a{\in}\pi$, $f_u{=}t_u, f'_u$ and $pos(t_i){=}pos(t_u)$	
5	$t_i \circ Merge(f'_i \mid f_u)$	if $a{\notin}\pi$	

FIGURE 2.6: The function *Merge*

$c.1$	$CMerge(f_i \mid f_u) =$	f_u	if $f_i{=}()t$,
$c.1'$		$()$	if $f_u{=}()$,
		otherwise assume $f_u{=}t_u, f'_u$	
$c.2$	$t_u \circ CMerge(f_i \mid f_u)$	if $t_u{=}s$ or $new(t_u)$,	
		otherwise assume $f_i{=}t_i, f'_i$	
$c.3$	$CMerge(f'_i \mid f_u)$	if $t_i{=}s$ or $label(t_i){=}a$ with $a{\in}\pi$ and $pos(t_u) > pos(t_i)$	
$c.4$	$TreeMerge(t_i \mid t_u) \circ CMerge(f'_i \mid f_u)$	if $a{\in}\pi$, $label(t_i){=}a$, and $pos(t_i){=}pos(t_u)$	
$c.5$	$t_i \circ CMerge(f'_i \mid f_u)$	if $a{\notin}\pi$ and $label(t_i){=}a$	

FIGURE 2.7: The function *CMerge*

Définition 2.9 (Type Projection) Given a DTD DTD (r, E), the type projector $\pi{=}(\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}, \pi_{\mathbf{eb}})$ and the document $t \in (r, E)$, the projection $t\backslash\pi$ is defined according to what follows.

$$
\begin{array}{lcll}
l_{\mathbf{i}}[f]\backslash\pi & = & l_{\mathbf{i}}[f\backslash\pi] & l \in \pi_{\mathbf{no}} \\
l_{\mathbf{i}}[f]\backslash\pi & = & l_{\mathbf{i}}[\Pi^{\mathbf{olb}}_{\pi}(f)] & l \in \pi_{\mathbf{olb}} \\
l_{\mathbf{i}}[f]\backslash\pi & = & l_{\mathbf{i}}[f] & l \in \pi_{\mathbf{eb}}
\end{array}
$$

where

$$
\Pi^{\mathbf{olb}}_{\pi}(()) = () \quad \Pi^{\mathbf{olb}}_{\pi}(s_{\mathbf{i}}) = s_{\mathbf{i}} \quad \Pi^{\mathbf{olb}}_{\pi}(l_{\mathbf{i}}[f], f') = l_{\mathbf{i}}[f\backslash\pi], \Pi^{\mathbf{olb}}_{\pi}(f')
$$

The closure property 3) of definition 2.8 entails that the result of a type projection is a well-formed tree, although it may not conform to the DTD *D*.

Concerning the inference of a the type-projector from an update *u* and an input DTD, this operation relies on path extraction from updates. We do no report here path extraction rules (see [BBC+11b] for details). In a nutshell, the rules extract from an update *u* three kinds of paths, $\boldsymbol{P}_{\mathbf{no}}$, $\boldsymbol{P}_{\mathbf{olb}}$ and $\boldsymbol{P}_{\mathbf{eb}}$, respectively selecting **no**, **olb** and **eb** nodes during update evaluaion. The 3-components $\pi_{\mathbf{no}}$, $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$ of the type-projector are then obtained by using type-projector inference as described in Section 2 starting form, respectively, the extracted paths $\boldsymbol{P}_{\mathbf{no}}$, $\boldsymbol{P}_{\mathbf{olb}}$ and $\boldsymbol{P}_{\mathbf{eb}}$.

The merge phase

The task of *Merge* is to build the result $u(t)$ of the update *u* over *t* starting from the initial tree *t* and the updated partial tree $u(\pi(t))$. The main assumption here is that the input document *t* is a p-store,

implying that node identifiers correspond to node positions in the document. The function *Merge* uses this information, the 3-level projector π and nothing else.

For the purpose of insert and replace operations, it is assumed that the update u generates 'new' (not already used in t) identifiers. Also, when $t = a_{\mathbf{i}}[f]$, in definitions we use $pos(t)$ to indicate the position identifier $\mathbf{i}$ of the root element of t, and $label(t)$ to indicate that label a of this last one. Finally, the predicate $new(t)$ holds when t is a newly inserted tree.

The functions *Merge* and *CMerge* are formalized in Fig. 2.6 and Fig. 2.7. For the sake of simplicity, the update projector π is kept implicit in the specification.

The functions *Merge* and *CMerge* have to be thought of as mechanisms parsing in parallel two forests: f_i belonging to the initial tree t and f_u belonging to the updated partial tree $u(\pi(t))$; parsing synchronization is captured by the fact that the parent nodes of f_i and f_u are assumed to share the same identifier; because of projection and update, f_u contains identifiers belonging to t, besides the new ones due to insert and replace operation.

The two functions differ on the following pre-conditions:

- *Merge* assumes that (†) the parent node n of the forest f_i is of category 'node only' which implies that, because of synchronization, i) none of the top level trees in f_u is of type *String*, ii) root identifiers of top level trees in f_u belong to those of f_i.

- *CMerge* considers that (††) the node n is of category 'one level below' which implies that each node in $roots(f_i)$ has been projected and that $roots(f_u)$ are exactly the top level nodes of f_u that have to be output by *CMerge*.

The function *Merge* proceeds as follows:
Line 2 takes care of the case where the current parsed tree t_i of f_i is a string value. The assumption † entails that it has been pruned out by π. Thus, the string t_i is simply output.
Line 3 deals with the case where the label a of the root of t_i belongs to π (thus a subtree of t_i has been projected) and the t_i root does not occur in f_u (t_i has been projected and then deleted by the update). When f_u is not empty, this latter fact is identified by comparing the identifiers of the currently parsed nodes (which are positions in f_i): $pos(t_u) < pos(t_i)$ indicates that the tree t_i comes after the tree t_u in the forest f_i. Thus t_i is not output.
Line 4 takes care of synchronization on the root nodes of t_u and t_i: these nodes can only differ by their labels because of some renaming. In that case, the tree $TreeMerge(t_i \mid t_u)$ is output.

Finally, line 5 deals with the case where the label a of t_i root does not belong to the projector π implying that t_i has been pruned out. Hence t_i is output.

As said before, the function *CMerge*, specified in Fig. 2.7, is built assuming (††). Parsing f_i and f_u in parallel is thus essentially guided by f_u, as opposed to *Merge*.
Line c.2 deals with the case where the current parsed tree t_u of f_u is either of type *String* or a newly inserted element. This latter case is identified by checking whether the identifier $pos(t_u)$ is new. Hence, the tree t_u is output. The reader may notice that no move on f_i is performed: a simple case analysis shows that synchronization is recovered through other cases.
Line c.3 is similar to line 3, although it should be paid attention to the sub-case where the root of t_i is of type *String*: t_i is then ignored because the *corresponding* string element in f_u (updated or not by u) has, eventually, already been output by a previous application of line c.2.
Lines c.4, c.5 are the dual of lines 4, 5 of the *Merge* definition. The reader should pay attention to line c.5 where, although implicit, the equality $pos(t_i) = pos(t_u)$ holds (as opposed to the case "line 7" of *Merge*): even if $a \notin \pi$, because of (††), the node identified by $pos(t_i) = pos(t_u)$ is in both forests f_i and f_u.

In [BBC+11b], we have proved that our update mechanism based on the 3-level type projection and the merge process preserves update semantics:

Theorem 2.5 *Let u be an update over D and π be the inferred type projector for u. Then for each tree t∈D, we have:* $Merge(t \mid u(\pi(t))) \sim u(t)$.

Above, the equivalence $\sim$ means equality up to node identifiers.

Implementations and Experiments

In order to validate the effectiveness of our method, both projection and merge algorithms have been implemented in Java. The algorithm *Merge* has been implemented by means of two threads, parsing t and $\pi(t)$ respectively. These threads are defined in terms of classes obtained by extending existing SAX parser classes [saxa]. The two threads interact with each other according to the Producer-Consumer pattern.

Experiments

Several tests have been performed using our Java implementation and 7 updates U1 - U7 on XMark documents of growing size. These updates, together with their associated projectors, cover the main update operations made available by XQuery Update Facility (insert, rename, replace and delete). All experiments were performed on a 2.53 Ghz Intel Core 2 Duo machine (2 GB main memory) running Mac OSX 10.6.4 .

The main aim of our tests was to evaluate our projection based technique. We focused on two systems Saxon and QizX, and used the whole set of 7 updates.

Concerning Saxon, tests results showed that our technique succeeds in its primarily purpose: making possible to update very large documents with in-memory systems, in the presence of memory limitations.

Concerning QizX, this system has less severe memory limitations, being able to process rather large documents. However we still have great improvements in terms of memory: with projection, we could update up to 2GB for updates, while without projection the limit was 520 MB. However, for QizX, projection also ensures sensible total execution time reduction. This is in part due to the fact that QizX needs a significant time to build auxiliary indexes at loading time. This improvement also testifies the effectiveness of our design choices at the projector, path extraction, and Merge function level. For the 52MB document, we had the following reductions of execution times, expressed in percentages: U1 (45,4%), U2 (60,3%), U3 (74,3%), U4 (72,2%), U5 (45,2%), U6 (63,6%), U7 (24%). We had similar percentages for documents of other sizes.

A last kind of tests we made concerns the computation of a unique projection for all the updates, executed in the following order: U1, U2, U3, ..., U7. The document has been projected once, then all the updates have been evaluated on the projection, and finally Merge has been executed once to obtain the final document. With Saxon and QizX this took, respectively, 82 and 64 seconds on the 128MB document. For this document, the sum of total times needed to projecting, updating and merging for each single update was much higher, respectively 181 and 194 seconds for Saxon and QizX.

4 Conclusion

In Section 2 we have presented a type-based projection approach for optimizing main-memory XML query processors. Experiments we have conducted showed clear advantages of applying our optimisation technique to query XML documents. Also, our technique improved the state of the art for several aspects: for performances (better pruning, more speedup, less memory consumption), for the analysis techniques (linear pruning time, negligible memory and time consumption), for its

generality (handling of all axes and of predicates), and, last but not least, for the formal foundation it provides (correctness formally proved, limits of the approach formally stated).

The technique we have presented in Section 3 has been the first XQuery update optimization technique based on the use of projection and schema information. One of its main distinctive features is a new notion of projector allowing to strictly minimize the resulting projection, and to efficiently propagate updates from the updated projection to the initial database. Another distinctive feature is that the proposed framework can be exploited without changing any internal part of the query/update engine.

Both presented techniques deal with DTDs. The extension to XML Schema has been considered in [BCCN11], a full version of [BCCN06]. Concerning projection for updates, the extension to XML Schema is subject of current investigations, and results will be integrated in the full version of [BBC+11b].

Sommaire

CHAPTER

3

PROJECTION-BASED DETECTION OF CORRUPTED XML SCHEMA MAPPINGS

Current data integration technologies provide *uniform* and *declarative* interfaces to access data dispersed on *multiple* sources, possibly *heterogeneous* and *autonomous*.

A main task in data integration is the maintenance of schema mappings. A schema mapping from a source schema $\mathscr{S}$ to a target schema $\mathscr{T}$ describes how to translate data conforming to $\mathscr{S}$ into data conforming to $\mathscr{T}$, and it can be used to reformulate queries on $\mathscr{S}$ into queries over $\mathscr{T}$, and *vice versa*, according to the *Global-As-View* (GAV), *Local As View* (LAV), and *Global-And-Local-As-View* (GLAV) paradigms [Ull88, Ull89, FLM99]. Schema mappings are used during query answering for reformulating queries or, as in data exchange systems [AL05b], for generating canonical solutions; schema mappings, hence, allow the system to retrieve data that are semantically similar but described by different schemas.

A schema mapping is *corrupted* when it fails in matching the source or the target schema. The presence of a corrupted mapping can significantly affect query processing, as it may make no more accessible the corresponding remote data source or may produce meaningless query results.

Mapping maintenance is a time-consuming, complex, and expensive activity, and is usually performed by the system/site administrator, who manually inspects schemas and mappings in order to find errors in mappings definitions; as a consequence, quick responses to sudden mapping corruptions are not possible. To aid and accelerate errors detection, several tools and techniques for assisting the administrator in maintaining schema mappings have been described in the recent past (see [MAL+05], for instance). These techniques are usually based on the monitoring of some arbitrary parameter, like, for instance, the “quality” of samples of transformed data instances or transformed

queries, and usually do not offer guarantees about the completeness of the approach.

In more detail, the above depicted techniques have two main drawbacks. First, they are not complete since wrong rules that are not used for reformulating a query or for transforming a sample data instance cannot be discovered. Second, they usually require an interaction with query answering or data transformation algorithms; this implies that these techniques cannot directly check for mapping correctness, but, instead, check for the correctness of a mapping wrt a given reformulation or transformation algorithm. Hence, a significant weakness of these techniques is that they refer to a notion of mapping correctness which is strongly related to the properties of a particular reformulation and/or transformation algorithm.

This chapter describes results presented first [CS05b, CS06] and then in a journal paper [CS09]. The chapter is about a static analysis technique for maintaining schema mappings in XML data integration systems, where mappings are specified by means of XQuery clauses [XQub].

Given a schema mapping m from $\mathscr{S}$ to $\mathscr{T}$, we assume that a schema $\mathscr{S}_m$ describing the structure of the image $m(\mathscr{S})$ is available (through an inference process), and then we compare $\mathscr{S}_m$ wrt the target schema $\mathscr{T}$ according to a *type projection* notion, which generalizes the notion of relational projection, and captures and formalizes the intended semantics of mapping correctness of typical data integration systems [HIMT03]. If this comparison for projection succeeds, we are sure that the mapping rules describe data that are "compatible" with the target schema; moreover, as an important consequence, if the mapping is deemed as correct, then reformulated queries will always be consistent with the target schema.

The above depicted framework requires the existence of an output type $\mathscr{S}_m$ for m; in order to show that this assumption is not restrictive, we provide a quite efficient type inference system, able to infer such upper-bound $\mathscr{S}_m$ at static time, starting from m and its input schema $\mathscr{S}$; we will also show that the inferred schema $\mathscr{S}_m$ is quite close to $m(\mathscr{S})$, thus entailing an high degree of precision in the corruption checking process.

As already mentioned, one of the main strengths of this approach lies on the fact that the combination of type projection and type inference results in a technique that is independent from queries posed against the integrated database, does not rely on query reformulation algorithms/techniques, and it is *complete*, i.e., any incorrect mapping will be detected. As a final remark, the solution proposed in this paper can be used in both traditional and decentralized data integration systems, as well as in data exchange systems ([FKMT05]).

This chapter is structured as follows. Section 1 provides an overview trough examples of the proposed framework. Section 2 describes the notion of mapping correctness, relying on the type projection relation. Then Section 3 discusses decidability, complexity and an efficient algorithm to check type projection. A brief overview of a technique for mapping type inference is given in Section 4, while experiments, related works and conclusions are respectively discussed in Sections 5, 6 and 7.

1 Motivating Scenario

In both examples and formalizations, in this chapter we adopt a notation for types which is different from that of previous chapters. It is based on *regular expression types*, introduced and used in the XDuce [HP03] and XQuery [DFF+10] type languages. Examples and the formal treatment are better handled with this notation.

We motivate our technique by referring to a decentralized data integration scenario, where multiple data sources are connected by means of one-to-one mappings. This scenario is a generalization of centralized approaches, where each data source is mapped into a (single) global schema. For the sake of simplicity, we assume a minimal configuration, comprising two data sources only (p_1 and p_2), so as to focus on mapping correctness rather than on query reformulation or routing issues.

Each data source hosts a bunch of XML data, described by a schema ($\mathscr{S}$ for p_1 and $\mathscr{T}$ for p_2);

these schemas are connected through a schema mapping (in the following we will use the expression "schema mapping" to denote any mapping between types). The mapping can be defined according to the *Global-As-View* (GAV) approach, or to the *Local-As-View* (LAV) approach. Our approach is based on LAV mappings, where the target (local) schema is described in terms of the source (global) schema; nevertheless, this approach applies to GAV mappings too, since, as noted in [Tat04], a LAV mapping from p_1 to p_2 can be interpreted as a GAV mapping from p_2 to p_1.

In our framework, a mapping m from $\mathscr{S}$ to $\mathscr{T}$ is a set of queries that specify how to translate data belonging to $\mathscr{S}$ into data conforming to a projection of $\mathscr{T}$. A mapping, hence, can be regarded as a specification rather than an actual transformation from $\mathscr{S}$ to $\mathscr{T}$, as it is not forced to detail the construction of all target elements.

Mapping queries are expressed in the same query language used for posing general queries: this language, called μXQ, is roughly equivalent to the FLWR core of XQuery. We refer the reader to [Col04, CGMS04, Che08b] for detailed presentations and results about μXQ.

Data integration scenarios like this are usually managed with *mediation* approaches, where queries are reformulated by means of schema mappings and no centralized warehouse is used. The correctness of the query answering process for a given query depends on the properties of the reformulation algorithm as well as on the correctness of the mappings involved in the transformation: indeed, if the mapping fails in matching the target schema, the transformed query will probably fail as well.

The evolution of the integrated database, namely the changes in data source schemas, can dramatically affect the quality of schema mappings and, in particular, lead to the corruption of existing mappings. This will reflect on query answering and on existing optimization techniques for decentralized and centralized systems, such as the mapping composition approach described in [TH04].

The following example illustrates the basic concepts of the query language, provides an intuition of the *projection-based* mapping correctness notion, and shows how mapping incorrectness can reflect on query answering and data transformation.

Consider a decentralized data sharing system for music information. The system allows users to share data about their (legally owned) music files, so to discover information about their preferred songs and singers. Each user publishes, on a voluntary basis, the description of all the songs she is storing on her computer or iPod.

Assume that a user in Cupertino publishes her music database according to the following schema.

```
CupMDB = mySongs[(Song)*]
Song = song[Title, Artist, Album, MyRating]
Title = title[String]
Artist = artist[String]
Album = album[String]
MyRating = myRating[Integer]
```

This schema groups data by song, and, for each song, represents the title, the artist name (a singer or a band), the album title, as well as a personal rating information.

Suppose now that another user in Seattle publishes her database according to the following (different) schema.

```
SeattleMDB = musicDB[Artist*]
Artist = artist[Name, Provenance, Track*]
Name = name[String]
Provenance = provenance[Continent, Country]
Continent = continent[String]
Country = country[String]
Track = track[Title, Year, Genre]
Title = title[String]
Year = year[Integer]
```

```
Genre = genre[String]
```

This schema groups data by artist and, for each artist, details her name and provenance, as well as the list of corresponding tracks.

To make these databases interact together, a proper schema mapping is required, as schemas nest data in very different ways. Assume that the user in Cupertino employs the following mapping (a set of XQuery queries) to map her schema into the Seattle-based schema.

```
SeattleMDB <-
Q1($input):  for $a in $input/song/artist
             return artist[
                       name[$a/data()],
                       for $s in $input/song,
                           $art in $s/artist
                       where $art/data() = $a/data()
                       return track[for $t in $s/title,
                                    return $t]]
Q2($input):  for $db in /mySongs
             return musicDB[Q1($db)]
```

This mapping can be expressed by a single query, by nesting Q_1 into Q_2, however the above presentation offer a modular view of the mapping. When expressing a complex mapping, it is convenient to decompose its XQuery specification into several related queries.

This mapping specifies how data conforming to a fragment of the Cupertino schema (album and myRating elements are discarded) can be transformed into data conforming to a fraction of the Seattle-based schema (for instance, provenance elements are discarded). In other words, the schema mapping takes into account a *projection* of the two schemas. This is a very common situation in data integration systems, as usually only a fraction of semantically related heterogeneous schemas can be reconciled. In particular, as the Seattle user schema does not support album and myRating elements, they must be ignored in the mapping. Furthermore, since the Cupertino schema does not provide information about song genre, corresponding elements are not generated, hence any transformed data instance must be regarded as a projection of a Seattle-compliant data instance. As this mapping is only a specification, the actual transformation can be derived by applying, for instance, a chase-like approach à la Clio.

Assume now that Seattle slightly changes its schema and, in particular, the way artist names are represented: instead of a simple name element, information about artist's first name and second name is inserted into the name element: `Name = name[first[String],second[String]]`.

This change in the target schema makes the Cupertino $\rightarrow$ Seattle mapping incorrect. Indeed, this mapping specifies the construction of simple content name elements, which are now no more allowed in the target schema. It should be observed that chasing the mapping cannot fix this problem, as incorrect mappings generate incorrect transformations.

The incorrectness of the mapping from Cupertino to Seattle has two main consequences. First of all, the actual transformation that we can derive from the mapping fails in creating instances of SeattleMDB from instances of CupMDB: indeed, the transformation still generates simple name elements in the target instance, which can no longer be accepted and validated against the target schema. A second consequence is that, just as for data transformation, even query reformulation fails, in the sense that any query involving name elements is incorrectly reformulated. To illustrate this point, consider the query shown in Figure 3.1 (a). This query, submitted by a user in Cupertino, asks for the titles of all songs published by Burt Bacharach. The query is first executed locally in Cupertino. Then, the system reformulates the query so to match Seattle schema; this reformulation is performed by using standard LAV query rewriting algorithms (one can think of CupMDB as the global schema and SeattleMDB as the local schema) [MH03, TH04][2].

[2]We show a minimal transformed query, obtained by minimizing the original transformed query and by deleting all

```
songs_Bacharach[
for $s in $cupdb//song,
    $t in $s/title,
    $a in $s/artist
let $bb := ‘‘Burt Bacharach’’
where $a = $bb
return song[$t]
```

(a) Cupertino user query.

```
songs_Bacharach[
for $a in $seadb//artist,
    $n in $a/name,
    $t in $a/track
let $bb := ‘‘Burt Bacharach’’
where $n = $bb
return song[$t/title]
```

(b) Transformed Cupertino user query.

FIGURE 3.1: Reformulation of a user query.

At the end of the reformulation process, the reformulated query, shown in Figure 3.1 (b), is then sent to the Seattle site. Unfortunately, the transformed query does not match the new schema of Seattle users, so the Cupertino user cannot gather results from the Seattle site.

This example clearly pinpoints the maintenance issues that arise in data integration systems and, in particular, in those systems reconciling autonomous, web-based data sources. Furthermore, this example highlights that the relationship between a schema mapping and its target schema cannot be modeled through a standard subtyping relation *à la* XQuery/XDuce/CDuce, as this form of subtyping is based on set inclusion. Indeed, the output type of the mapping is not a subtype of the target type, as the target schema prescribes the presence of a genre element, which, instead, is not generated by the mapping. The nature of schema mappings imposes a more flexible and general way of comparing types than a *subtyping-based* comparison. This is the main motivation for the introduction of *type projection*, which captures the Piazza [HIMT03] intuition of mappings as "transformation + projection" (i.e., non-functional transformation). We quote a part of this work:

> At the core, the semantics of mappings can be defined as follows. Given an XML instance, I_S, for the source node S and the mapping to the target T, the mapping defines a subset of an instance, I_T, for the target node. The reason that I_S is a subset of the target instance is that some elements of the target may not exist in the source (e.g., the publisher element in the examples). In fact, it may even be the case that required elements of the target are not present in the source. In relational terms, I_T is a *projection* of some complete instance I'_T of T on a subset of its elements and attributes.

This characterization is, indeed, common to most data integration and data exchange systems, and points out that a schema mapping specifies a *non-functional* transformation from a source schema $\mathscr{S}$ to a target schema $\mathscr{T}$. In the following sections we will provide a formalization of type projection for an even wider class of schema mappings: we will regard a mapping as a set of rules that specify how to transform a source data instance $I_S : \mathscr{S}$ into a *fragment* of one or more data instances I_T conforming to $\mathscr{T}$, the actual transformation from $\mathscr{S}$ to $\mathscr{T}$ being obtained through a chase-like process.

2 Mapping Validity and Correctness

This section describes the notions of mapping validity (no wrong rules wrt the source schema) and mapping correctness (no wrong rules wrt the target schema). These notions are central to our approach, and allow for the definition of an operational checking technique.

The syntax of the type language we adopt is is shown in Figure 3.2. $()$ is the type for the empty sequence value, B denotes the type for base values (without loss of generality, we only consider string base values), types T,U and $T \mid U$ are, respectively, product and union types, and, finally,

redundant subqueries.

Types	T	::=	$()$	*empty sequence*
		|	B	*base type*
		|	$l[T]$	*element type*
		|	T, T	*sequence type*
		|	$T \mid T$	*union type*
		|	$T*$	*repetition type*
Base Type	B	::=	`String`	

FIGURE 3.2: Type language

$T*$ is the type for repetition. Types are unordered, as no global order on XML data dispersed on multiple sources can be established: this aspect significantly increases the hardness of comparing two XML types, as usual heuristics and optimizations based on type ordering cannot be applied in this context.

Since types are unordered, in the following we will consider a product type $T_1, \ldots, T_n$ as identical to all its possible permutations $T_{\pi(1)}, \ldots, T_{\pi(n)}$. Moreover, as our types actually are XDuce unordered types, we also have that $T, ()$ is identical to T, and that $(T, T'), T''$ is identical to $T, (T', T'')$. This conforms to the corresponding laws over the data model.

Furthermore, the type language includes horizontal recursive types (allowed by types $T*$) but does not include vertical recursive types, like, for instance, the one defined by this recursive definition

```
Part=partname[Description, Part*]
```

This is motivated by the fact that most mapping languages are not powerful enough to transform trees with arbitrary depth, whose structure can only be defined by vertical recursive types. Also, it should be observed that many mapping tools like Clio do not support recursive types, as chasing (i.e., the closure of a mapping against a schema) may not terminate on recursive types. For these reasons we believe that discarding vertical recursive types is not restrictive in the study of schema mapping languages.

The semantics of types is standard: we use $[\![T]\!]$ to denote the set of forests described by the type T; the definition is standard [CS09].

Subtyping, which is used next to prove decidability of type projection, is defined via type semantics, as shown below.

Définition 3.1 (Semantic subtyping) Given two types T and U, T is a subtype of U if and only if the semantics of T is contained into the semantics of U:

$$T < U \iff [\![T]\!] \subseteq [\![U]\!]$$

Schema Mappings

In this section we introduce and formalize our notion of schema mappings. In our vision, a schema mapping is the specification of a transformation from a source schema $\mathscr{S}$ to a target schema $\mathscr{T}$.

Définition 3.2 (Mapping) Let $\mathscr{S}$ be a source schema and let $\mathscr{T}$ be a target schema. A schema mapping m from $\mathscr{S}$ to $\mathscr{T}$ is an assertion from $\mathscr{S}$ to $\mathscr{T}$ of the form $(Q, \{q_i\}_i)$, where q_i is a query from $\mathscr{S}$ to $\mathscr{T}$, and Q is an outer query referring each q_i.

The previous definition states that a mapping is composed by a *primary* query (Q), which defines the overall structure of the mapping, and by a set of *secondary* assertions $\{q_i\}_i$, which correlate fragments of $\mathscr{S}$ with fragments of $\mathscr{T}$. The intuition behind this distinction is to provide more modularity to a schema mapping by separating the way assertions are assembled from their specification. For instance, in the previous `SeattleMDB` mapping, Q_2 is the primary query, while Q_1 is a secondary assertion. For the sake of simplicity, we will denote the combination of a primary query and a set of secondary assertions as $Q_{[\{q_i\}_i]}$.

By the previous definition, a schema mapping is a specification of the actual transformations between source data and target data. Indeed, both the primary query and secondary assertions can be incomplete, in the sense that they do not cover all the elements of the target schema. Of course, our mappings can be enriched and modified to represent a complete transformation by using, for instance, a chasing strategy. We prefer to focus on mappings as a specification tool because this notion captures the essence of schema mappings; furthermore, once a mapping has been deemed as correct, the actual transformation can be easily and automatically generated by existing tools.

Correctness

In this section we will introduce the notion of mapping validity and mapping correctness. Validity is characterized by the following definition.

Définition 3.3 (Mapping validity) A mapping $m = (Q, \{q_i\}_i)$ from $\mathscr{S}$ to $\mathscr{T}$ is *valid* if and only if the combination of the primary query and the secondary queries is correct wrt $\mathscr{S}$, in the sense that, for each non-empty subquery q of $Q_{[\{q_i\}_i]}$, there exists a data instance d of $\mathscr{S}$ such that, when evaluated on d, q will return a non-empty result.

Consider the schemas the previous `SeattleMDB` mapping. This mapping is valid wrt the source schema, as each subquery (path expression, in particular) returns no empty results for some valid input.

Assume now that the source schema is modified as follows:

```
...
Song = song[EnglishTitle, Artist, Album, MyRating]
EnglishTitle = englishTitle[String]
...
```

The mapping now becomes invalid wrt the new source schema. Indeed, assertion Q_1 contains a nested query accessing title elements, which are no longer present in the schema.

Mapping validity implies that a valid mapping must be correct wrt the source schema, i.e., it must match the structure of the source schema. We adopt the query correctness notion described in [Col04, CGMS04] and [CS05a]. Mapping validity allows for identifying mappings that are *obsolete*, i.e., that contain rules referring to fragments of the source schema that have been changed or deleted. From now on, we will assume that each mapping is valid, and focus on *mapping correctness*, and therefore on the detection of errors wrt the target schema.

Our notion of mapping correctness is based on the following notion of data projection. Intuitively, f_1 is a projection of f_2, denoted as $f_1 \lesssim f_2$, if there exists a subterm f_3 in f_2 such that f_3 matches f_1; this is very close (up to simulation) to the relational projection, where $r_1 = \pi_A r_2$ if r_1 is equal to the fragment of r_2 obtained by discarding non-A attributes.

The notion of data projection we are going to formalize is essentially the same as the one proposed in previous chapters, the only difference comes from commutativity of forest concatenation.

Définition 3.4 (Value projection) The value projection relation $\lesssim$ is the minimal relation such that:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l} () & \lesssim & f & f_1, f_2 & \lesssim & f_3, f_4 & \text{if } (f_1 \lesssim f_3 \wedge f_2 \lesssim f_4) \\ b_1 & \lesssim & b_2 & f_1 & \lesssim & f_3 & \text{if } \exists f_2 : f_1 \lesssim f_2 \wedge f_2 \lesssim f_3 \\ f & \lesssim & f, () & l[f_1] & \lesssim & l[f_2] & \text{if } f_1 \lesssim f_2 \\ f_1, f_2 & \lesssim & f_2, f_1 & & & & \end{array}$$

Note that in data projection we require structural matching, while exact matching on leaf base values b is not required.

Définition 3.5 (Mapping correctness) A mapping $m = (Q, \{q_i\}_i)$ from $\mathscr{S}$ to $\mathscr{T}$ is *correct* if and only if, for each $\mathscr{S}$ data instance $d_{\mathscr{S}}$, there exists a $\mathscr{T}$ data instance $d_{\mathscr{T}}$ such that $Q_{[\{q_i\}_i]}(d_{\mathscr{S}}) \lesssim d_{\mathscr{T}}$.

The above definitions state that a mapping from $\mathscr{S}$ to $\mathscr{T}$ is correct if and only the result of $Q_{[\{q_i\}_i]}$ on a value conforming to $\mathscr{S}$ is mapped, according to the $\lesssim$ relation, into a value conforming to $\mathscr{T}$. $\lesssim$ is an *injective* simulation relation among values, inspired by the projection operator of the relation data model.

Our notion of mapping correctness relies on the comparison between the semantics of the mapping, i.e., the set of its results when applied to instances of the source schema, and the semantics of the target schema. In this sense, we can say that our notion is *semantic*, as it only depends on the semantics of the source and target schemas, as well as of the mapping. This does not imply that our notion is able to capture the intended semantics of a mapping: this problem, indeed, is AI-complete and cannot be completely solved by an automatic tool. Before concluding this section, some final remarks are needed. The notion of XML projection we adopt is a generalization of that introduced by [MS03], where leaf values are taken into account too. Also, our notion of correctness is independent from the mapping specification language, since it is defined in terms of query (mapping) outputs, hence it is applicable in other data integration scenarios where mappings are inferred by semi-automatic tools; for instance, our approach can be easily applied to mappings described in terms of source-to-target dependencies. More generally, our approach can be applied to any mapping language for which suitable notions of type inference and type projection can be defined.

3 Checking Correctness via Type-Projection

Definitions 3.5 and 3.4 describe our notion of mapping correctness, but they cannot directly be used to check whether a mapping is correct or not. To obtain a constructive definition, we need to switch from values to types.

Définition 3.6 (Type projection) Given two types T_1 and T_2, we say that T_1 is a projection of T_2 ($T_1 \lesssim T_2$) if and only if: $\forall d_1 : T_1 \; \exists d_2 : T_2 .\; d_1 \lesssim d_2$.

As for the value projection relation, the type projection relation is semantic, and states that a type T_1 is a projection of a type T_2 if, for each data instance d_1 conforming to T_1, there exists a data instance d_2 conforming to T_2 such that d_1 is a projection of d_2.

Type projection is quite different from standard subtyping, since it is based on the idea that $T_1 \lesssim T_2$ if T_1 matches a fragment of T_2, while $T_1 < T_2$ implies that T_1 is more specific than T_2.

Consider now our initial example, and the following type:

```
TinyMDB = musicDB[Artist*]
Artist = artist[Name, Provenance]
Name = name[String]
Provenance = provenance[Continent, Country]
Continent = continent[String]
Country = country[String]
```

This type is actually a projection of SeattleMDB, as each TinyMDB instance is a projection of a SeattleMDB instance. However, TinyMDB is not a subtype of SeattleMDB, as TinyMDB instances lack track elements, which are mandatory for SeattleMDB instances. To use type projection in mapping correctness checking, we must correlate type projection and mapping correctness. To this aim, if we assume that for each query we can infer a type containing all the query results, we can use the inferred type to check mapping correctness, as indicated in the following proposition.

Proposition 3.1 (Mapping correctness via type projection) *Given two schemas $\mathscr{S}$ and $\mathscr{T}$, if $m = (Q, \{q_i\}_i)$ is a schema mapping from $\mathscr{S}$ to $\mathscr{T}$, and U an upper-bound for $Q_{[\{q_k\}_k]}$ (for each $f \in [\![\mathscr{S}]\!]$, $Q_{[\{q_k\}_k]}(f) \in [\![U]\!]$), then m is correct if $U \lesssim \mathscr{T}$.*

As we will see next, quite precise query upper bound types can be systematically inferred by means of a type-inference algorithm able to prove judgments of the form $\Gamma \vdash Q : U$, where Γ is an environment containing information about the source schema $\mathscr{S}$, and U is the inferred upper bound type for the mapping Q.

Referring to our initial example, the output type of the `SeattleMDB` mapping is the following:

```
OutputType = musicDB[A*]
A = artist[N, T1*]
N = name[String]
T1 = track[T2]
T2 = title[String]
```

This type is a projection of the target SeattleMDB schema, hence the mapping can be deemed as correct.

The notion of mapping correctness just presented addresses changes in the structure of a schema. As there are several kinds of updates that can be applied to a schema, it is worth to explore the various forms of schema changes, so to understand to what extent our notion is effective.

In its most common interpretation, a schema consists of a type, describing the structure of the instances of the schema, and of a set of constraints over data instances. As a consequence, a schema change may affect the type, the set of constraints, or both.

In our work we focus our attention on the type component of a schema, hence any change in the set of constraints is not supported. This choice is motivated by the fact that, as previously said, we assume that data sources are autonomous, hence it is unlikely that a data source makes constraints externally visible.

We can consider five main kinds of structural changes that can be applied to a schema: the removal of existing type definitions (e.g. the removal of an element type); the change of a datatype inside an element content type (e.g., the switch from `String` to `Int`), the relocation of a fragment of a schema to a new location; the renaming of an element type; and the insertion of new type definitions inside the schema. In the following paragraphs we will explore the applicability of our approach to these kinds of updates on both the source and the target schema.

Source schema Changes in the source schema of a mapping may affect its validity. When a type definition is removed from the source schema, the validity of a mapping is affected only if the definition was used and referred in the mapping (we assume, of course, that the new schema is well-formed). For instance, assume that the definition `Artist = artist[String]` is removed from the source schema (`CupMDB`) seen in a previous example. The mapping becomes invalid, as it tries to access a no longer existing fragment of the schema. The query correctness notion we described in [Col04, CGMS04] can easily capture all errors implied by a type definition removal.

The same considerations apply to the relocation of a fragment of the schema to a new location, and to the renaming of element type definitions. These changes may induce errors in a mapping (remember that we see a mapping as a specification, hence it can be incomplete) only if they affect

the portion of the schema that is visited by the mapping; again, the technique we developed in [CGMS04] is able to identify and notify all errors induced by these schema updates.

Consider the schemas and the mapping of our main example and assume that the source schema is modified as follows.

```
CupMDB = (Song)*
Song = song[Name, Artist, Album, MyRating]
Name = name[String]
Artist = artist[String]
Album = album[String]
MyRating = myRating[Integer]
```

The new source schema contains a name element in place of the old title element (element renaming); furthermore, the collection of song elements has been relocated to the outermost position in the schema (type relocation). Both changes make the mapping no longer valid and induce errors that are easily identified by the approach we described in [CGMS04], since the mapping tries to access schema fragments no longer existing. The switch from a datatype to another one is not directly supported in the approach we described in [CGMS04], as our type system uses a single base type. However, it can be easily seen that an extension to multiple datatypes is trivial and that all corruptions induced by this kind of changes can be identified.

The enrichment of the source schema with new type definitions never alters the validity of a mapping. Indeed, all type definitions accessed by the mapping are still present, so there is no error that a type-checking algorithm can detect. For instance, if we modify the source schema of Example **??** as follows:

```
Song = song[Title, Artist, Album, MyRating, ChartPosition]
ChartPosition = chartPosition[String]
```

all type definitions used by the mapping are still accessible.

Target schema The considerations we did for changes in the source schema apply also to the changes on the target schema. In particular, all corruptions induced by the removal of a type definitions, by the renaming of an element type definition, and by the relocation of a fragment of the schema are identified and notified. Furthermore, value projection and type projection can be easily extended to support multiple datatypes.

As for mapping validity, mapping correctness is not affected by the insertions of new type definitions in the target schema. This is fully reasonable, as our mappings may be incomplete.

As we have seen, our approach is able to capture all errors induced by a *structural* update on the target schema. However, when a structural update is coupled with a modification in the intended semantics of the schema, things change. Referring to our previous example, assume that `SeattleMDB` is modified as follows:

```
Artist = artist[Name, WName, Provenance, Track*]
WName = wName[String]
```

where `WName` models the working name of an artist and `Name` its actual name. Observe that `CupMDB` does not distinguish between the actual and the working name of an artist, so all `CupMDB` name elements are interpreted as working names.

By adding `WName`, we do not violate the correctness notion of the previous section. However, the intended semantics of the schema has now been modified, as `Name` now represents actual names. Such a change makes the mapping no more adequate, as it maps working names from `CupMDB` into `SeattleMDB` actual names.

Any change in the intended semantics of a schema can make the mapping no more adequate. However, checking that a mapping is "semantically" adequate to its source and target schema cannot be automatically performed by a maintenance tool, as this problem is definitely AI-complete. It should be observed that even detecting a change in the intended semantics of the schema of an autonomous data source is, in the general case, not feasible and requires the intervention of a developer.

Dealing with this kind of corruptions, hence, it is definitely not easy. However, our approach can be easily extended to provide useful support to the developer. Indeed, type projection checking can be used to identify the fragments of the target schema that are covered by the mapping; in the same way, validity checking relates the mapping queries with the fragments of the source schema visited by the mapping. By exploiting this information, we can analyze the behavior of the mapping on both the source and the target schema (a preliminary implementation of this approach can be found in the current version of our maintenance tool. [3]

By observing that a change in the intended semantics of a schema is more likely to corrupt a mapping if it affects a fragment that is close to those touched by the mapping, we can notify to the developer any schema change that is sufficiently close (according to some proper metric) to the portions of the schema involved in the mapping. As an example, in the case of working names and actual names, this extension will pinpoint the schema change as potentially harmful, as it is very close to name elements, which are touched (covered) by the mapping. Of course, no formal properties can be stated and proved for this extension; however, it provides useful information at a very low extra cost.

Decidability and Complexity of Type Projection

As we have seen in the previous sections, and in Proposition 3.1 in particular, if one can establish a projection relation between the inferred type and the target schema of a mapping, the correctness of the mapping is proved. In order to move towards a practical correctness checking technique, we first need to prove decidability of the type-projection relation.

To prove that type projection is decidable we rely on a particular notion of *type approximation*. Type approximation weakens types by enriching base and element types with a union with the empty sequence type; this allows one to relate type projection to standard subtyping for commutative types, whose decidability has been proved by [Huy85].

Type approximation is based on the idea of weakening types by introducing unions with the empty sequence type.

Définition 3.7 (Type approximation) Given a type U, we indicate with $U^{\triangleleft}$ the type obtained by U just by replacing each subexpression U', corresponding to a tree type $l[\_]$ or B, with $U'?$ (that is $(U' \mid ())$). Formally:

$$
\begin{array}{llllllll}
()^{\triangleleft} & \triangleq & () & & T \mid U^{\triangleleft} & \triangleq & T^{\triangleleft} \mid U^{\triangleleft} & & l[T]^{\triangleleft} & \triangleq & l[T^{\triangleleft}]? \\
\mathsf{B}^{\triangleleft} & \triangleq & \mathsf{B}? & & T,U^{\triangleleft} & \triangleq & T^{\triangleleft},U^{\triangleleft} & & T*^{\triangleleft} & \triangleq & T^{\triangleleft}*
\end{array}
$$

Decidability of type projection relation is stated by the following theorem.

Theorem 3.1 (Type projection as sub-typing)

$$T \lesssim U \quad \Leftrightarrow \quad T < U^{\triangleleft}$$

The previous theorem states that type projection between T and U can be checked by weakening U and, then, by checking for the existence of a subtyping relation between T and $U^{\triangleleft}$. This result

[3] It is available at `http://www.unibas.it/sartiani/projects/gamma.html`).

proves decidability of type projection, since [Huy85] proved that inclusion for commutative regular grammars is Π_2^p-hard and is in CoNEXPTIME. This result also identifies an upper bound for the complexity of type projection; however, in the following we will show that a better upper bound for complexity of projection can be found by relying on a type simulation approach.

The previous theorem, stating that type projection is equivalent to subtyping, once the right hand-side of the comparison has been approximated, gives rise to a fundamental question, i.e., whether type projection can be replaced, in the context of data integration and data exchange, by subtyping. A strong reason for discarding subtyping, in favor of type projection, is its algorithmic complexity.

So in order to efficiently check type projection, we propose in the following an alternative characterization of type projection which is not based on subtyping and which is a first step towards an efficient algorithm.

Before illustrating our technique for checking type projection, it is worth to analyze the computational complexity of type projection. From Theorem 3.1 we know that type projection is equivalent to subtype-checking when the right hand-side of the comparison has been *weakened*, i.e., $T \lesssim U \iff T < U^{\lhd}$. Since U can be transformed into $U^{\lhd}$ in polynomial time and space, this theorem also states that type projection cannot be more expensive than subtype-checking. Inclusion among commutative type is known to be in CoNEXPTIME [Huy85], hence CoNEXPTIME is an upper bound for type projection too. We will see next that this upper bound can be refined to EXPTIME.

For what concerns the complexity lower bound, we first introduce a supplementary operation called $\lesssim$*-membership*.

Définition 3.8 ($\lesssim$-membership) Given a data model instance f and a type T, we say that $f \lesssim_{mem} T$ if and only if $\exists f' \in [\![T]\!].f \lesssim f'$.

The relation $\lesssim_{mem}$ is here called $\lesssim$-membership as it is the counterpart of membership for inclusion and equivalence problems. It should be observed that $f \lesssim f'$ can be decided in polynomial time in the size of f and f'.

The following theorem shows that $\lesssim$-membership is NP-complete.

Theorem 3.2 *$\lesssim$-membership is NP-complete.*

The complexity of $\lesssim$-membership provides a lower bound for the complexity of type projection, as shown by the following corollary.

Corollary 3.1 *Type projection is NP-hard.*

Type Projection as Type Simulation

Type simulation is a *symbolic* relation among types, whose main aim is to provide a convenient way to characterize and check for type projection.

Type simulation is defined among types in *disjunctive normal form*, i.e., types where products are distributed across unions. A type T can be normalized by applying the normalization function $norm(T)$, defined as shown in Table 3.1. It is easy to show that the evaluation of $norm(T)$ always terminates.

$norm()$ works by transforming types, while preserving their semantics, so that the transformed types can be easily compared by the simulation relation (and by the corresponding algorithm). For instance, $norm(T'*, U'*, U)$ transforms a product of repetition types, which is hard to formalize in the simulation rules, into a *-guarded union, for which much easier simulation rules exist.

To eliminate some ambiguity, the rules of the $norm()$ must be applied in the order in which they are defined. $norm()$ can be applied to any type, and its relevance resides in the proof of equivalence between simulation and projection, as it will be clear in the rest of the paper.

Table 3.1. *norm*() *function.*

$norm(())$	$\triangleq$	$()$
$norm(\mathsf{B})$	$\triangleq$	B
$norm(l\,[T])$	$\triangleq$	$\begin{cases} \bigcup l\,[A_i] & \text{if } norm(T) = A_1 \mid \ldots \mid A_n \\ l\,[norm(T)] & \text{otherwise} \end{cases}$
$norm(T \mid U)$	$\triangleq$	$norm(T) \mid norm(U)$
$norm(T'*, U'*, U)$	$\triangleq$	$norm((T' \mid U')*, U)$
$norm(T, U)$	$\triangleq$	$\begin{cases} norm(A_1, U) \mid norm(A_2, U) & \text{if } norm(T) = (A_1 \mid A_2) \\ norm(T, A_1) \mid norm(T, A_2) & \text{if } norm(U) = (A_1 \mid A_2) \\ norm(T), norm(U) & \text{otherwise} \end{cases}$
$norm(T*)$	$\triangleq$	$norm(T)*$

The core of normalization is the transformation of type expressions in conjunctive normal form into equivalent ones in disjunctive normal form (see the third and the sixth rule in Table 3.1). As a consequence, *norm*() has an EXPTIME worst case time complexity, and the normalized type may have an exponential size wrt the original type. Despite this, for a vast class of types *norm*() can be computed in PTIME. This class contains types where unions are always guarded by a $*$-operator ($*$-guarded types), as shown by the following definition.

Définition 3.9 (SGT) A type T is in SGT (star-guarded types) if it can be generated by the following grammar:

***-Types** $T ::= () \mid \mathsf{B} \mid l[T] \mid T, T \mid U*$

Union Types $U ::= T \mid T \mid U$

Proving that for $*$-guarded types *norm*() is polynomial is straightforward. $*$-guardedness is a property enjoyed by a large number of commonly used DTDs and XSDs. For instance, the reader can refer to [Cho02] and to [BNdB04] for a detailed classification of real world DTDs: this classification shows that non-$*$-guarded unions are quite infrequent. In any case, in order for *norm*() to blow up, the $*$-guarded union restriction must be systematically violated, so a few occurrences of $*$-unguarded are harmless.

It is worth to notice that optional types of the form $A \mid ()$, even though representing a relatively frequent kind of non-$*$-guardedness, does not affect the complexity of *norm*() since they can be rewritten into A by preserving projection. So, from now on we can make the assumption that types do not contain optional types $A \mid ()$. This is assumption is non-restrictive since, as proved in [CS09], they can be safely eliminated.

It is easy to prove that that *norm*() preserves the semantics of types: for each type T, $[\![T]\!] = [\![norm(T)]\!]$.

Définition 3.10 A type T is *prime* if and only if $norm(T) = T$ and $T \neq A \mid B$.

Prime types play a crucial role in our framework. Since prime types are invariant under normalization and they cannot be union types, their semantics never contain mutually exclusive tree structures. For this reason, a prime type can be considered as a whole during projection checking, without the need of any kind of further transformation. This is ensured by the following lemma, formalizing the main property enjoyed by prime types.

Lemma 3.1 (Upward closure) *If T is prime, then* $\forall f_1, f_2 \in [\![T]\!].\exists f \in [\![T]\!].\ f_i \lesssim f$

We will need the following lemma that deals with projection among *-types. Essentially, as far as prime types are concerned, this lemma states that a type $T*$ is in the projection relation only wrt

to types U containing a *-type at the top level, that is $U = U_1*, A$ with A not containing *-types at the top level; moreover, only the *-type contributes to the projection, the proof being based on the cardinality of sequences.[4]

Lemma 3.2 *If $T*$ and U are prime, then $T* \lesssim U \Leftrightarrow U = (U_1)*, A$ and $T* \lesssim (U_1)*$.*

We can now state the definition of *type simulation*.

Définition 3.11 (Type simulation) The type simulation relation $\preceq$ among normalized types is defined as follows.

1)	B	$\preceq$	B	
2)	$()$	$\preceq$	U	
3)	$l[T]$	$\preceq$	$l[U]$	if $T \preceq U$
4)	T_1	$\preceq$	U_2, U_3	if $T_1 \preceq U_2 \vee T_1 \preceq U_3$
5)	T_1, T_2	$\preceq$	U_3, U_4	if $(T_1 \preceq U_3 \wedge T_2 \preceq U_4)$
6)	T_1	$\preceq$	$U_2 \mid U_3$	if $T_1 \preceq U_2 \vee T_1 \preceq U_3$ and $T_1 \neq V_1 \mid V_2$
7)	$T_1 \mid T_2$	$\preceq$	U	if $T_1 \preceq U \wedge T_2 \preceq U$
8)	T	$\preceq$	$U*$	if $T \preceq U$
9)	$T*$	$\preceq$	$U*$	if $T \preceq U*$
10)	T_1, T_2	$\preceq$	$U*$	if $T_1 \preceq U* \wedge T_2 \preceq U*$

Rules 1-3 are straightforward as well as Rule 8. Rules 4-5 describe the simulation among product types, while Rules 6-7 illustrate the simulation among union types. Rules 8-10, finally, are dedicated to repetition types.

Rules for product types are of special interest. In particular, Rule 5 shows that simulation between product types is *injective*, hence capturing the injective nature of projection: for instance, $T = Album, Album$ cannot be projected into $U = Album$, as data conforming to T have two distinct album elements, while data conforming to U have only one album element. Injectivity may be broken by repetition types, or when sequence types are in the immediate scope of a repetition type.

Rules 6-7 describe the simulation for union types. These rules pinpoint the commutative and *non-injective* nature of union types.

Type projection is equivalent to type simulation. The proof is a bit involved and can be found in [CS09].

Theorem 3.3 *Given two normalized types T and U:*

$$T \lesssim U \Leftrightarrow T \preceq U$$

Type Projection Checking

In the previous section we showed the equivalence between type projection and type simulation. This allows for the construction of an *efficient*, *simulation-based* projection-checking algorithm. The algorithm is actually a *not-so-naive* implementation of the type simulation rules. Indeed, a naive implementation of these rules would lead to a super-exponential algorithm, due to the following reasons. First of all, a recursive comparison of two types T_1 and T_2, as suggested by the simulation rules, would lead to many *backtracking* operations, in particular when comparing union or product types: for instance, when comparing $l[m[T]]$ with $l[m[\mathsf{B}]], l[m[T]]$ (where $T \neq \mathsf{B}$), a naive algorithm would (i) apply Rule 4 for product types and choose $l[m[T]]$ and $l[m[\mathsf{B}]]$ as types to be compared, (ii) start the comparison of the chosen types, and (iii) go back to Rule 4 and step (i) when the comparison fails. This problem can be solved by *flattening* T_1 and T_2, and by constructing a *type matrix* (*simTypes* in our algorithm), whose rows and columns are associated, respectively, to type

[4] Recall that each prime type can have at most one *-type at the top level.

terms in T_1 and in T_2. The type matrix is then used to compare each type term in T_1 with each type term in T_2 according to the hierarchy of type terms (hence, terms occurring in very distant fragments of T_1 and T_2 are not compared); by doing so, the algorithm does not perform backtracking, nor it performs comparisons among types that are "incompatible" according to the type term hierarchy.

The second key factor that makes naive algorithms super-exponential is the comparison among product types. The type simulation definition states that, if outside the immediate scope of a repetition type, $Z_1,\ldots,Z_n \preceq V_1,\ldots,V_m$ if and only if each type Z_i can be mapped into a distinct type in $V_1,\ldots,V_m$, so that there do not exist Z_i and Z_j ($i \neq j$) such that Z_i and Z_j are mapped into the same type term V_h. This problem is also present in many subtype-checking algorithms, and can be naively solved by generating all possible assignments of Z_i to V_j types and by choosing an injective one: this can be done in super-exponential time, as the possible assignments are $O(\binom{m}{n})$.

An alternative solution for the comparison of product types can be obtained by observing that this problem is equivalent to a *0-1 maximum flow* problem on *bipartite graphs*. Indeed, one can build a bipartite graph $\mathscr{G}$, whose first partition $\mathscr{P}_1$ contains one node per each Z_i type, and whose second partition $\mathscr{P}_2$ contains one node per each V_j type; nodes in $\mathscr{P}_1$ are connected to a source s, while nodes in $\mathscr{P}_2$ are connected to a sink t. $\mathscr{P}_1$ and $\mathscr{P}_2$ are connected together through edges satisfying the simulation relation, i.e., an edge from Z_i to V_j is inserted in $\mathscr{G}$ if $Z_i \preceq V_j$. Each edge has two possible values for its flow: 0 and 1. The source s emits a flow of n units, so $Z_1,\ldots,Z_n$ simulates $V_1,\ldots,V_m$ if and only if a flow of n units reaches the sink t. This can be determined by using a quite standard $0-1$ maximum flow algorithm on bipartite graph, whose complexity is $O((n+m)^3)$ [Gol98]. A similar technique is also used in [CPR05] for subtype-checking of product types in a rather restrictive type language.

Our algorithm for type projection checking is formalized in [CS09]. Concerning its complexity, we proved that its worst case complexity, while comparing T and U, is $O(nm(n+m)^3)$, where n is the number of terms in T, and m is the number of terms in U.

We previously showed that type projection is NP-hard. The above PTIME complexity does not conflict with that result, as the simulation checking algorithm works on normalized types only, and normalization has exponential complexity in the case of non *-guarded types.

4 Mapping Type Inference

As previously stated, our approach can be used to verify that a mapping m from T to U is correct only if we are able to infer a type U' describing the structure of the output of m, that is a type U' such that: for each $f : T$, $m(f) : U'$. So, m is correct if $U' \lesssim U$ holds.

Depending on the "precision" of the inferred type U', it may happen that m is correct while $U' \lesssim U$ does not hold. Such a *false-negative* is due to the fact that the type system has not been clever enough to infer a very precise type for the mapping, that is, a type whose semantics is quite close to the set of the mapping co-domain. Of course, in the presence of false negatives we bother the user with error warnings without any real motivation; furthermore, in this case it is very likely that the user makes unneeded (the mapping is correct!) efforts to change the mapping. So, it is crucial to develop inference techniques that return quite precise inferred types and decrease false negatives, so to improve the effectiveness of our approach.

In [CS09] we illustrate how a quite precise output type can be inferred for a mapping expressed in the μXQ language [Col04, CGMS04, Che08b]: since μXQ is rather expressive, we are quite confident that the proposed inference technique can be generalized to different mapping approaches. To this end we provide opportune query typing rules, prove soundness of the resulting type system, and also show that the resulting typing algorithm is precise and efficient at the same time.

5 Experimental Evaluation

In the previous sections we analyzed the theoretical properties of a projection-based approach for capturing errors in schema mappings. Furthermore, we proved that this approach has exponential complexity in the general case, and it is polynomial on many practical cases.

To experimentally validate the usefulness of the proposed approach, we have performed extensive tests by using a Java implementation of the type inference system and the projection checking algorithm. Experiments focused key aspects: precision and scalability.

An experimental analysis of the precision properties of our approach is important as type inference may generate over approximations of the output type of a given schema mapping, which, in turn, may lead to *false negatives*, i.e., mappings that are deemed as incorrect even though they are perfectly legal. Hence, our approach can be considered practically useful only if it keeps false negatives very low (or does not generate false negatives at all in most common practical scenarios). As a consequence, our first battery of experiments analyzes the precision of our approach on the data integration benchmark described in [ATV08]. This benchmark comprises most transformations used in practical scenarios and represents an interesting way to validate our approach on "real world" scenarios. Performed tests confirmed the high precision of the type inference system, no false negatives have been produced.

The second battery of experiments focused on scalability. We performed our scalability tests on the schema of the XML-encoded version of DBLP (available at `http://dblp.uni-trier.de/xml/`). This dataset has already been used for experimental evaluations of data integration and exchange techniques, and reflects the features of commonly used schemas [HHP+07, BMP+08]. While the source schema we used essentially consists of the DBLP DTD, the target schema we used features a significantly different element nesting wrt the source schema.

As our approach has a (single) exponential worst case complexity, its practical usefulness lies in its ability to beautifully scale and perform even on large schemas or complex mappings: a very precise maintenance algorithm is useless if it is slow on complex schemas and/or mappings. As a consequence, we analyzed the behavior of the algorithms when the number of rules in the mapping or the size of the source and/or target schemas change. In all considered cases, both type inference and projection checking algorithms scaled well, by showing that despite the exponential worst case complexity, our algorithms perform well in practical scenarios.

6 Related Work

Only a few works deal with the problem of mapping maintenance in data integration systems. [VMP04] present a framework and a semi-automatic tool (called ToMAS) for the incremental maintenance of Clio-like mappings. The key objectives of the paper are to preserve as much as possible the semantics of the mappings to be adapted and to avoid the reformulation of the whole mapping system, so to decrease the efforts required for the maintenance. To achieve these goals, the framework adopts an incremental maintenance strategy, based on the knowledge of a detailed list of the basic update steps applied to the schemas (either the source or the target schema). Of course, this strategy can be applied only when this information is known to the mapping designer, which is unlikely in the case of autonomous data sources. This approach, hence, is best suited for integration contexts where all data sources are controlled by cooperating organizations (or the same organization at all); in this sense, this approach is complementary to our one, which assumes that the data sources are fully autonomous and can be applied even in the absence of a detailed list of incremental updates.

It should be observed that, unlike our approach, ToMAS supports changes in both the structure and constraints of a schema; however, the correctness notion of ToMAS has a coarser grain than our one, as a mapping is deemed as incorrect (and adapted) when it just works on the same fragment of

the schema that has been modified.

The same correctness notion of [VMP04] is used in [YP05], where Hu and Popa propose an *ex-post* adaptation technique. The idea, inherited from [MRB03], is to create a mapping from the old version of the schema to the new version of the schema, and to compose this mapping with the existing mapping. Of course, these approaches are subject to well-known non-closure issues related to mapping composition.

Both the approaches of [VMP04] and [YP05] are based on the theoretical framework of Clio. Adapting our approach to support this framework is relatively easy, as mappings are expressed through correspondences and logical associations, whose output type can be easily inferred. Furthermore, this framework uses a nested-relational type system, where only labelled union types are supported; this kind of union types is less powerful than non-labelled unions (used in our type system as well as in XML Schema), and we strongly suspect that type projection can be decided in polynomial time when unions are labelled (see [BP99] for a detailed discussion about the reasons why labeled union types are not adequate for semistructured data and XML).

An alternative technique for detecting corrupted mappings in XML data integration systems is the one described in [CS05a]. This technique is based on the use of a type system capable of checking the correctness of a query, in a XQuery-like language [CGMS04], wrt a schema, i.e., if the structural requirements of the query are matched by the schema. By relying on this type system, a distributed type-checking algorithm verifies that, at each reformulation step, the transformed query matches the target schema, and, if an error is raised, informs the source of the target peers that there is an error in the mapping.

The technique described in [CS05a] has two main drawbacks. First, it is not *complete*, since wrong rules that are not used for reformulating a given query cannot be discovered. Second, the algorithm requires that a query were reformulated by the system before detecting a possible error in the mapping; this implies that the algorithm cannot directly check for mapping correctness, but, instead, it checks for the correctness of a mapping wrt a given reformulation algorithm. Hence, mapping correctness is not a *query-independent*, *semantics-based* property, but is strongly related to the properties of the reformulation algorithm.

Most works on mapping maintenance, in the context of data integration or data exchange systems, focus on the problem of detecting corrupted data sources *wrappers*. These approaches [Kus00, LMK03] are based on checkers that learn the most prominent syntactical features of data sources, and warn the administrator when newly probed data fail in matching these features. Since they focus on syntactical changes only, these approaches are quite limited and unsatisfactory. Essentially the same approach forms the basis for the Maveric system [MAL+05], which systematically monitors the characteristics of wrappers and mappings in data integration systems. The novelty of Maveric is its improved accuracy and efficiency, but it still does not offer any correctness or completeness properties for error discovering. These approaches can be integrated with our maintenance technique, as checkers can be instructed to periodically infer the schema of external data sources, hence allowing for a type projection checking.

Our system bears some resemblance with Spider [CT06]. Spider is a debugger for schema mappings based on the logical dependencies framework of Clio. Spider works by analyzing the correspondences between a source data instance and a target data instance, so to help the mapping designer in understanding the behaviour of a mapping. These correspondences are expressed by a forest of minimal routes, which link target elements with source elements (and other target elements too). Our system is not an alternative to Spider, but, instead, can be regarded as a complementary tool that can be used after a mapping has been created and successfully deployed: indeed, our tool comes into play when the integration system is running, while Spider is used before setting up the system.

There exist some similarities between our notion of type projection and the subsumption relation described in [KS01], but these similarities are quite vague, as subtyping implies projection while the same does not hold for subsumption.

7 Conclusion

This chapter has presented a technique based on type inference and type projection checking for detecting corrupted mappings in XML data integration systems. This technique can be used in any context where a schema mapping approach is used, and it is based on a semantic notion of mapping correctness, unrelated to the query transformation algorithms being used. This form of correctness works on the ability of a mapping to satisfy the target schema, and it is independent from queries.

By reducing type projection to standard subtyping among weakened types, we proved that type projection is decidable [Huy85]. We characterized type projection in terms of type simulation, and, then, used the type simulation rules to define a checking algorithm. The algorithm employs an alternative technique for comparing product types, based on the use of a $0-1$ maximum flow algorithm.

The equivalence between type projection and type simulation allowed us to discover some interesting properties of type projection, such as the injective nature of product types and the behavior of product and union types inside repetition types.

The use of a maximum flow algorithm for the projection of product types allowed for designing a correct and complete projection-checking algorithm with polynomial time complexity on normalized types.

Sommaire

CHAPTER

4

EFFICIENT XML SUBTYPE CHECKING

This chapter is dedicated to results about efficient algorithms for checking inclusion among XML schemas, namely a quadratic algorithm for the asymmetric case where the sub type is any type and the super type is a conflict-free type (Section 1), and a more efficient algorithm, still for the asymmetric case, which is linear when compared types meet some structural similarity properties, and which is quadratic otherwise (Section 2). The chapter ends with conclusive remarks (Section 3).

1 Efficient Asymmetric XML Subtyping

Different extensions of Regular Expressions (REs) with interleaving operators and counting are used to describe the content models of XML in the major XML type languages, such as DTDs, XML Schema, and RELAX-NG [FW04, rel01]. This fact raised new interest in the study of such extended REs, and, specifically, in the crucial problem of language inclusion. As pointed out by Mayer and Stockmeyer [MS94] and by Gelade et al. [GMN07], the problem is EXPSPACE-complete. This prevents any practical use of unrestricted versions of regular expressions extended with interleaving and counting.

In [CGS09b] we introduced a class of "conflict-free" REs with interleaving and counting, whose inclusion problem is in PTIME. The class is characterized by the single occurrence of each symbol and the limitation of Kleene-star and counting to symbols. Hence, an expression $a^*\&b^*$, denoting the interleaving of a sequence of a's and b's, is conflict-free, while $a \cdot b \cdot a$ and $(a \cdot b)^*$ are not. These very strict constraints have been repeatedly reported as being actually satisfied by the overwhelming

majority of content models that are published on the Web,[5] which makes that result very promising, and of immediate applicability to the problem of comparing two different human-designed content models.

However, the main use of subtype-checking is in the context of *type-checking*, where *computed types* are checked for inclusion into *expected types*. This happens in three major cases (in each case we use U for the human-defined expected type, and E for the expression whose type is computed by the compiler):

(i) when a function, expecting a type U for its parameter, is applied to an expression E;

(ii) when the result of an expression E is used to update a variable, whose type U has been declared;

(iii) when the body E of a function is compared with the human-declared output type U of the function, in order to declare it type-correct.

In all these cases, the expected type is defined by a programmer, hence we can restrict it to a conflict-free type with little harm. However, the computed type reflects the structure of the code. Hence, the same symbol may appear in many different positions, and Kleene star may appear everywhere. In this situation, the ability to compare two conflict-free types is too limited, and we have to generalize it somehow. Consider, for instance, the following XQuery-like function.

```
function alpha($y : int*) as (a* & b*) {
        for $x in $y
        return if ($x ⩽ 0)
               then a
               else a,b,a
}
```

A type-checker would infer a type $(a + (a \cdot b \cdot a))^*$ for the body of this function, a type that correspond to the structure of the code. This inferred type is not conflict-free, and must be compared for inclusion with the conflict-free declared output type $(a^* \,\&\, b^*)$.

Handling situations like this seemed very hard for a time. The result in [CGS09b] is based on an exact description of conflict-free types through constraints, which reduces type inclusion to constraint implication. The smallest generalization of the conflict-free single-occurrence and Kleene-star limitations makes types impossible to be exactly described by our constraints. This problem does not arise if types are extended with intersection, since our constraints are closed by intersection. However, we showed in [CGS09b] that just one outermost use of binary intersection in the subtype makes inclusion NP-hard.

This chapter presents a generalization of results in [CGS09b] without leaving PTIME. This is obtained by considering the *asymmetric inclusion problem*, i.e., the problem of verifying whether T is included in U, where T is unconstrained and U is conflict-free. Surprisingly enough, for this case inclusion is still in PTIME.

This result entails that a programmer must only declare conflict-free types, but the compiler can use the whole power of extended REs to approximate the result of any expression. The key for this result is understanding that, while the supertype has to be exactly described by the constraints, this is not necessary for the subtype.

To summarize, the contributions presented in this chapter are the following:

- we show that type inclusion can be reduced to constraint implication if the constraint extraction function fully characterizes the supertype;

[5] Quoting Bex et al. [BNST06] "*an examination of the 819 DTDs and XSDs gathered from the Cover Pages (including many high-quality XML standards) as well as from the web at large, reveals that more than 99% of the REs occurring in practical schemas are CHAREs (and therefore also SOREs)*" (see also Martens et al. [MNSB06]); our conflict-free types are more expressive than CHAREs; similar results, in the high range of 90%, have been reported by Barbosa et al. in [BLS06] and by Choi et al. in [Cho02].

- for each of the different kinds of constraints that our constraint language can express, we provide a polynomial algorithm to verify whether a generic type T satisfies that constraint;
- by combining the previous two contributions, we provide a quadratic algorithm to test whether T is included in U, where T and U are extended REs with interleaving and counting, provided that U is conflict-free, with no limitations on T.

Apart from the practical interest of a PTIME inclusion algorithm with no limitation on the subtype, this work also shows that the constraint approach is able to deliver interesting results in situations where traditional automata-based techniques are not easy to apply.

Results presented in this chapter are those published in [CGS09a], but we will present them according to the full version [CGS11a].

Related Work

Some Flavors of Determinism

Membership testing for full REs with interleaving and counting is NP-hard [MS94], hence extended languages meant for practical use are usually endowed with some restrictions, aimed to reduce membership complexity. These restrictions are typically designed to allow for the efficient construction of a compact deterministic automaton, and we introduce them here, since they also play a relevant role for the complexity of RE inclusion.

A typical restriction is *1-unambiguity*, that means (informally) that, when a string is analyzed, any analyzed character can be matched against one specific character in the regular expression, that is determined by the part of string that has been read so far. For example, $(ab)^+a$ is 1-unambiguous, but $(a?b)^*a$ is not: while reading $ba\ldots$, we do not know whether a should be matched against the first a or the second one. Single-occurrence is a stronger form of this constraint, meaning that no character occurs twice in an expression, which trivially implies 1-unambiguity. Conflict-freedom as defined here implies single-occurrence, hence also implies 1-unambiguity.

Strong determinism is a constraint stronger than 1-unambiguity, having to do with Kleene-star and with counting. Consider the expression $(a[1..2])[2..3]$. While reading $aa\ldots$, we do not know whether the second a matches the second repetition of a in $a[1..2]$, or whether we should match the whole $a[1..2]$ with the first a, and the second a with the first character of the second repetition of $a[1..2]$. Strong determinism means, very informally, that the part of string that has already been read and the current character determine both the next leaf to match and which counting operator (or Kleene star) is affected (see [GGM09] for a formal definition). Conflict-freedom implies strong determinism: since the content of a counting operator is just one character, there is no ambiguity about the effect of each character on the only counting operator that may contain it.

XML Schema is an important example of a language that is based on REs with counting, plus an extremely limited form of interleaving: the *all* group, that only allows symbols to be interleaved. XML Schema adopts a constraint known as *Unique Particle Attribution* (UPA) ([TBMM04], Section 3.6.6). There is some debate about the actual meaning of that constraint, but it is usually interpreted as a way to require 1-unambiguity [KT07, GGM09]. RELAX-NG [rel01] is another important language based on REs extended with a form of interleaving, but no counting. They adopt unordered concatenation, rather than shuffling, so that $(ab)\&(cde)$ only recognizes the two words *abcde* and *cdeab*. RELAX-NG does not impose any form of unambiguity in general, but they impose that, for any instance of $\&(E_1,\ldots,E_n)$, the first characters recognized by the E_i expressions are all mutually disjoint.

Conflict-freedom is very restrictive, but is trivial to define and check. The precise definition and automated checking of 1-ambiguity and strong determinism are a bit less trivial. In [GGM09], cubic time algorithms to test for 1-unambiguity and strong determinism are presented. In [Kil10], Kilpeläinen presents a $O(n^2/log(n))$ algorithm to test whether a RE with counting is 1-unambiguous, and

describes how some well-known studies and implementations of the same notion are actually incorrect.

Inclusion of regular expressions with interleaving and counting

The problem of inclusion of regular expressions with interleaving has been studied in many papers, but none of them provides PTIME inclusion algorithms for languages with interleaving, counting, and an expressive power that is acceptable for our intended application.

In [MS94], Mayer and Stockmeyer studied the complexity of membership, inclusion, and inequality for several classes of regular expressions with interleaving and intersection. In particular, interleaving is proved to make inclusion EXPSPACE-complete.

Starting from the results of [MS94], Gelade et al. [GMN07] studied the complexity of decision problems for DTDs, single-type EDTDs, and EDTDs with interleaving and counting. By considering several classes of regular expressions with interleaving and counting, they showed that their inclusion is almost invariably EXPSPACE-complete, even when counting is restricted to terminal symbols only; they also showed how these results extend to various kinds of schemas for XML documents. We did not discuss here how to extend our results from REs to XML schema languages because the problem is indeed solved in [GMN07], where it is shown how an inclusion algorithm for REs can be lifted to schema languages that use that class of REs without changing the complexity class. In [KT03, KT07] Kilpeläinen and Tuhkanen proved that inclusion is NP-hard for regular expressions with counting even if attention is restricted to 1-unambiguous REs.

The properties of a commutative type language for XML data have been discussed by Foster et al. in [FPS07]. Here, the authors essentially described the techniques they used while implementing a type-checker for commutative XML types. Their type language resembles our language of conflict-free types, as repetition types can be applied to element types only, and interleaving is supported. The paper is focused on heuristics that improve scalability, but do not affect computational complexity.

RELAX-NG [rel01] and XML Schema [FW04] are two well-known type languages that allow some form of interleaving and counting.

XML Schema adopts counting plus a very weak form of interleaving, with the UPA constraint. The coNP-hard problem presented for 1-unambiguous REs with counting in [KT03, KT07] can be easily expressed by a 1-unambiguous XML Schema, hence XML Schema inclusion is coNP-hard.

RELAX-NG restricts the use of interleaving ([rel01], Section 7.4) and has no counting. However, it does not restrict the expressions that use no interleaving, hence inclusion for RELAX-NG is PSPACE-hard [Koz77].

In [CGS09b] we defined a polynomial time algorithm for inclusion of conflict-free types, but we were not able to extend the result to reach any more general class. In that paper, we specified the constraint extraction procedure that we use here, and we proved that it is exact for conflict-free types.

Asymmetric inclusion of XML types has been studied elsewhere in the recent past. We discuss some of these papers here, but they are not very relevant to our problem since they deal with languages without interleaving and without counting. In [CS07] Colazzo and Sartiani showed that complexity of RE inclusion can be lowered from EXPSPACE to EXPTIME when a weaker form of conflict-freedom is satisfied by the supertype. In [CGLN09], by using automata-based encodings of types, Champavère et al. provide polynomial algorithms to check inclusion among EDTDs, with the restriction that the supertype is 1-unambiguous. In [Hov10] Hovland provides an efficient algorithm to check inclusion of standard REs. The algorithm runs in polynomial time. It is sound and complete when the supertype is 1-unambiguous, otherwise the algorithm may either terminate with a definite answer or may signal its inability to answer because the supertype is not 1-unambiguous. The algorithm is defined via an inference system driven by the REs syntax, hence avoiding possibly expensive automata construction.

Types and Constraints

We use the term "types" as a synonym for "extended regular expressions". Hence a "type" denotes a set of words. A *constraint* is a simple word property expressed in the constraint language we introduce below; a constraint denotes the set of words that satisfy it. We say that a type T satisfies a constraint F when every word in T satisfies F, that is, when the denotation of T is included in that of F. Hence, every type is upper-approximated by the set of all constraints that it satisfies. In [CGS09b] we introduced conflict-free types. For these types, this "approximation" is exact, meaning that a word belongs to a conflict-free type if and only if it satisfies all of its associated constraints.

Our algorithm is based on translating the supertype into a corresponding set of constraints and verifying, in polynomial time, that the subtype satisfies all of these constraints. In a mixed comparison, constraints provide an exact characterization for the conflict-free supertype, but just an upper-approximation for the subtype; we will prove below that this does not affect the correctness or completeness of the algorithm.

The Type Language

We describe here the specific syntax that we use for types.

We adopt the usual definitions for words concatenation $w_1 \cdot w_2$, and for the concatenation of two languages $L_1 \cdot L_2$. The *shuffle*, or *interleaving*, operator $w_1 \& w_2$ is also standard, and is defined as follows.

Définition 4.1 ($v\&w$, $L_1\&L_2$) The shuffle set of two words $v, w \in \Sigma^*$, or two languages $L_1, L_2 \subseteq \Sigma^*$, is defined as follows; notice that each v_i or w_i may be the empty word ε.

$$\begin{array}{lcl} v\&w & \stackrel{def}{=} & \{v_1 \cdot w_1 \cdot \ldots \cdot v_n \cdot w_n \mid v_1 \cdot \ldots \cdot v_n = v,\ w_1 \cdot \ldots \cdot w_n = w,\ v_i \in \Sigma^*,\ w_i \in \Sigma^*,\ n > 0\} \\ L_1\&L_2 & \stackrel{def}{=} & \bigcup_{w_1 \in L_1,\ w_2 \in L_2} w_1\&w_2 \end{array}$$

When $v \in w_1\&w_2$, we say that v is a shuffle of w_1 and w_2; for example, $w_1 \cdot w_2$ and $w_2 \cdot w_1$ are shuffles of w_1 and w_2.

We consider the following type language for words over an alphabet Σ:

$$T ::= \quad \varepsilon \mid a \mid T\,[m..n] \mid T + T \mid T \cdot T \mid T\&T \mid T!$$

where: $a \in \Sigma$, $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, and $n \geq m$. The set $\mathbb{N}_*$ is $\mathbb{N} \cup \{*\}$, where $*$ behaves as $+\infty$, i.e., for any $n \in \mathbb{N}_*$, $* \geq n$.

ε is a singleton type that only contains the empty word ε. The type $T!$ denotes the set of T words minus ε; a $T!$ type is well-formed only if a subterm of T has shape a. The type $T\,[m..n]$ denotes words that are formed by concatenating i words from T, with $m \leq i \leq n$.

Note that expressions like $T\,[0..n]$ are not allowed, due to the domain $(\mathbb{N} \setminus \{0\})$ of m, but the type $T\,[0..n]$ can be equivalently represented by $T\,[1..n] + \varepsilon$. The mandatory presence of an a subterm in $T!$ guarantees that T contains at least one word that is different from ε, hence $T!$ is never empty, which, in turn, implies that we have no empty types.

Définition 4.2 ($sym(w)$, $sym(T)$) For any word w, $sym(w)$ is the set of all symbols appearing in w. For any type T, $sym(T)$ is the set of all symbols appearing in T.

The semantics of types is inductively defined by the following equations.

$$\begin{array}{rcl}
[\![\varepsilon]\!] & = & \{\varepsilon\} \\
[\![a]\!] & = & \{a\} \\
[\![T_1 + T_2]\!] & = & [\![T_1]\!] \cup [\![T_2]\!] \\
[\![T_1 \cdot T_2]\!] & = & [\![T_1]\!] \cdot [\![T_2]\!] \\
[\![T_1 \& T_2]\!] & = & [\![T_1]\!] \& [\![T_2]\!] \\
[\![T!]\!] & = & [\![T]\!] \setminus \{\varepsilon\} \\
[\![T\,[m..n]]\!] & = & \{w \mid w = w_1 \cdot \ldots \cdot w_j, \quad \forall i \in 1..j.\ w_i \in [\![T]\!],\ m \leq j \leq n\}
\end{array}$$

We will use $\otimes$ to range over product operators $\cdot$ and & when we need to specify common properties, such as, for example: $[\![T \otimes \varepsilon]\!] = [\![\varepsilon \otimes T]\!] = [\![T]\!]$. We will use $\circledast$ to range over $\cdot$, &, and $+$.

Types that contain the empty word ε are called *nullable* and are characterized as follows. Observe that $\mathrm{N}(T\,[m..n]) = \mathrm{N}(T)$ because m cannot be 0.

Définition 4.3 $\mathrm{N}(T)$ is a predicate on types, defined as follows:

$$\begin{array}{rcl}
\mathrm{N}(\varepsilon) & = & \text{true} \\
\mathrm{N}(a) & = & \text{false} \\
\mathrm{N}(T!) & = & \text{false} \\
\mathrm{N}(T\,[m..n]) & = & \mathrm{N}(T) \\
\mathrm{N}(T + T') & = & \mathrm{N}(T) \text{ or } \mathrm{N}(T') \\
\mathrm{N}(T \otimes T') & = & \mathrm{N}(T) \text{ and } \mathrm{N}(T')
\end{array}$$

Property 4.1 ($\mathrm{N}(T)$**)**

$$\varepsilon \in [\![T]\!] \;\Leftrightarrow\; \mathrm{N}(T)$$

In this system, no type is empty, and any symbol in $sym(T)$ appears in some word of T.

Lemma 4.1 (Not empty) *For any type T:*

$$\begin{array}{ll}
[\![T]\!] \neq \varnothing & (1) \\
a \in sym(T) \;\Leftrightarrow\; \exists w \in [\![T]\!].\ a \in sym(w) & (2)
\end{array}$$

Constraints

Constraints are expressed using the following logic, where $a, b \in \Sigma$, $a \neq b$ in $a \prec b$, $A \subseteq \Sigma$, $B \subseteq \Sigma$, $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, and $n \geq m$:

$$F \;::=\; A^+ \mid A^+ \Rightarrow B^+ \mid a?[m..n] \mid \mathrm{upper}(A) \mid a \prec b$$

We do not explicitly consider conjunctive constraints $F \wedge F'$ since we will always associate types with *sets* of constraints, whose conjunction the type has to satisfy. Constraint semantics is defined in Figure 4.1.

The following special cases are worth noticing.

$$\begin{array}{lll}
a & \varepsilon \models \mathrm{upper}(A) & \varepsilon \models a?[m..n] \\
\varepsilon \models a \prec b & b \models a \prec b & a \\
a & w \models \varnothing^+ \Rightarrow A^+ & w \models \varnothing^+ \Rightarrow \varnothing^+
\end{array}$$

$$
\begin{array}{rcl}
w \models A^+ & \Leftrightarrow & \mathit{sym}(w) \cap A \neq \varnothing, \text{ i.e. some } a \in A \text{ appears in } w \\
w \models A^+ \mapsto B^+ & \Leftrightarrow & a \text{ or } w \models B^+ \\
w \models a?[m..n]\ (n \neq *) & \Leftrightarrow & \text{if } a \text{ appears in } w, \text{ then it appears at least } m \text{ times and at most } n \text{ times} \\
w \models a?[m..*] & \Leftrightarrow & \text{if } a \text{ appears in } w, \text{ then it appears at least } m \text{ times} \\
w \models \mathrm{upper}(A) & \Leftrightarrow & \mathit{sym}(w) \subseteq A \\
w \models a \prec b & \Leftrightarrow & \text{there is no occurrence of } a \text{ in } w \text{ that follows one occurrence of } b \text{ in } w
\end{array}
$$

FIGURE 4.1: Constraint semantics.

Observe that A^+ is monotone, i.e., $w \models A^+$ and w is subword of w' imply that $w' \models A^+$, while $\mathrm{upper}(A)$ and $a \prec b$ are anti-monotone.

A constraint F denotes the set of words that satisfy it, and a set of constraints $\mathscr{F}$ denotes the words that satisfy each $F \in \mathscr{F}$, as follows.

Définition 4.4 ($[\![F]\!]$ and $[\![\mathscr{F}]\!]$) For any constraint F, set of constraints $\mathscr{F}$:

$$[\![F]\!] \stackrel{def}{=} \{w \mid w \models F\} \qquad\qquad [\![\mathscr{F}]\!] \stackrel{def}{=} \cap_{F \in \mathscr{F}} [\![F]\!]$$

A type satisfies a constraint if all of its words do. The previous definition allows us to express this as set inclusion.

Définition 4.5 ($L \models F$, $T \models F$, $T \models \mathscr{F}$) For any set of words L, type T, constraint F, set of constraints $\mathscr{F}$:

$$L \models F \quad \Leftrightarrow_{def} \quad L \subseteq [\![F]\!] \qquad T \models F \quad \Leftrightarrow_{def} \quad [\![T]\!] \subseteq [\![F]\!] \qquad T \models \mathscr{F} \quad \Leftrightarrow_{def} \quad [\![T]\!] \subseteq [\![\mathscr{F}]\!]$$

Constraints and Subtyping

Consider a constraints language $\mathscr{F}$, a type T and a set of constraints $\mathscr{F}_T$. We define three properties that $\mathscr{F}_T \subseteq \mathscr{F}$ may satisfy for T:

- soundness: $\mathscr{F}_T$ is sound for T if $T \models \mathscr{F}_T$, that is, $[\![T]\!] \subseteq [\![\mathscr{F}_T]\!]$.
- $\mathscr{F}$-completeness: a sound $\mathscr{F}_T$ is complete for $\mathscr{F}$ and T if $[\![\mathscr{F}_T]\!] = [\![\{F \in \mathscr{F} \mid T \models F\}]\!]$, that is, $\mathscr{F}_T$ is the most precise description of T that can be expressed through $\mathscr{F}$.
- exactness: $\mathscr{F}_T$ is exact for T if $[\![T]\!] = [\![\mathscr{F}_T]\!]$.

A complete $\mathscr{F}_T$ is not necessarily exact for T, for example, no constraint set in our language is exact for the type $(aa)[1..2]$ (denoting $\{aa, aaaa\}$). However, if T admits in $\mathscr{F}$ an exact constraint set $\mathscr{F}_T$, then all and only its $\mathscr{F}$-complete sets of constraints are exact.

In the same way, a function $\mathscr{C}$ mapping types to sets of constraints, is called sound/$\mathscr{F}$-complete/exact, if $\mathscr{C}(T)$ is, respectively, sound, $\mathscr{F}$-complete, or exact, for any T.

In [CGS09b] we defined a class of "conflict-free types", defined as those types that respect the following restrictions (hereafter we will use the meta-variable U for conflict-free types):

- *symbol counting*: if U has a subterm $U'[m..n]$, then U' must be the type a, for some $a \in \Sigma$ (only symbols can be counted or subject to Kleene-star);

- *single occurrence*: if U has a binary subterm $U_1 \circledast U_2$, then $sym(U_1) \cap sym(U_2) = \varnothing$ (no symbol appears twice).

The symbol-counting restriction means that, for example, types like $(a \cdot b)^*$ cannot be expressed. However, it has been found that DTDs and XSD (XML Schema Definition) schemas use repetition almost exclusively as a^{op} or as $(a + \ldots + z)^{\mathrm{op}}$ (where $\mathrm{op} \in \{+, *\}$, see [BNST06]), which can be immediately translated to types that only count symbols, thank to the $U_1 \& U_2$ and $U!$ operators. For instance, $(a + \ldots + z)^*$ can be expressed as $(a^* \& \ldots \& z^*)$, where a^* is a shortcut for $a\,[1..*] + \varepsilon$, while $(a + \ldots + z)^+$ can be expressed as $(a^* \& \ldots \& z^*)!$.

The main result of [CGS09b] is the following exactness theorem.

Theorem 4.1 *There exists an exact constraint extraction function for conflict-free types.*

The proof of [CGS09b] is constructive, since we actually define a constraint extraction function $\mathscr{C}(U)$ satisfying $[\![U]\!] = [\![\mathscr{C}(U)]\!]$. This function can be used to reduce asymmetric inclusion to constraint-checking, as follows.

Proposition 4.1 (Mixed subtyping) *If $\mathscr{C}$ is exact for U, then $[\![T]\!] \subseteq [\![U]\!] \Leftrightarrow T \models \mathscr{C}(U)$.*

The property is asymmetric because U must admit an exact constraint-extraction function, but T can be any type.[6]

This observation is obvious once it is framed in the right context, but it provides a way to generalize our previous results that is very interesting: rather than hunting for generalizations of the conflict-free family in the narrow precinct of those types that can be exactly described, we can aim for the whole set of extended REs in the left hand side of $[\![T']\!] \subseteq [\![T'']\!]$, if we stay modest with the right hand side.

To exploit this observation, we need now to complement the exact constraint-extraction of [CGS09b] with a procedure to test for $T \models \mathscr{C}(U)$. In [CGS09b] we provided a cubic algorithm for the case when T is conflict free, while we proved that the problem is NP-hard when T ranges over conflict-free types with intersection.

In [CGS09b], we defined a constraint-extraction function that is exact for conflict-free types. For each type, this function extracts five classes of constraints: *co-occurrence* constraints $\mathscr{CC}(U)$, *order* constraints $\mathscr{OC}(U)$, *cardinality* constraints $ZeroMinMax(U)$, *lower-bound* constraints $SIf(U)$, and *upper-bound* constraints $upperS(U)$, that is, the exact function that we are going to use is defined as

$$\mathscr{C}(U) \quad = \quad \mathscr{CC}(U) \cup \mathscr{OC}(U) \cup ZeroMinMax(U) \cup upperS(U) \cup SIf(U)$$

To apply Proposition 4.1, in [CGS09b] we exhibit, for each component $\mathscr{C}_i(U)$ (where $\mathscr{C}_i(U)$ is one of $\mathscr{CC}(U)$, $\mathscr{OC}(U)$, etc.), an algorithm to verify whether, for each $F \in \mathscr{C}_i(U)$, $T \models F$, where T is a general type.

The algorithms for co-occurrence, ordering and cardinality constraints run all in quadratic time, while the upper and lower bound constraints are checked by a linear time algorithm.

Co-Occurrence Constraints

The first component $\mathscr{CC}(U)$ of $\mathscr{C}(U)$ extracts a set of co-occurrence constraints with shape $A^+ \Rightarrow B^+$, and is defined as follows, where $\{F \mid \neg \mathrm{N}(U)\}$ denotes the singleton $\{F\}$ when $\mathrm{N}(U)$ is false,

[6] We use the letter U since we apply this theorem to conflict-free types only, but it actually holds for any type U that is exactly described by $\mathscr{C}(U)$.

and denotes the empty set otherwise [CGS09b].

$$\begin{array}{lcl}
\mathscr{CC}(\varepsilon) & \stackrel{def}{=} & \varnothing \\
\mathscr{CC}(a\,[m..n]) & \stackrel{def}{=} & \varnothing \\
\mathscr{CC}(U!) & \stackrel{def}{=} & \mathscr{CC}(U) \\
\mathscr{CC}(U_1 + U_2) & \stackrel{def}{=} & \mathscr{CC}(U_1) \cup \mathscr{CC}(U_2) \\
\mathscr{CC}(U_1 \otimes U_2) & \stackrel{def}{=} & \{sym(U_1)^+ \Rightarrow sym(U_2)^+ \mid \neg \mathrm{N}(U_2)\} \\
 & & \cup\, \{sym(U_2)^+ \Rightarrow sym(U_1)^+ \mid \neg \mathrm{N}(U_1)\} \\
 & & \cup\, \mathscr{CC}(U_1) \cup \mathscr{CC}(U_2)
\end{array}$$

Example 4.1

- $\mathscr{CC}(a\,[1..2] \cdot b\,[2..*]) = \{a^+ \Rightarrow b^+,\ b^+ \Rightarrow a^+\}$
- $\mathscr{CC}(a\,[1..2] \cdot (b\,[2..*] + \varepsilon)) = \{b^+ \Rightarrow a^+\}$; $a^+ \Rightarrow b^+$ is not in $\mathscr{CC}(U)$ because $(b\,[2..*] + \varepsilon)$ is nullable.
- $\mathscr{CC}(a\,[1..2] \cdot (b\,[2..*] + c\,[1..*] + \varepsilon)) = \{bc^+ \Rightarrow a^+\}$;
- $\mathscr{CC}(a\,[1..1] \cdot (b\,[2..2] \cdot c\,[3..3])) = \{a^+ \Rightarrow bc^+,\ bc^+ \Rightarrow a^+,\ b^+ \Rightarrow c^+,\ c^+ \Rightarrow b^+\}$;

Since $\mathscr{CC}(U)$ yields constraints with shape $A^+ \Rightarrow B^+$, in the following we present an algorithm to test $T \models A^+ \Rightarrow B^+$ for any general type T.

We focus on constraints with shape $a^+ \Rightarrow A^+$ because of the following property, that is an immediate consequence of the definition of $A^+ \Rightarrow B^+$.

Property 4.2 (Union) *For any word w and constraint $A^+ \Rightarrow B^+$:*

$$w \models A^+ \Rightarrow B^+ \;\Leftrightarrow\; \forall a \in A.\ w \models a^+ \Rightarrow B^+$$

Our algorithm reduces $a^+ \Rightarrow B^+$ to $\Sigma^+ \Rightarrow B^+$, as it will be shown later. From now on, we abbreviate $\Sigma^+ \Rightarrow A^+$ as A^{++}. This notation is justified by the strict relationship between A^{++} and A^+: every non-empty word satisfies Σ^+, hence a word w satisfies A^{++} if w is empty or if w satisfies A^+ (Property 4.3).

Définition 4.6 (A^{++})

$$A^{++} \stackrel{def}{=} \Sigma^+ \Rightarrow A^+$$

Property 4.3 ($T \models A^{++}$ and $T \models A^+$)

$$\begin{array}{lcl}
T \models A^{++} & \Leftrightarrow & [\![T]\!] \setminus \{\varepsilon\} \models A^+ \\
T \models A^+ & \Leftrightarrow & [\![T]\!] \models A^{++} \ \&\ \neg \mathrm{N}(T) \quad (3)
\end{array}$$

Our algorithm to test $T \models a^+ \Rightarrow B^+$ may be based either on A^+ or on A^{++}. We focused on a version that is based on A^{++} because of the following lemma, that specifies an easy way to compute A^{++}. An inductive computation of A^+ is more involved, since it needs to resort on A^{++} for the $T!$ case.

Lemma 4.2 ($T \models A^{++}$) *For any type T, T_1 and T_2:*

$$
\begin{array}{lll}
(1) & \varepsilon \models A^{++} & \\
(2) & a \models A^{++} & \Leftrightarrow \quad a \in A \\
(3) & T\,[m..n] \models A^{++} & \Leftrightarrow \quad T \models A^{++} \\
(4) & T_1 \otimes T_2 \models A^{++} & \Leftrightarrow \quad (T_1 \models A^{++} \;\&\; \neg \mathrm{N}(T_1)) \vee (T_2 \models A^{++} \;\&\; \neg \mathrm{N}(T_2)) \\
 & & \qquad \vee\, (T_1 \models A^{++} \;\&\; T_2 \models A^{++}) \\
(5) & T_1 + T_2 \models A^{++} & \Leftrightarrow \quad T_1 \models A^{++} \;\&\; T_2 \models A^{++} \\
(6) & T! \models A^{++} & \Leftrightarrow \quad T \models A^{++}
\end{array}
$$

Our algorithm for co-occurrence constraints checking stronly relies on Theorem 4.2. It specifies that $T \models a^+ \Rightarrow B^+$ holds iff, for each occurrence a_i of a inside T, we can find a subterm T' of T that contains a_i and such that $T' \models B^{++}$.

As a small optimization, we show that the search for an ancestor of a that satisfies B^{++} can be restricted to product terms $T_1 \otimes T_2$.

Theorem 4.2 ($T \models a^+ \Rightarrow B^+$ from $T' \models B^{++}$) *For any type T, any $B \subseteq \Sigma$, any $a \in (\Sigma \setminus B)$, the following sentences are equivalent.*

1. $T \models a^+ \Rightarrow B^+$;
2. *for each occurrence of a inside T, the occurrence is part of a subterm T' of T such that $T' \models B^{++}$ and such that $T' = T_1 \otimes T_2$, for some T_1 and T_2.*

Based on previous results (Property 4.3 and Lemma 4.2), we have devised an algorithm CoImplies to verify that, for each $A^+ \Rightarrow B^+ \in \mathscr{CC}(U)$, $T \models A^+ \Rightarrow B^+$ holds (see [CGS11a, CGS09a] for details).

CoImplies builds $\mathscr{CC}(U)$, which can easily be made in time $O(|U| \times |U|)$, and then for each $A^+ \Rightarrow B^+ \in \mathscr{CC}(U)$, the following two operations are performed:

- by a bottom-up visit of the parse tree of T, $T \models B^{++}$ is checked, and also each subtree T' such that $T' \models B^{++}$ is marked; this phase is performed by mimicking Lemma 4.2, in $O(|T| + |B|)$ time.
- for each $a \in A$, the node corresponding to a in the T parse-tree is checked to verify whether it has been marked by the previous step; this operation can be performed in $O(|T| + |A|)$ time.

The algorithm concludes that $T \models A^+ \Rightarrow B^+$ if the second above step succeeds.

CoImplies performs the two above operations once for each $A^+ \Rightarrow B^+ \in \mathscr{CC}(U)$, i.e., at most twice for each $\otimes$ in U. Since $|A| + |B| \leq |U|$, CoImplies has $O(|U| \times (|T| + |U|))$ worst case time complexity, which is even better than the algorithm that we defined in [CGS09b] for the pure conflict-free case.

Remark 4.1
Although $\mathscr{C}(U)$ is $\mathscr{F}$-complete for a conflict-free type U, $\mathscr{CC}(U)$ is not complete for U with respect to constraints with shape $A^+ \Rightarrow B^+$. For example, $\mathscr{CC}(a)$ is the empty set, which denotes the whole Σ^*, and it could be made more precise by adding any non-trivial $A^+ \Rightarrow B^+$ sound for a, such as, for example, $b^+ \Rightarrow c^+$, which is sound since b is disjoint from a, and excludes words such as b.

$\mathscr{C}(U)$ is $\mathscr{F}$-complete because it complements $\mathscr{CC}(U)$ with the constraint upper($sym(U)$). For example, in this case, upper(a) makes $b^+ \Rightarrow c^+$ redundant.

A similar remark holds for the order constraints that we define in the next section: $\mathscr{OC}(U)$ is complete for order constraints that only use symbols in $sym(U)$, but is not complete for every possible order constraint. However, our result does not require that every component of $\mathscr{C}(U)$ is complete on its class of constraints, but only that the whole of $\mathscr{C}(U)$ is $\mathscr{F}$-complete.

Order Constraints

Let us define $\mathscr{P}(T)$ as the set of all pairs of different symbols (a,b) such that there exists a word in $[\![T]\!]$ where an a comes before a b.

Définition 4.7 (Pairs)

$$\mathscr{P}(T) \stackrel{def}{=} \{(a,b) \mid \quad a \neq b, \quad \exists w_1,w_2,w_3.\ w_1 \cdot a \cdot w_2 \cdot b \cdot w_3 \in [\![T]\!]\}$$

Order constraints specify which pairs cannot appear in a word, hence $\mathscr{P}(T)$ is related to order constraints as follows.

Property 4.4 $T \models a \prec b \ \Leftrightarrow\ a \neq b$ *and* $(b,a) \notin \mathscr{P}(T)$

We verify whether a pair $(b,a) \in \mathscr{P}(T)$ by testing, for each instance of a and b in T, their Lower Common Ancestor (LCA) in the syntax tree of T; to this aim, we will manipulate a decorated version of T, $l(T)$, where each instance of a leaf is decorated with a distinct index i, and is denoted as a_i, and will consider the words generated by $l(T)$.

For example, if $T = a+b$, then $l(T) = a_1+b_2$, and the LCA of a_1 and b_2 in $l(T)$ ($LCA_{l(T)}[a_i,b_j]$) is $+$. The fact that the LCA of a_1 and b_2 is $+$ implies that a_1 and b_2 never appear together in a word of $l(T)$, hence $(b_2,a_1) \notin \mathscr{P}(l(T))$, hence, since no other instance of a and b is present in T, $(b,a) \notin \mathscr{P}(T)$. In $a_1 \& b_2$, the LCA is &, meaning that both (a,b) and (b,a) are in $\mathscr{P}(T)$. The use of LCA is justified by Lemma 4.1: with any two types T_1 and T_2, as soon as $a_i \in sym(T_1)$ and $b_j \in sym(T_2)$, then T_1 has a word with a and T_2 has a word with b, hence (a,b) and (b,a) are in $\mathscr{P}(T_1 \& T_2)$. In a type $a \cdot b$, order is relevant: $(a,b) \in \mathscr{P}(T)$ but $(b,a) \notin \mathscr{P}(T)$. We express this by extending the usual definition of $LCA_{l(T)}[a_i,b_j]$, assuming that it returns a pair $\circledast^d$, where the direction d is $\rightarrow$ if the leaf a_i comes before b_j in T, and is $\leftarrow$ otherwise; we ignore the direction when $\circledast \neq \cdot$ (see Example **??**).

$LCA_{l(T)}[a_i,b_j] \in \{\&,\cdot^{\rightarrow}\}$ implies that $(a,b) \in \mathscr{P}(T)$, but $(a,b) \in \mathscr{P}(T)$ also holds when $LCA_{l(T)}[a_i,b_j] \in \{+,\cdot^{\leftarrow}\}$, provided that the LCA is in the scope of a $T\,[m..n]$ operator with $n > 1$, as in $(a+b)\,[1..2]$ or in $(b \cdot a)\,[1..2]$; for this reason, in $l(T)$, we mark as $\circledast_r$ (for *repeated*) all binary operators $\circledast$ in the scope of a $T\,[m..n]$ with $n > 1$, and use $\circledast_1$ for all the other operator instances. Finally, if many occurrences of a and b appear in T, then $(a,b) \in \mathscr{P}(T)$ as soon as one pair (a_i,b_j) satisfies the test we described.

The formalization of an algorithm implementing the above method is detailed in [CGS11a, CGS09a]. The algorithm complexity is $O(|T|^2 + |U|^2)$. Hence, also in this case, the extension from conflict-free inclusion to asymmetric inclusion

Cardinality Constraints

A cardinality constraint $T \models a?[m..n]$ specifies the minimum and maximum number of times that a may appear in a word of T where a actually appears, which we will denote here as $\mathrm{Min}^{app}(T,a)$ and $\mathrm{Max}(T,a)$. $\mathrm{Min}^{app}(T,a)$ is different from the minimum number of times that a may appear in any word of T, which we will denote here as $\mathrm{Min}(T,a)$. For example, $\mathrm{Min}^{app}((a\,[3..*]+b),a) = 3$ while $\mathrm{Min}((a\,[3..*]+b),a) = 0$. Observe that this distinction has no meaning when we count the maximum number of occurrences of a in a word of T.

The cardinality constraints for a conflict-free type simply correspond to the instances of the counting operator. In particular, the cardinality constraint component of $\mathscr{C}(U)$ is $ZeroMinMax(U)$, defined as follows; $ZeroMinMax(U)$ is trivially complete for conflict-free types and for constraints with shape $a?[m..n]$ and $a \in sym(U)$ [CGS09b]:

$$ZeroMinMax(U) = \{a?[m..n] \mid a\,[m..n] \text{ subterm of } U\}$$

General types are trickier, because of symbol repetition and generalized counting. In particular, the lowest allowed cardinality for a in T may depend on the validity of a^+ on some subterm of T. Consider, for example, the type $a\,[2..*]\cdot a\,[3..*]$: it clearly satisfies $a?[5..*]$. However, the type $(a\,[2..*]+\varepsilon)\cdot(a\,[3..*]+\varepsilon)$ only satisfies $a?[2..*]$: since a is optional on both sides, we consider here $\min(2,3)$ rather than $2+3$. Finally, $(a\,[2..*]+\varepsilon)\cdot(a\,[3..*])$ satisfies $a?[3..*]$: since a is optional in the first subterm, we have to consider the bound of the second. In the same way, while $a\,[3..*]\,[4..*]$ satisfies $a?[12..*]$, the type $(a\,[3..*]+\varepsilon)\,[4..*]$ only satisfies $a?[3..*]$.

We solve this issue by recursively computing both $\mathrm{Min}^{app}(T,a)$ and $\mathrm{Min}(T,a)$ at the same time. This allows us to compute $\mathrm{Min}^{app}(T_1\otimes T_2,a)$ and $\mathrm{Min}^{app}(T\,[m..n],a)$ as follows.

$$\begin{array}{lcl}\mathrm{Min}^{app}(T_1\otimes T_2,a) & = & \min(\mathrm{Min}^{app}(T_1,a)+\mathrm{Min}(T_2,a),\ \mathrm{Min}(T_1,a)+\mathrm{Min}^{app}(T_2,a))\\ \mathrm{Min}^{app}(T\,[m..n],a) & = & \mathrm{Min}^{app}(T,a)+(m-1)\cdot\mathrm{Min}(T,a)\end{array}$$

The first equation corresponds to the fact that any word of $T_1\otimes T_2$ that contains a is built by combining a word of T_1 that contains a with any word of T_2, or by combining a word of T_2 that contains a with any word of T_1. The reader may verify that the formula works with all the examples we presented. The second equation is very similar: a word of $T\,[m..n]$ that contains a is obtained by combining a word of T that contains a with $m-1$ words of T. Unfortunately, the recursive computation of $\mathrm{Min}(T!,a)$ needs one further notion, the minimum number of occurrences of a in a non-empty word of T, which we will denote as $\mathrm{Min}^{!}(T,a)$. To sum up, we need the following three functions to be computed.

Définition 4.8 ($\mathrm{Min}(T,a)$, $\mathrm{Min}^{!}(T,a)$, $\mathrm{Min}^{app}(T,a)$) Let W be a set of words, a a symbol, and T a type.

$$\begin{array}{lcll}\mathrm{MinOrStar}(W,a) & \stackrel{def}{=} & \min_{w\in W}|w|_a & \text{if } W\neq\varnothing\\ \mathrm{MinOrStar}(W,a) & \stackrel{def}{=} & * & \text{if } W=\varnothing\\ \mathrm{Min}(T,a) & \stackrel{def}{=} & \mathrm{MinOrStar}([\![T]\!],a) & \\ \mathrm{Min}^{!}(T,a) & \stackrel{def}{=} & \mathrm{MinOrStar}(([\![T]\!]\setminus\{\varepsilon\}),a) & \\ \mathrm{Min}^{app}(T,a) & \stackrel{def}{=} & \mathrm{MinOrStar}(\{w\mid w\in[\![T]\!]\ \&\ w\models a^+\},a) & \end{array}$$

In [CGS11a] we give a set of properties of these three functions that allow them to be inductively computed in $O(|T|)$ time.

The upper bound is much easier, and is defined and computed as follows. We need no special symbol for $\mathrm{Max}(\varnothing,a)$, since we never apply $\mathrm{Max}(T,a)$ to an empty set.

Définition 4.9 ($\mathrm{Max}(T,a)$)

$$\begin{array}{lcll}\mathrm{Max}(T,a) & \stackrel{def}{=} & \max_{w\in[\![T]\!]}|w|_a & \text{if } (\max_{w\in[\![T]\!]}|w|_a)\in\mathbb{N}\\ \mathrm{Max}(T,a) & \stackrel{def}{=} & * & \text{if } \forall n\in\mathbb{N}.\ \exists w\in[\![T]\!].\ |w|_a>n\end{array}$$

Lemma 4.3 ($\mathrm{Max}(T,a)$)

$$\begin{array}{lcl}\mathrm{Max}(T_1+T_2,a) & = & \max(\mathrm{Max}(T_1,a),\mathrm{Max}(T_2,a))\\ \mathrm{Max}(T_1\otimes T_2,a) & = & \mathrm{Max}(T_1,a)+\mathrm{Max}(T_2,a)\\ \mathrm{Max}(b,a) & = & \mathit{if}\ \ b=a\ \mathit{then}\ 1\ \mathit{else}\ 0\\ \mathrm{Max}(T\,[m..n],a) & = & n\cdot\mathrm{Max}(T,a)\\ \mathrm{Max}(T!,a) & = & \mathrm{Max}(T,a)\\ \mathrm{Max}(\varepsilon,a) & = & 0\end{array}$$

By the definition of $\mathrm{Min}^{app}(T,a)$ and $\mathrm{Max}(T,a)$, cardinality constraint satisfaction can be decided as follows.

Corollary 4.1

$$T \models a?[m..n] \;\Leftrightarrow\; m \leq \mathrm{Min}^{app}(T,a) \;\wedge\; \mathrm{Max}(T,a) \leq n$$

From above commented properties, we can derive an algorithm,CARDIMPLIES, to verify that a general type T satisfies every F in $\mathit{ZeroMinMax}(U)$. The algorithm computes, in one pass, the values of $\mathrm{Min}(T,a)$, $\mathrm{Min}^{!}(T,a)$, $\mathrm{Min}^{app}(T,a)$ and $\mathrm{Max}(T,a)$. The values of $\mathrm{Min}^{app}(T,a)$ and $\mathrm{Max}(T,a)$ are then used to verify the constraint satisfaction. CARDIMPLIES can be computed in time $O(|U| \times |T|)$.

Upper Bounds and Lower Bounds

The upper bound and lower bound components of $\mathscr{C}(U)$ are defined below.

Définition 4.10 (Upper and Lower components of $\mathscr{C}(U)$)

$$\begin{aligned} \text{Lower-bound: } \mathit{SIf}(U) &\stackrel{def}{=} \text{if } \neg\mathrm{N}(U) \text{ then } \{\mathit{sym}(U)^{+}\} \text{ else } \varnothing \\ \text{Upper-bound: } \mathit{upperS}(U) &\stackrel{def}{=} \{\mathrm{upper}(\mathit{sym}(U))\} \end{aligned}$$

Notice that the problem of constraint implication is simplified by verifying the implication of lower and upper bounds at the same time, as we do here: we do not need to explicitly test whether $T \models \mathit{sym}(U)^{+}$; by restricting ourselves to the case when $T \models \mathit{upperS}(U)$, we only have to check that $\mathrm{N}(T) \Rightarrow \mathrm{N}(U)$, as proved below.

Theorem 4.3 (Implication of $\mathit{SIf}(T_2)$ and $\mathit{upperS}(T_2)$) *For any two types T_1 and T_2:*

$$\begin{aligned} & T_1 \models \mathit{SIf}(T_2) \cup \mathit{upperS}(T_2) \\ \Leftrightarrow\; & (\mathrm{N}(T_1) \Rightarrow \mathrm{N}(T_2)) \;\;\&\;\; \mathit{sym}(T_1) \subseteq \mathit{sym}(T_2) \end{aligned}$$

The corresponding function UpperLowerImplies simply executes the test of Theorem 4.3, hence we provide no pseudocode.

Summing up

We have recalled each of the five components of the constraint-extraction function $\mathscr{C}(U)$, and, for each component $\mathscr{C}_i$, we defined a function that verifies, for any general T, whether $T \models \mathscr{C}_i(U)$. Since the union of these five components is exact for conflict-free types, the following theorem holds.

Theorem 4.4 *For any type T, for any conflict-free type U, $[\![T]\!] \subseteq [\![U]\!]$ iff all of CoImplies(T, U), OrderImplies(T, U), CardImplies(T, U), UpperLowerImplies(T, U) return* ***true****.*

CoImplies, OrderImplies, and CardImplies have quadratic time-complexity, while UpperLowerImplies is linear. The only case whose complexity is affected by the presence of general types in the subtype position is that of cardinality constraints, where the presence of multiple occurrences of a symbol and the nesting of $T\,[m..n]$ operators both concur in making the problem less trivial.

2 Linear XML subtyping

The algorithm presented in the previous chapter is quadratic both in the best and in the worst cases. Indeed, that algorithm extracts a set of "constraints" from the candidate supertype and verifies that each constraint is satisfied by the subtype, which involves a quadratic amount of work, even in cases when the two types are very similar, or equal.

This chapter presents a new algorithm, still for the asymmetric case $T \leq U$ considered in the previous chapter, which is linear-time in the common situations where compared types are similar, and resorts to the quadratic approach only for those specific portions of the two types where it seems to be necessary.

This new algorithm has a more traditional "structural" approach: it visits both types in parallel from the top, matching the topmost operator and recurring on the children. This approach is not complete, since EREs may be included in cases when the topmost operators are permuted in quite complex ways, hence the structural approach should be combined with the quadratic approach to yield a complete algorithm. A naïve algorithm could just apply an incomplete set of structural rules and, when these fail, go back to the original types and apply the quadratic algorithm. Hence the algorithm would be better than the quadratic one in those cases when the structural rules suffice, but would impose an overhead otherwise. Unfortunately, choosing the optimal set of structural rules is impossible under this approach. A simple set of rules would be very effective in only a small set of cases. A richer set of rules would enlarge the set of cases where the algorithm is effective, but would impose a higher overhead in those cases where the structural work is useless. Our understanding of the "typical" workload of a type-checking compiler is too limited for a reasoned choice of an optimal set of rules.

We overcome this problem by designing a set of no-backtracking structural rules: whenever these rules rewrite a comparison into a set of simpler comparisons, the new set is not just a sufficient condition for the previous comparison, but it is equivalent. In this way, once a comparison that matches no rule is found, we do not need to go back to the initial types, but we can apply the quadratic algorithm to the smaller type fragments, so that the algorithm is always convenient over the baseline. These no-backtracking rules for EREs are the main contribution of this work, together with a technique to select the applicable rule in constant time.

In this chapter we present a linear structural algorithm for comparing binary types, and prove that it can resort to the quadratic one in case of failure without any backtracking. This algorithm can be then extended to to n-ary types, so to generalize the structural to most comparison cases. We will make an overview about this extension, and report some experimental results showing that the flat structural algorithm can be up to ten times faster than the quadratic algorithm presented in the previous chapter.

Structural approach

Introduction

Our algorithm is based on the assumption that subtyping would typically be applied to types that are very similar, such as $b \leq a + b + c$, or $a \cdot b \cdot c \leq a\&b\&c$ or $a \cdot b \leq a? \cdot b?$, where $a?$ abbreviates $a + \varepsilon$. Most of these cases may be proved by combining transitivity, associativity and commutativity with some obvious structural rules, such as:

- Monotonicity: $T_1 \leq U_1 \ \wedge \ T_2 \leq U_2 \Rightarrow T_1 \circledast T_2 \leq U_1 \circledast U_2$
- Union: $T_1 \leq T_1 + T_2$
- Product: $T_1 \cdot T_2 \leq T_1 \& T_2$.

Our algorithm is defined, as usual, by a set of deduction rules which are meant to be used to deterministically reduce the consequence to the premises. For example, one may encode monotonicity of '$\cdot$' by the following rule.

$$\frac{T_1 \leq U_1 \qquad T_2 \leq U_2}{T_1 \cdot T_2 \leq U_1 \cdot U_2} \qquad (\text{TENTATIVE-}\cdot\cdot)$$

Unfortunately, this rule requires backtracking: if we apply it to $(a\cdot b)\cdot c \leq a\cdot(b\cdot c)$, it would reduce it to $a\cdot b \leq a,\ c \leq b\cdot c$; the second set does not hold, but still the original judgement was true. Our basic observation is that backtracking is not needed if the symbols of U_1 and U_2 are disjoint, as always happens since U is conflict-free, and if the symbols of T_1 and T_2 are respectively included in those of U_1 and U_2. In this case, the rule needs no backtracking: the judgement in the conclusion holds if, and only if, all those in the premises do hold, as expressed by the following property.

$$sym(T_1) \subseteq sym(U_1) \ \wedge\ sym(T_2) \subseteq sym(U_2) \Rightarrow\ (T_1\cdot T_2 \leq U_1\cdot U_2 \Leftrightarrow T_1 \leq U_1 \wedge T_2 \leq U_2)$$

The situation is slightly more complex for union types: if we substitute $\cdot$ with $+$ in the above property, the double implication does not hold, as in the following example: $a? + b \leq a + b?$ holds, symbol inclusion holds, but still $a? \leq a$ does not hold. This problem can be solved by separating empty words from non-empty words, as follows. We first define a *kernel-subtyping* relation $T \leq_k U$, as follows.

Définition 4.11 ($T \leq_k U$)

$$T \leq_k U \Leftrightarrow_{def} [\![T]\!] \setminus \{\varepsilon\} \subseteq [\![U]\!] \setminus \{\varepsilon\}$$

Now, we have the following double implication.

$$\begin{array}{ll} & sym(T_1) \subseteq sym(U_1) \ \wedge\ sym(T_2) \subseteq sym(U_2) \\ \Rightarrow & T_1 + T_2 \leq U_1 + U_2 \\ & \Leftrightarrow T_1 \leq_k U_1 \wedge T_2 \leq_k U_2 \wedge \mathrm{N}(T_1 + T_2) \Rightarrow \mathrm{N}(U_1 + U_2) \end{array}$$

While it is quite natural to define $\leq$ and $\leq_k$ by mutual recursion, our algorithm recursively computes the *kernel-subtyping* relation $T \leq_k U$ only, since the standard subtyping relation $T \leq U$, defined as $[\![T]\!] \subseteq [\![U]\!]$, can be easily derived by adding a linear-time test $(\mathrm{N}(T) \Rightarrow \mathrm{N}(U)))$ to $T \leq_k U$, as specified by the following, obvious, property.

Property 4.5 $T \leq U \Leftrightarrow (T \leq_k U \wedge\ (\mathrm{N}(T) \Rightarrow \mathrm{N}(U)))$

Symbol inclusion tests and use of $T \leq_k U$ give us the ability to write rules that require no backtrack. Our algorithm uses these rules to reduce the problem as far as possible and, once no rule can be applied, it resorts to the quadratic algorithm. We need now to define a good set of rules, rich enough to deal with a good set of cases, but still with the property that looking for the next rule to apply should be extremely fast, so that the algorithm is linear on the size of the types.

We now introduce the rules, and will then formalize the algorithm, and discuss its correctness and its cost.

Structural sub-type rules

As previously discussed, our structural rules have the following shape.

$$\frac{Cond_R(T,U)}{\begin{array}{c} T_1^R \leq_k U_1^R, \ldots, T_n^R \leq_k U_n^R \\ \hline T \leq_k U \end{array}} \qquad \text{(R)}$$

The premise is formed by a condition $Cond(T,U)$, and a finite number of predicates $T_i^r \leq_k U_i^r$, with T_i^r and U_i^r sub-terms of, respectively, T and U. The rule means that, if $Cond_R(T,U)$ holds, then $T \leq_k U$ is equivalent to $T_1^R \leq_k U_1^R, \ldots, T_n^R \leq_k U_n^R$. The algorithm selects a rule whose $Cond_R(T,U)$ holds, and uses it to rewrite the conclusion to the premises.

The condition $Cond_r(T,U)$ of every rule is composed by a pattern-matching part and a part that depends, among other things, on symbol inclusion. The pattern-matching part is usually written in the rule conclusion, hence we will follow this habit, and write

$$\frac{\begin{array}{c} sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \\ T_1 \leq_k U_1, T_2 \leq_k U_2 \end{array}}{T_1 + T_2 \leq_k U_1 + U_2} \qquad (\text{ABBR-DIVIDE}{+}{+})$$

instead of:

$$\frac{\begin{array}{c} T = T_1 + T_2 \wedge U = U_1 + U_2 \wedge sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \\ T_1 \leq_k U_1, T_2 \leq_k U_2 \end{array}}{T \leq_k U} \qquad (\text{DIVIDE}{+}{+})$$

The structural subtyping rules are collected in Table 4.1. All of these rules are 'bidirectional', meaning that, when all the conditions hold, the premises are equivalent to the conclusion. As previously discussed, bidirectionality is a consequence of the fact that $sym(U_1)$ and $sym(U_2)$ are disjoint, of the symbol inclusion conditions, and of the use of $\leq_k$ in the $+\!+$ case.

Most of the rules are self explicative, and their bidirectionality is proved in [CGPS09]. The only non-trivial detail is the use of nullability in the premises. In the three (DIVIDE$\otimes\otimes$) rules the nullability condition is needed for the direct implication to be sound. If the first nullability condition were violated, we would have $\varepsilon \in [\![T_1]\!]$, a non empty word w_2 in $[\![T_2]\!]$ and $\varepsilon \notin [\![U_1]\!]$. Hence, w_2 would belong to $T_1 \otimes T_2$ and w_2 would not contain any symbol from U_1, hence it could not belong to $U_1 \otimes U_2$, which only contains words that contain some symbol from U_1. Observe that this complication derives from the use of $\leq_k$, since $T_1 \leq U_1$ would imply $\mathrm{N}(T_1) \Rightarrow \mathrm{N}(U_1)$, and similarly for T_2.

Nullability of U_2 in the (NFOCUS$\otimes$) rules is not related to $\leq_k$, but it is the kernel of the rule itself, which is based on the observation that, if $\varepsilon \in [\![U_2]\!]$, then $[\![U_1]\!] \subseteq [\![U_1 \otimes U_2]\!]$. The same observation, applied to both factors, justifies the (NDIVIDE) rules.

Observe that this set of rules is by no means complete. For example, one may add the following rule, to take commutativity of '+' into account.

$$\frac{\begin{array}{c} sym(T_1) \subseteq sym(U_2) \wedge sym(T_2) \subseteq sym(U_1) \\ T_1 \leq_k U_2, T_2 \leq_k U_1 \end{array}}{T_1 + T_2 \leq_k U_1 + U_2} \qquad (\text{REV-DIV}{+}{+})$$

Unfortunately, associativity is at least as important as commutativity, but is far more difficult to deal with. We hence present here just a minimal set of rules, to illustrate the basic ideas, and we discuss our approach to associativity and commutativity later.

The algorithm

The algorithm is described below. It first calls the auxiliary algorithm PREPROCESS(T,U), which prepares the types for efficient subtype checking. The algorithm then verifies whether a rule r exists such that $Cond_r(T,U)$ holds. If the rule exists, it is applied, and the problem is split in simpler problems, to be solved in subsequent iterations of the while-loop. When we find a subproblem where no rule is applicable, the algorithm resorts to the quadratic algorithm ORACLE(T,U) described in [CGS09a].

$$\frac{sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \quad T_1 \leq_k U_1, T_2 \leq_k U_2}{T_1 + T_2 \leq_k U_1 + U_2} \quad (\text{DIVIDE}{+}{+})$$

$$\frac{\begin{array}{c} sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \\ \wedge\ (\mathrm{N}(T_1) \wedge (sym(T_2) \neq \varnothing) \Rightarrow \mathrm{N}(U_1)) \\ \wedge\ (\mathrm{N}(T_2) \wedge (sym(T_1) \neq \varnothing) \Rightarrow \mathrm{N}(U_2)) \\ T_1 \leq_k U_1 \wedge T_2 \leq_k U_2 \end{array}}{T_1 \& T_2 \leq_k U_1 \& U_2} \quad (\text{DIVIDE}\&\&)$$

$$\frac{\begin{array}{c} sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \\ \wedge\ (\mathrm{N}(T_1) \wedge (sym(T_2) \neq \varnothing) \Rightarrow \mathrm{N}(U_1)) \\ \wedge\ (\mathrm{N}(T_2) \wedge (sym(T_1) \neq \varnothing) \Rightarrow \mathrm{N}(U_2)) \\ T_1 \leq_k U_1 \wedge T_2 \leq_k U_2 \end{array}}{T_1 \cdot T_2 \leq_k U_1 \& U_2} \quad (\text{DIVIDE} \cdot \&)$$

$$\frac{\begin{array}{c} sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \\ \wedge\ (\mathrm{N}(T_1) \wedge (sym(T_2) \neq \varnothing) \Rightarrow \mathrm{N}(U_1)) \\ \wedge\ (\mathrm{N}(T_2) \wedge (sym(T_1) \neq \varnothing) \Rightarrow \mathrm{N}(U_2)) \\ T_1 \leq_k U_1 \wedge T_2 \leq_k U_2 \end{array}}{T_1 \cdot T_2 \leq_k U_1 \cdot U_2} \quad (\text{DIVIDE} \cdot \cdot)$$

$$\frac{\begin{array}{c} sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \\ \wedge \mathrm{N}(U_1) \wedge \mathrm{N}(U_2) \\ T_1 \leq_k U_1 \wedge T_2 \leq_k U_2 \end{array}}{T_1 + T_2 \leq_k U_1 \& U_2} \quad (\text{NDIVIDE}{+}\&)$$

$$\frac{\begin{array}{c} sym(T_1) \subseteq sym(U_1) \wedge sym(T_2) \subseteq sym(U_2) \\ \wedge \mathrm{N}(U_1) \wedge \mathrm{N}(U_2) \\ T_1 \leq_k U_1 \wedge T_2 \leq_k U_2 \end{array}}{T_1 + T_2 \leq_k U_1 \cdot U_2} \quad (\text{NDIVIDE}{+}\cdot)$$

$$\frac{sym(T) \subseteq sym(U_1) \quad T \leq_k U_1}{T \leq_k U_1 + U_2} \quad (\text{FOCUS}{+})$$

$$\frac{sym(T) \subseteq sym(U_1) \quad T \leq_k U_1 \ \wedge\ \mathrm{N}(U_2)}{T \leq_k U_1 \& U_2} \quad (\text{NFOCUS}\&)$$

$$\frac{sym(T) \subseteq sym(U_1) \quad T \leq_k U_1 \ \wedge\ \mathrm{N}(U_2)}{T \leq_k U_1 \cdot U_2} \quad (\text{NFOCUS} \cdot)$$

TABLE 4.1: The structural rules

CHECK(T,U)

```
1  PREPROCESS(T,U)
2  push (T,U) in todo
3  while (todo ≠ ∅)
4      do
5          pick (T,U) from todo
6          if (∃ r such that Cond_r(T,U))
7            then push T_1^r ≤ U_1^r, ..., T_n^r ≤ U_n^r in todo
8            else if (not ORACLE(T,U))
9                    then return false
10 return true
```

The following theorems specify some sufficient conditions about $Cond_r(T,U)$ which guarantee that the algorithm is correct and is linear$^{\text{ORACLE}}$, meaning that it runs in linear time, apart from the time spent by ORACLE.

Theorem 4.5 (Correctness) *The structural algorithm is correct if, for any rule r and any pair of types T and U, the following holds.*

$$Cond_r(T,U) \Rightarrow (T \leq U \Leftrightarrow T_1^r \leq U_1^r \wedge \ldots \wedge T_n^r \leq U_n^r)$$

Theorem 4.6 (Linearity$^{\text{ORACLE}}$) *The structural algorithm is linear$^{\text{ORACLE}}$ provided that:*

- PREPROCESS(T,U) *is in $O(|T|+|U|)$;*
- *every rule consumes some input, i.e., for any rule r an integer $k_r > 0$ exists such that, for any pair of types T, U:*

$$(|T|+|U|) - (|T_1^r|+|U_1^r|+\ldots+|T_n^r|+|U_n^r|) \geq k_r$$

- *the test "find r such that $Cond_r(T,U)$" runs in time $O(|T|+|U|)$ when is negative, and in time $O(k_r)$ when it finds the rule r.*

Correctness and linearity

We do not report here about details of correctness and linearity proofs, the reader can refer to [CGPS09] for details, we only give some hints about the main ingredients.

Concerning correctness of presented subtype rules, this consists of proving that each rule corresponds to a double implication.

Proof of linearity is based on the fact that rule selection can be performed in constant time. Applicability conditions are a combination of the following components:

1. pattern matching, such as $T = T_1 \circledast T_2$
2. boolean combination of nullability and symbol emptiness, such as $\mathrm{N}(T_1) \wedge (sym(T_2) \neq \varnothing) \Rightarrow \mathrm{N}(U_1)$
3. symbol set inclusion, such as $sym(T_1) \subseteq sym(U_1)$

Component (1) is obviously in $O(1)$. A linear time bottom-up traversal can be used to decorate each node of T and U with attributes recording its nullability and the emptiness of symbol set, hence solving component (2). Component (3) requires a bit more of work, and it relies on a kind of decoration of both types T and U aiming at relating sub expression T' of T with the smallest sub-expression U' in U such that $sym(T') \subseteq sym(U')$. This can be obtained by a linear bottom-up parsing of the parse tress of T and U, and once available it allows to check symbol set inclusion in constant time.

Beyond binary types

The algorithm as presented embodies the main ideas behind our structural approach. However it is quite limited because it ignores basic commutativity and associativity properties of type operators. For example, it would fail on all of the following examples.

$a+b \leq b+a$

$a+b+a \leq b+a+c$

$a\cdot(b\cdot c) \leq (a\cdot b)\cdot c$

$a\&(b\&c) \leq (c\&b)\&a$

$a+(b+c) \leq b?\&((a?+d)+c)$

The first example shows that commutativity should be taken care of, and the second one elaborates a bit on this. The third example illustrates associativity. The fourth example shows that the simple approach of normalizing how operators are associated is not sufficient, because associativity and commutativity should be treated together. We solve this issue by adopting a flat version of all type operators, where every operator has an arbitrary number of arguments. Flattening solves associativity, but leaves commutativity open; we may solve this by reordering all addends alphabetically, but that would require more than linear time. Moreover, the last example shows that flattening and then reordering is not enough: since the product of nullable factors is a supertype of union, one would need to consider some pairs of operators together.

The approach we presented in [CGPS09] solves all these issues. First of all, we generalize all binary operators to their n-ary version, and we preprocess the types, in linear time, to collapse all consecutive application of the same binary operator into one application of an n-ary operator. Second, when applying a *divide* rule to a pair of types $\circledast(T_1,\ldots,T_n)$ and $\circledast(U_1,\ldots,U_m)$, we find, for each T_i, the minimum subterm U'_j of some U_j that contains all of symbols of T_i — that is, $U{\downarrow_{sym(T_i)}}$ — and we recur on the pair (T_i,U'_j) rather than on (T_i,U_j), and this solves the issue presented in the fifth example. By recurring on $(T_i,U{\downarrow_{sym(T_i)}})$, we merge the effect of the *divide* rule with that of the *focus* rules, and avoid two separate sets of rules.

We do not report here details about the rules, and refer the reader to [CGPS09] for a detailed and formal presentation.

Test results

As for any improvement, it is mandatory to show that the "optimized" algorithm is more efficient than the original one and that its applicability conditions can be easily satisfied, so to justify its implementation.

In [CGPS09] we report results about extensive experiments. Starting from the observation in [GMN07] that most human designed XML types are in *conjunctive normal form*, where each factor has the form $(a_1+\ldots+a_k)$, $(a_1+\ldots+a_k)?$, $(a_1+\ldots+a_k)^*$, or $(a_1+\ldots+a_k)^+$, we focused our experiments on CNF types and compared the performance of the structural algorithm with that of the quadratic algorithm on the four main kinds of factors.

Both the structural algorithm and the quadratic algorithm presented in the previous chapter have been implemented in Java 1.5 and all experiments were performed on a 2.16 Ghz Intel Core 2 Duo machine (3 GB main memory) running Mac OSX 10.5.7.

As already stated, in our experiments we evaluate the performance of our algorithm on CNF types, with four main categories of factors: $(a_1+\ldots+a_k)$, $(a_1+\ldots+a_k)?$, $(a_1+\ldots+a_k)^*$, and $(a_1+\ldots+a_k)^+$. For the sake of completeness, we also evaluate our algorithm on a DNF types scenario, where types are in *disjunctive normal form* (e.g., the subtype and the supertype are a union of products). In the supertype we impose the conflict freedom constraint, hence terminal symbols are unique and counting is applied only to terminal symbols, while these restrictions are relaxed in the subtype, which can be any legal type. In our experiments we compared the performance of the

structural algorithm with that of the plain mixed algorithm of [CGS09a]; in particular, we evaluate the scalability of the algorithms by increasing the number of addenda in each factor of both the supertype and the subtype from 10 to 100. To make the experiments even more realistic and test the flat algorithm, the supertype contains a 20% of randomly distributed labels. We only generated pairs of types which satisfy the subtype test, since this is the dominating situation when the algorithm is run by a compiler.

Test results reported in [CGPS09] show that the structural algorithm significantly outperforms the plain one in most of considered cases.

3 Conclusion

In the first part of this chapter, we presented an algorithm to check subtyping among EREs types with the only restriction that the supertype must be conflict-free, as it commonly happens while typechecking XML programs. This algorithm has quadratic complexity, both in the best and worst cases, it strongly exploits the conflict-free restriction over the supertype, but does not exploit any structural similarities between the subtype and the supertype to further accelerate inclusion checking.

In the second part we have presented a more efficient algorithm, still dealing with the kind of mixed comparisons considered in the first part, but which also exploits possible structural similarities between the types being compared. The new algorithm proceeds in a top-down fashion, and is based on a set of structural subtyping rules, that are applied whenever a structural similarity is detected; when these similarity conditions are not satisfied, the algorithm just resorts to the quadratic algorithm.

Sommaire

CHAPTER 5

WORK IN PROGRESS AND PERSPECTIVES

This chapter discusses works in progress and future directions related to topics dealt with in this Thesis.

1 Work in Progress

Schemas for Detecting XML query-update independence

A query and an update are independent when the query result is not affected by update execution, on any possible input database. Detecting query-update independence is of crucial importance in many contexts. It is crucial to minimize view re-materialization after updates, when the view is defined by a query; it is crucial to ensure isolation, when queries and updates are executed concurrently; as outlined in [BC09], it is also crucial to enforce access control policies, when the query is used to define the part of the database that must not be changed by a user update.

In all these contexts, benefits are amplified when query-update independence can be checked statically. In order to be useful, every static analysis technique must be sound: if query-update independence is statically detected, then independence does hold. The inverse implication (completeness) can not be ensured in the general case, since static independence detection is undecidable (see [BC09]). This means that if a static analyzer is used, for instance, in a view maintenance system, sometimes views are re-materialized after updates even if not needed, because the analysis has not been smart enough to statically detect a view-update independence. Useless view re-materialization frequently occurs if a static analyzer with low precision is adopted. This can lead to great waste of time, since view materialization cost can be proportional to the database size.

Schema-based detection of XML query-update independence has been recently investigated. The state of the art technique has been presented in [BC09]. This technique infers from the schema the set of node types traversed by the query, and the set of node types impacted by the update. The

query and the update are then deemed as independent if the two sets does not overlap. This technique is effective since the static analysis: i) is able to manage a wide class of XQuery queries and updates, ii) can be performed in a negligible time, and iii) as a consequence, even on small documents it can avoid expensive query re-computation when independence wrt an update is detected. However, the technique has some weaknesses. As illustrated in [BC09], in some cases independence is not detected due to the nature of the proposed type inference rules, and in particular by some kind of over approximation made by some of them.

For example, this technique can not detect independence between the query $Q_1 = //a//c$ and $U_1 = delete\ //b//c$, when the schema says that c descendants of b nodes are never descendants of a nodes as well. This is because the type inference technique proposed in [BC09] infers the type $\{c\}$ for the query path and the update path, without considering contextual information about the inferred types. Since the query and update types overlap, independence is wrongly excluded. Also, the technique is likely to not detect independence, when XPath axes requiring ancestor or descendant navigation are used.

Another kind of low precision of this technique is independent from the way XPath axes are typed. Consider $Q_2 = //title$ and $U_2 = \texttt{for}\ x\ \texttt{in}\ //book\ \texttt{return insert} < author/ > \texttt{into}\ x$, over data typed by the well known bibliographic DTD used in the XQuery Use Cases [CFF+07]. The technique proposed in [BC09] infers $\{book\}$ as the set of impacted types of U, while $\{bib, book, title\}$ as the set of types traced by the query. Since the two sets share the type *book*, the system does not detect independence, while it should.

In none of the above mentioned cases, independence can be detected by techniques not using schema information. This is the case for the path-based approach proposed by in [GRS08], which deals with the problem of update-commutativity detection, and which can be directly extended for detecting query-update independence. The same holds for the recent destabilizers-based approach proposed in [BC10]. Both approaches do not consider/use schema information, and as a consequence deem paths like $//a//c$ and $//b//c$ ($Q_1 - U_1$ example) as overlapping, since, for instance, documents matching the path $/a/b/c$ match both paths; a similar reasoning holds for $//title$ and $//book$ ($Q_2 - U_2$ example), since schema constraints are not considered.

Contributions

In the context of Federico Ulliana PhD thesis we are working on a novel schema-based approach for detecting XML query-update independence. Differently from traditional type systems for XQuery [BC09, Che08a, CGMS06], our system is able to infer sequences of labels (hereafter called *chains*) that are vertically navigated in a schema instance by query and update paths. More precisely, for each node that can be selected by a query/update path in a schema instance, the system infers a chain recording: a) all labels that are encountered from the root to the selected node, and b) the order in which these labels are traversed.

The contextual and ordering information respectively provided by a) and b) is at the basis of an extremely precise static independence analysis. For instance, by considering the simple schema $r \leftarrow (a|b)*,\ a \leftarrow c,\ b \leftarrow c$, for the Q_1 path $//a//c$ we infer the chain *r.a.c*, while for the U_1 path $//b//c$ we infer the chain *r.b.c*. Disjointness of these two chains can be simply checked, thus allowing, differently from the existing approaches [BC09, BC10], to detect independence for the $Q_1 - U_1$ pair. For the $Q_2 - U_2$ the query chain *bib.book.title* and the update chain *bib.book.author* are inferred from XQuery Use Cases DTD [CFF+07]; as these two chains diverges after the *book* symbol, we can conclude independence.

Our independence analysis technique is based on chain inference rules able to deal with all XPath axes. The resulting technique ensures all the advantages and the precision level of both [BC09] and [GRS08], while it improves on precision to a large extend: it enables to detect independence for difficult cases, like the ones previously illustrated, for which [BC09] and/or [GRS08] fail.

A key feature of our technique concerns the way recursive DTDs are handled. These DTDs

require special care in order to avoid inferring infinite sets of chains when XPath expressions use recursive axes (e.g. descendant and following). We show how a finite upper bound, on the number of chains to be inferred, can be determined in terms of structural properties of the query and the update.

Some preliminary results about this work have been published in [BCU10b], while more mature results are considered for future submissions, and are described below.

- We provide a chain inference system able to infer a set of chains from a DTD and XQuery query/update. We proved that chain inference soundly approximates the set of chains a query/update has to traverse in the DTD instances in order to compute the result. We then provide a notion of chain-based independence and prove it to be sound wrt the semantics notion of query-update independence.

- In the presence of recursive schemas, chain-based independence analysis may involve an infinite number of chains. We showed that the analysis can be carried out by restricting on a finite subset of the possibly infinite sets of query and update chains. We prove that the resulting *finite* analysis turns out to be equivalent to the *infinite* analysis. This is reminiscent of the well known Finite Model Property technique used in the context of finite model theory [P. 01].

- We have performed extensive tests by using a Java implementation of the finite analysis. We used the XMark testbed used in [BC09]. Concerning precision, obtained test results show that our technique ensures sensible improvements on [BC09]. Test results also show that sensible improvements in terms of time savings can be ensured by avoiding re-evaluation of queries deemed as independent of an update.

We are currently investigating optimizations for a succinct representation of inferred chains. Obtained results, highlighted that in many cases inferred chains share common prefixes and/or suffixes. While redundancy of prefixes can be easily dealt with by using a tree-based representation of inferred chains, controlling the redundancy of suffixes is subtle and more difficult to deal with.

Partitioning and Projecting XML Data

As we have seen, XML projection is a well established technique allowing main-memory XML query engines to query very large documents. These projection techniques are quite effective in a wide class of cases, but still fail when the projected documents are too big to be loaded in main memory. This can happen in two basic scenarios. First, when the query traverses a large part of the input documents, projection may become ineffective as the projected documents may almost coincide with the original ones; for instance, this can be the case for full-text search queries, or queries performing content-preserving transformation of a large fragment of the input. Furthermore, when working on very large documents, e.g., the XML dump of Wikipedia, no projection technique is currently able to trim the input documents to fit the size of the available main memory.

To overcome these limitations, in the context of Noor Malla PhD thesis, we are investigating a new projection technique, based on a path-based approach, so that it can be used even in the absence of a schema.

The technique relies on the observation that, in many practical cases, queries first select a sequence of nodes by means of a subquery (e.g, an XPath expression), and then iterate on this sequence to do some operations on the subtrees rooted at nodes in the sequence. We dub such queries as *iterative* queries

In this work we are devising novel projection techniques that deal with iterative queries, and improve the scalability of existing main memory engines on this class of queries.

To deal with the cases where the size of projection is likely to exceed the maximal size that a main-memory XML processor can manage, we provide a static analysis technique, based on path

analysis, allowing one to recognize when Q is iterative, and to infer path information to partition the input document D in several partition D_1, ..., D_n, such that $Q(D)$ is equal to the concatenation $Q(D_1),\ldots,Q(Q_n)$. Inferred path information is used for both partitioning and projecting, so that each partition is guaranteed to contain only information strictly necessary for processing Q. The maximal size of each partition is determined in terms of the particular main-memory engine.

As a second contribution, we extend the above technique to the case where a document has to be partitioned and projected in order to be queried by several queries taking part of a workload.

Both techniques have been formalized and implemented. Extensive test results on XMark documents whose size ranges from to 1GB to 5GB have shown that: i) for most main-memory query engines standard projection fails in many cases either by considering XMark queries or other practical relevant queries; ii) in the cases where standard projection work, even if we apply partitioning plus projection execution time does not increase, and iii) in cases involving iterative queries and where standard-projection fails, out method scales beautifully, allowing to process until 5GB documents (actually, since partitioning and results composition can be done in streaming, there is no size upper-limit for iterative queries), iv) when projection o is performed for a workload of multiple queries, standard projection is more likely to fail, while our partitioning technique still scales up.

These results will be the subject of future submissions for conference publication. Currently, we are investigating applications/extensions m to XML updates. We aim at identifying a class of iterative updates, and devise a partitioning algorithm allowing one to update large documents, without the need of a merge operation (Chapter 2). Also, we plan to investigate how to exploit the potential parallelism inherent in our techniques: once partitions are created, a query/update can be run in parallel on all the partitions.

Compact Representation of Temporal XML Documents

The management of temporal data is a crucial issue in many applications such as finance, banking, travel reservations, geographical information systems etc. With the increasing use of XML for data exchange and representation, the issue of developing temporal extensions for XML is gaining importance.

Current work on temporal XML concentrate on time-stamp XML documents, a concrete model. Although many proposals have addressed the issue of querying time-stamp XML documents, there has been less in-depth investigation of how to efficiently build or maintain temporal XML documents, keeping track of data evolution over time.

To fill this gap, in the context of Amine Baazizi PhD thesis, we are investigating techniques ensuring compact representation of temporal XML documents under updates. Each time an update is performed on the current snapshot, the document resulting from the update is opportunely merged with the historical database, recording all changes.

Two merging techniques have been devise, the first one uses no information about the particular update, while the second one does, and also uses projection-based techniques to efficiently manage both the update and the merging process.

Both methods have been implemented, and tests have shown that the second method outperforms the first one in terms of compactness of the historical document.

A paper including these results has been recently accepted for publication [BBC11a].

We are currently working on extensions of these results. Namely, on more extensive tests, and on new merge/projection techniques in the presence of multiple updates from the n to $n+1$ instances.

XQuery Type-Checking

As we have seen, XQuery has been designed by World Wide Web Consortium (W3C) as a standard query language for XML. XQuery is a functional, Turing-complete, strongly typed language. A key feature of XQuery is its type system, and a formal specification is proposed by the W3C [DFF+10].

In XQuery any language expression is statically typed and its type is used during program type-checking, even though the programmer can disable this feature.

In XQuery, types of input data and functions are defined in terms of regular expression types, but it is quite easy to write queries that generate non-regular languages. As a consequence, any type system for XQuery has to rely on a *type inference* process that approximates the (possibly non-regular) output type of a query with a regular type. This approximation process, while mandatory and unavoidable, may significantly decrease the precision of the inferred types. This is the case of the W3C proposed type system, which relies on some over-approximating rules for expressions widely used in practice (e.g., `for`-iterations). Another source of undesired over-approximation is given by rules to type horizontal and upward XPath axes, for which the type *any* is always inferred.

It is a common folklore that W3C has sacrificed precision in favor of better complexity, and that the W3C typing algorithm runs in polynomial time. An alternative and more precise approach for typing XQuery has been proposed in [Col04, CGMS04] and used as a basis for other proposals [Che08a, BCCN06]. This type system, used in the μXQ language, has a more precise type inference, at the price of a potential exponential explosion of the query output type.

Though the two above mentioned approaches are relatively well known today by the database and programming language communities, a formal, rigorous, and complete analysis showing in which cases the two proposals differ in terms of precision and complexity for type inference, is still missing. Such formal analysis could have a practical relevance as well, since it would provide important information to implementation designers.

In a recent work we filled this gap by providing a first comparative analysis. Besides providing a clean and simple formalization of the main typing mechanisms of both approaches, we formally studied their complexity, showed in which cases the W3C excessively over-approximates inferred types, identified cases for which inference precision can be dramatically improved, and propose new type rules to better handle these cases. We also showed that, contrary to the common belief, the W3C type system may itself infer types of exponential size wrt the query and the input size.

A paper collecting these results has been recently accepted for conference publication [CS11].

2 Perspectives

Future research directions are described in the next sections.

Type-based debugging of XML transformations

One of the main use of subtype checking is in the checking of correctness of transformations from a schema S_1 to another schema S_2. In many cases the transformation is defined via an XQuery query Q, and its correctness is checked by first inferring a type S for Q results starting from the query itself and the input schema S_1, and then by checking the inclusion $S \leq S_2$. If this inclusion check fails then the transformation is deemed as incorrect and should be revised.

In many cases the transformation is made of complex query expressions, and in the case type checking fails the correction process is likely to be quite cumbersome. To facilitate this task new type-based techniques could be devised. First of all the process of inferring a query type S should be equipped with *query-provenance* mechanisms allowing to opportunely decorate each subexpression of the inferred schema S with the parts of the query this subexpression is inferred from. Secondly, the sub-typing checking algorithm should be instrumented with mechanisms able to precisely locate sub expressions of compared types responsible for the failure of the inclusion checking.

In the case of failure of the inclusion checking $S \leq S_2$, parts of the query Q that should be changed can be located by opportunely combining information inferred by the two above techniques. A similar technique can be devised to aid the debugging of corrupted schema-to-schema mappings (Chapter 3)

Another future direction in the context of query type-checking, concerns the problem of type inference for XQuery in the presence of interleaving and counting. This problem is mostly unexplored, and entails the study of interesting subproblems, like the analysis of approaches to ensure precise type inference at a reasonable time cost.

Projection for XML security

A nice and powerful type-based technique to define and enforce access control policies on XML documents typed by a DTD has been proposed in [FCG04]. This technique has the advantage that security policies are: defined by opportunely enriching the DTD, and enforced by rewriting each query by matching it wrt the enriched DTD, so that the rewritten query can be safely executed on the original document, without the need of materializing security views.

A weakness of this technique is that is server-oriented: if a client asks for the execution of a query then the document can not be sent to the client for query execution, due to obvious security issues. Also, this solution excludes view materialization which can be fruitful in some contexts. A more serious weakness is that the technique only deals with the forward fragment of XPath. Hence, the technique does not take into account queries expressed in full XPath or XQuery.

To overcome these limitation, we plan to rely on schema based XML projection. Actually XML projection can be naturally used for enforcing access control policies, under the assumption that parts that do not have to be accessed by a user are pruned out during projection. Starting from this fact, new projection techniques can be devised for enforcing security. This new technique has to be such that the projection can be quickly computed, and at the same time only contain information that can be seen by the user (in other words the projection can contain only information that a user can obtain by using a query). This requires a special handling of information specified in security policies.

Another interesting aspect connected to security, is the handling of updates. We plan to investigate projection-based techniques to enforce secure execution of updates. In particular, this requires a new notion of projector, and a new merge process (see Chapter 2).

Efficient large scale management of Web data

The last years have seen a high concentration of research activities around the design and development of systems that scale to data volumes typically found in Web search indexes, large scale warehouses, and scientific applications. The main approach is based on massive parallelization, exploiting large numbers of cheap computers, often exploiting multicore hardware.

In this context, new architectures and programming paradigms have been proposed in order o overcome limitations of traditional DBMS architectures, typically their missing scalability, elasticity and fault tolerance. Among these proposals the MapReduce paradigm [DG04] has emerged as an effective and simple model, according to which data manipulation programs are written as map and reduce functions, which process key/value pairs and can be executed in many data-parallel instances. However, several complex database operations can not be easily programmed by means of map and reduce functions. This is the case, in particular, of operations requiring multiple inputs, like joins. Also, current database solutions based on map-reduce lack a tight integration of indexing and storage, which prevents data access optimization, a crucial ingredient for efficient query processing.

In the Leo team (which I joined on February 2010) we have recently started research activities aiming at conceiving efficient algorithms for processing queries and updates on Web data. To this end we will leverage the Stratosphere[7] platform in order to take advantage of its PACT programming model [ABE+10, BEH+10]. The PACT model is a powerful extension of MapReduce. One of its main strengths consists in second-order functions that define properties on the input and output data

[7] http://stratosphere.eu/

of their associated first-order functions. This is at the basis of simple and highly parallelizable program specifications for complex database operations.

In a first step, we are interested in the use of PACT primitives for the efficient management of extremely large indexes over XML and RDF data. We will focus on indexes formed by (key, value) pairs, in order to cope with the PACT data model. We will devise several kind of indexes and define optimal strategies by means of the PACT programming model in order to efficiently access and update the indexes.

As a subsequent step, we plan to switch to the problem of generating efficient PACT workflows starting from XQuery/SPARQL queries. We will first rely on available PACT primitives, and then, based on what learnt in the above depicted first step, we will propose extensions to the PACT programming model with new primitives that better fit with the generation of efficient workflows for XML/RDF management.

This research plan is part of a wider research project recently accepted by the European Institute of Innovation & Technology (EIT)[8]. For this project, that involves several european partners, I am the local scientific coordinator for the University of Paris Sud partner.

[8] http://eit.ictlabs.eu/

BIBLIOGRAPHY

[ABE+10] Alexander ALEXANDROV, Dominic BATTRÉ, Stephan EWEN, Max HEIMEL, Fabian HUESKE, Odej KAO, Volker MARKL, Erik NIJKAMP, et Daniel WARNEKE. « Massively Parallel Data Analysis with PACTs on Nephele ». *PVLDB*, 3(2):1625–1628, 2010. 74

[ABS99] Serge ABITEBOUL, Peter BUNEMAN, et Dan SUCIU. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999. 3

[AL05a] Marcelo ARENAS et Leonid LIBKIN. « XML Data Exchange: Consistency and Query Answering. ». Dans *PODS*, 2005. 5

[AL05b] Marcelo ARENAS et Leonid LIBKIN. « XML data exchange: consistency and query answering ». Dans Li [Li05], pages 13–24. 31

[AMN+01] Noga ALON, Tova MILO, Frank NEVEN, Dan SUCIU, et Victor VIANU. « XML with Data Values: Typechecking Revisited ». Dans *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2001. 3

[ATV08] Bogdan ALEXE, Wang Chiew TAN, et Yannis VELEGRAKIS. « STBenchmark: towards a benchmark for mapping systems ». *PVLDB*, 1(1):230–244, 2008. 46

[BBC11a] Mohamed-Amine BAAZIZI, Nicole BIDOIT, et Dario COLAZZO. « Efficient Encoding of Temporal XML Documents. ». Dans *International Symposium on Temporal Representation and Reasoning (TIME)*, 2011. . . . . 8, 72

[BBC+11b] Mohamed Amine BAAZIZI, Nicole BIDOIT, Dario COLAZZO, Noor MALLA, et Marina SAHAKYAN. « Projection for XML update optimization ». Dans *EDBT*, pages 307–318. ACM, 2011. . . . . 5, 7, 9, 27, 28, 30

[BBFV05] Michael BENEDIKT, Angela BONIFATI, Sergio FLESCA, et Avinash VYAS. « Verification of Tree Updates for Optimization ». Dans *CAV*, pages 379–393, 2005. 22

[BC07] Nicole BIDOIT et Dario COLAZZO. « Testing XML constraint satisfiability ». *Electr. Notes Theor. Comput. Sci.*, 174(6):45–61, 2007. 7

[BC09] Michael BENEDIKT et James CHENEY. « Schema-Based Independence Analysis for XML Updates ». *VLDB*, pages 61–72, 2009. . . . . 3, 22, 69, 70, 71

[BC10] Michael BENEDIKT et James CHENEY. « Destabilizers and Independence of XML Updates ». *PVLDB*, 3(1):906–917, 2010. 70

[BCCM09] Michele BUGLIESI, Dario COLAZZO, Silvia CRAFA, et Damiano MACEDONIO. « A type system for Discretionary Access Control ». *Mathematical Structures in Computer Science*, 19(4):839–875, 2009. 7

[BCCN06] Véronique BENZAKEN, Giuseppe CASTAGNA, Dario COLAZZO, et Kim NGUYEN. « Type-Based XML Projection ». Dans *International Conference on Very Large Data Bases (VLDB)*, 2006. ..... 3, 5, 8, 9, 10, 15, 21, 22, 26, 30, 73

[BCCN11] Véronique BENZAKEN, Giuseppe CASTAGNA, Dario COLAZZO, et Kim NGUYEN. « Optimizing XML querying using type-based document projection ». *CoRR*, abs/1104.2079, 2011. ..... 4, 16, 21, 30

[BCF+02] Michael BENEDIKT, Chee Yong CHAN, Wenfei FAN, Rajeev RASTOGI, Shihui ZHENG, et Aoying ZHOU. « DTD-Directed Publishing with Attribute Translation Grammars ». Dans *International Conference on Very Large Data Bases*, 2002. 3

[BCL+05] S. BRESSAN, B. CATANIA, Z. LACROIX, Y-G LI, et A. MADDALENA. « Accelerating queries by pruning XML documents. ». *Data Knowl. Eng.*, 54(2):211–240, 2005. 11, 12

[BCMS09a] Nicole BIDOIT, Dario COLAZZO, Noor MALLA, et Marina SAHAKYAN. « Optimisation de Mises a jour XML par typage et projection. ». Dans *25èmes Journées Bases de Données Avancées*, Lille France, 2009. ..... 5, 9

[BCMS09b] Nicole BIDOIT, Dario COLAZZO, Noor MALLA, et Marina SAHAKYAN. « Projection based optimization for XML updates. ». Dans *1st International Workshop on Schema Languages for XML (X-Schemas) year = 2009*, Riga Lituanie, 2009. ..... 5, 7, 9

[BCU10a] Nicole BIDOIT, Dario COLAZZO, et Federico ULLIANA. « Detecting XML Query-Update Independence. ». Dans *25èmes Journées Bases de Données Avancées, (BDA)*, 2010. 8

[BCU10b] Nicole BIDOIT, Dario COLAZZO, et Federico ULLIANA. « Detecting XML query-update independence ». Dans *26emes Journees Bases de Donnees Avancees*, Toulouse France, 2010. 71

[BEH+10] Dominic BATTRÉ, Stephan EWEN, Fabian HUESKE, Odej KAO, Volker MARKL, et Daniel WARNEKE. « Nephele/PACTs: a programming model and execution framework for web-scale analytical processing ». Dans *ACM Symposium on Cloud Computing (SoCC)*, 2010. 74

[BFG08] Michael BENEDIKT, Wenfei FAN, et Floris GEERTS. « XPath satisfiability in the presence of DTDs ». *J. ACM*, 55(2), 2008. 17

[BJH+05] Klemens BÖHM, Christian S. JENSEN, Laura M. HAAS, Martin L. KERSTEN, P. LARSON, et Beng Chin OOI, éditeurs. *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. ACM, 2005. ..... 83, 85

[Bla00] Patrick BLACKBURN. « Representation, Reasoning, and Relational Structures: a Hybrid Logic Manifesto ». *Logic Journal of the IGPL*, 8(3), 2000. 7

[BLS06] Denilson BARBOSA, Gregory LEIGHTON, et Andrew SMITH. « Efficient Incremental Validation of XML Documents After Composite Updates ». Dans *Database and XML Technologies, 4th International XML Database Symposium,XSym 2006, Seoul, Korea, September 10-11, 2006, Proceedings*, volume 4156 de *LNCS*, pages 107–121. Springer, 2006. 50

[BM04] Paul V. BIRON et Ashok MALHOTRA. « XML Schema Part 2: Datatypes Second Edition ». Rapport technique, World Wide Web Consortium, Oct 2004. W3C Recommendation. 3

[BMP+08] Angela BONIFATI, Giansalvatore MECCA, Alessandro PAPPALARDO, Salvatore RAUNICH, et Gianvito SUMMA. « Schema mapping verification: the spicy way ». Dans Alfons KEMPER, Patrick VALDURIEZ, Noureddine MOUADDIB, Jens TEUBNER, Mokrane BOUZEGHOUB, Volker MARKL, Laurent AMSALEG, et Ioana MANOLESCU, éditeurs, *EDBT*, volume 261 de *ACM International Conference Proceeding Series*, pages 85–96. ACM, 2008. 46

[BNdB04] Geert Jan BEX, Frank NEVEN, et Jan Van den BUSSCHE. « DTDs versus XML Schema: A Practical Study ». Dans *Proceedings of the Seventh International Workshop on the Web and Databases, WebDB 2004, June 17-18, 2004, Maison de la Chimie, Paris, France, Colocated with ACM SIGMOD/PODS 2004*, pages 79–84, 2004. . . . 6, 17, 43

[BNST06] Geert Jan BEX, Frank NEVEN, Thomas SCHWENTICK, et Karl TUYLS. « Inference of Concise DTDs from XML Data ». Dans Dayal et al. [DWL+06], pages 115–126. . 6, 50, 56

[BP99] Peter BUNEMAN et Benjamin C. PIERCE. « Union Types for Semistructured Data ». Dans Richard C. H. CONNOR et Alberto O. MENDELZON, éditeurs, *DBPL*, volume 1949 de *Lecture Notes in Computer Science*, pages 184–207. Springer, 1999. 47

[BPC+06] Tim BRAY, Jean PAOLI, Don CHAMBERLIN, C. M. SPERBERG-MCQUEEN, Eve MALER, et Francois YERGEAU. « Extensible Markup Language (XML) 1.0 (Fourth Edition) ». Rapport technique, World Wide Web Consortium, août 2006. W3C Recommendation. 3

[BZH11] Schema Matching and Mapping. Springher, 2011. 3

[CFF+07] Don CHAMBERLIN, Peter FANKHAUSER, Daniela FLORESCU, Massimo MARCHIORI, et Jonathan ROBIE. « XML Query Use Cases ». Rapport technique, World Wide Web Consortium, mars 2007. W3C Working Group Note. 13, 70

[CGG00] Luca CARDELLI, Giorgio GHELLI, et Andrew D. GORDON. « Secrecy and Group Creation ». Dans *CONCUR 2000 - Concurrency Theory, 11th International Conference*, 2000. 7

[CGLN09] Jérôme CHAMPAVÈRE, Rémi GILLERON, Aurélien LEMAY, et Joachim NIEHREN. « Efficient inclusion checking for deterministic tree automata and XML Schemas ». *Inf. Comput.*, 207(11):1181–1208, 2009. 52

[CGM+10] Dario COLAZZO, Giovanna GUERRINI, Marco MESITI, Barbara OLIBONI, et Emmanuel WALLER. Document and Schema XML Updates. Dans Changqing LI et Tok Wang LING, éditeurs, *Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies*. IGI Global, 2010. 7

[CGMS04] D. COLAZZO, G. GHELLI, P. MANGHI, et C. SARTIANI. « Types for Path Correctness for XML Queries ». Dans *ICFP '04, 9th ACM Int. Conf. on Functional Programming*, 2004. . . . . 4, 14, 17, 33, 37, 39, 40, 45, 47, 73

[CGMS06] Dario COLAZZO, Giorgio GHELLI, Paolo MANGHI, et Carlo SARTIANI. « Static analysis for path correctness of XML queries ». *J. Funct. Program.*, 16(4-5):621–661, 2006. . . . . 3, 14, 70

[CGPS09] Dario COLAZZO, Giorgio GHELLI, Luca PARDINI, et Carlo SARTIANI. « Linear inclusion for XML regular expression types ». Dans *ACM Conference on Information and Knowledge Management (CIKM)*, 2009. . . . . 6, 8, 64, 66, 67, 68

[CGS09a] Dario COLAZZO, Giorgio GHELLI, et Carlo SARTIANI. « Efficient asymmetric inclusion between regular expression types ». Dans *Database Theory - ICDT, International Conference*, 2009. . . . . 6, 8, 51, 58, 59, 64, 68

[CGS09b] Dario COLAZZO, Giorgio GHELLI, et Carlo SARTIANI. « Efficient inclusion for a class of XML types with interleaving and counting ». *Inf. Syst.*, 34(7):643–656, 2009. 6, 8, 49, 50, 52, 53, 55, 56, 57, 58, 59

[CGS11a] Dario COLAZZO, Giorgio GHELLI, et Carlo SARTIANI. « Asymmetric Inclusion Between Regular Expression Types With Interleaving And Counting ». Dans *Technical Report*, 2011. . . . . 6, 51, 58, 59, 60

[CGS11b] Dario COLAZZO, Giorgio GHELLI, et Carlo SARTIANI. « Schemas for Efficient and safe XML processing ». Dans *24th International Conference on Data Engineering, ICDE*. IEEE, 2011. 7

[Che08a] James CHENEY. « FLUX: functional updates for XML ». Dans *ACM SIGPLAN international conference on Functional programming (ICFP)*, 2008. . . . . 3, 70, 73

[Che08b] James CHENEY. « Regular Expression Subtyping for XML Query and Update Languages ». Dans *Programming Languages and Systems, th European Symposium on Programming, ESOP*, 2008. . . . . 33, 45

[Cho02] Byron CHOI. « What are real DTDs like? ». Dans *WebDB*, pages 43–48, 2002. . . 17, 43, 50

[Col04] D. COLAZZO. « *Path Correctness for XML Queries: Characterization and Static Type Checking* ». PhD thesis, Dip. di Informatica, Università di Pisa, 2004. . . 4, 5, 14, 17, 33, 37, 39, 45, 73

[CPR05] Roberto Di COSMO, François POTTIER, et Didier RÉMY. « Subtyping Recursive Types Modulo Associative Commutative Products. ». Dans Pawel URZYCZYN, éditeur, *TLCA*, volume 3461 de *Lecture Notes in Computer Science*, pages 179–193. Springer, 2005. 45

[CS05a] D. COLAZZO et C. SARTIANI. « Typechecking Queries for Maintaining Schema Mappings in XML P2P Databases ». Dans *Proc. of (PLAN-X)*, 2005. . . . . 37, 47

[CS05b] Dario COLAZZO et Carlo SARTIANI. « Mapping Maintenance in XML P2P Databases ». Dans *International Symposium on Database Programming Languages*, pages 74–89, 2005. . . . . 6, 8, 32

[CS06] Dario COLAZZO et Carlo SARTIANI. « An efficient algorithm for XML type projection ». Dans *International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 51–60, 2006. . . . . 6, 8, 32

[CS07] Dario COLAZZO et Carlo SARTIANI. « Efficient Subtyping for Unordered XML Types ». Rapport technique, Dipartimento di Informatica - Università di Pisa, 2007. 52

[CS09] Dario COLAZZO et Carlo SARTIANI. « Detection of corrupted schema mappings in XML data integration systems ». *ACM Trans. Internet Techn.*, 9(4), 2009. 3, 6, 8, 32, 36, 43, 44, 45

[CS11] Dario COLAZZO et Carlo SARTIANI. « Precision and Complexity of XQuery Type Inference. ». Dans *International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2011. . . . . 8, 73

[CT06] Laura CHITICARIU et Wang Chiew TAN. « Debugging Schema Mappings with Routes. ». Dans Dayal et al. [DWL+06], pages 79–90. 47

[DFF+10] Denise DRAPER, Peter FANKHAUSER, Mary FERNANDEZ, Ashok MALHOTRA, Kristoffer ROSE, Michael RYS, Jérôme SIMÉON, et Philip WADLER. « XQuery 1.0 and XPath 2.0 Formal Semantics ». Rapport technique, World Wide Web Consortium, 2010. . . . . 14, 32, 72

[DG04] Jeffrey DEAN et Sanjay GHEMAWAT. « MapReduce: Simplified Data Processing on Large Clusters ». Dans *OSDI*, pages 137–150, 2004. 74

[DWL+06] Umeshwar DAYAL, Kyu-Young WHANG, David B. LOMET, Gustavo ALONSO, Guy M. LOHMAN, Martin L. KERSTEN, Sang Kyun CHA, et Young-Kuk KIM, éditeurs. *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006. . . . . 79, 80

[DZLM04] Silvano DAL-ZILIO, Denis LUGIEZ, et Charles MEYSSONNIER. « A logic you can count on ». Dans *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004. 3

[exi] « eXist ». `http://exist.sourceforge.net/`. 4

[FB08] Wenfei FAN et Philip BOHANNON. « Information preserving XML schema embedding ». *ACM Trans. Database Syst.*, 33(1), 2008. 3

[FCB07] Wenfei FAN, Gao CONG, et Philip BOHANNON. « Querying xml with update syntax ». Dans *SIGMOD Conference*, 2007. 22

[FCG04] Wenfei FAN, Chee Yong CHAN, et Minos N. GAROFALAKIS. « Secure XML Querying with Security Views ». Dans *ACM SIGMOD International Conference on Management of Data*, 2004. 74

[Feg10] Leonidas FEGARAS. « A Schema-Based Translation of XQuery Updates ». Dans *XSym*, 2010. 22

[FKMT05] Ariel FUXMAN, Phokion G. KOLAITIS, Renée J. MILLER, et Wang Chiew TAN. « Peer data exchange ». Dans Li [Li05], pages 160–171. 32

[FLM99] Marc FRIEDMAN, Alon Y. LEVY, et Todd D. MILLSTEIN. « Navigational Plans for Data Integration ». Dans *Intelligent Information Integration*, volume 23 de *CEUR Workshop Proceedings*, 1999. 31

[FPS07] J. Nathan FOSTER, Benjamin C. PIERCE, et Alan SCHMITT. « A Logic Your Typechecker Can Count On: Unordered Tree Types in Practice ». Dans *Workshop on Programming Language Technologies for XML (PLAN-X), informal proceedings*, janvier 2007. 52

[Fra05] M. FRANCESCHET. « XPathMark - An XPath benchmark for XMark generated data ». Dans *XSym 2005, 3rd Int. XML Database Symposium*, LNCS n. 3671, 2005. . . 21

[FS98] Mary F. FERNANDEZ et Dan SUCIU. « Optimizing Regular Path Expressions Using Graph Schemas ». Dans *International Conference on Data Engineering (ICDE)*, 1998. 3

[FW04] David C. FALLSIDE et Priscilla WALMSLEY. « XML Schema Part 0: Primer – Second Edition », Oct 2004. W3C Recommendation. 49, 52

[FYL+09] Wenfei FAN, Jeffrey Xu YU, Jianzhong LI, Bolin DING, et Lu QIN. « Query translation from XPath to SQL in the presence of recursive DTDs ». *VLDB J.*, 18(4):857–883, 2009. 3

[gal] « Galax ». `http://www.galaxquery.org`. 4

[GCS07] Giorgio GHELLI, Dario COLAZZO, et Carlo SARTIANI. « Efficient Inclusion for a Class of XML Types with Interleaving and Counting ». Dans Marcelo ARENAS et Michael I. SCHWARTZBACH, éditeurs, *DBPL*, volume 4797 de *Lecture Notes in Computer Science*, pages 231–245. Springer, 2007. 6

[GCS08] Giorgio GHELLI, Dario COLAZZO, et Carlo SARTIANI. « Linear time membership in a class of regular expressions with interleaving and counting ». Dans James G. SHANAHAN, Sihem AMER-YAHIA, Ioana MANOLESCU, Yi ZHANG, David A. EVANS, Aleksander KOLCZ, Key-Sun CHOI, et Abdur CHOWDHURY, éditeurs, *CIKM*, pages 389–398. ACM, 2008. 7

[GGM09] Wouter GELADE, Marc GYSSENS, et Wim MARTENS. « Regular Expressions with Counting: Weak versus Strong Determinism ». Dans Rastislav KRÁLOVIC et Damian NIWINSKI, éditeurs, *MFCS*, volume 5734 de *Lecture Notes in Computer Science*, pages 369–381. Springer, 2009. 51

[GMN07] Wouter GELADE, Wim MARTENS, et Frank NEVEN. « Optimizing Schema Languages for XML: Numerical Constraints and Interleaving ». Dans Thomas SCHWENTICK et Dan SUCIU, éditeurs, *Proceedings of the 11th International Conference on Database Theory - ICDT 2007, Barcelona, Spain, January 10-12, 2007*, volume 4353 de *Lecture Notes in Computer Science*, pages 269–283. Springer, 2007. . . . . 6, 49, 52, 67

[Gol98] Andrew V. GOLDBERG. « Recent Developments in Maximum Flow Algorithms (Invited Lecture) ». Dans Stefan ARNBORG et Lars IVANSSON, éditeurs, *SWAT*, volume 1432 de *Lecture Notes in Computer Science*, pages 1–10. Springer, 1998. 45

[GRS07] Giorgio GHELLI, Kristoffer Høgsbro ROSE, et Jérôme SIMÉON. « Commutativity Analysis in XML Update Languages ». Dans *ICDT*, pages 374–388, 2007. 22

[GRS08] Giorgio GHELLI, Kristoffer Høgsbro ROSE, et Jérôme SIMÉON. « Commutativity analysis for XML updates ». *ACM Trans. Database Syst.*, 33(4), 2008. . . . . 22, 70

[GvKT03] Torsten GRUST, Maurice van KEULEN, et Jens TEUBNER. « Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps ». Dans *VLDB*, 2003. 4

[HHP+07] Mauricio A. HERNÁNDEZ, Howard HO, Lucian POPA, Ariel FUXMAN, Renée J. MILLER, Takeshi FUKUDA, et Paolo PAPOTTI. « Creating Nested Mappings with Clio ». Dans *ICDE*, pages 1487–1488. IEEE, 2007. 46

[Hid03] Jan HIDDERS. « Satisfiability of XPath Expressions ». Dans *DBPL*, 2003. 17

[HIMT03] Alon Y. HALEVY, Zachary G. IVES, Peter MORK, et Igor TATARINOV. « Piazza: data management infrastructure for semantic web applications ». Dans *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003*, pages 556–567. ACM, 2003. . . . . 5, 32, 35

[Hov10] Dag HOVLAND. « The Inclusion Problem for Regular Expressions ». Dans *Language and Automata Theory and Applications, 4th International Conference, LATA*, pages 309–320, 2010. 52

[HP03] Haruo HOSOYA et Benjamin C. PIERCE. « XDuce: A statically typed XML processing language. ». *ACM Trans. Internet Techn.*, 3(2):117–148, 2003. 32

[Huy85] Dung T. HUYNH. « The Complexity of Equivalence Problems for Commutative Grammars ». *Information and Control*, 66(1/2):103–121, 1985. . . . . 41, 42, 48

[Kil10] Pekka KILPELÄINEN. « Checking determinism of XML Schema content models in optimal time ». *Informat. Systems*, 2010. in press. 51

[Koz77] Dexter KOZEN. « Lower Bounds for Natural Proof Systems ». Dans *FOCS*, pages 254–266, 1977. 52

[KS01] Gabriel M. KUPER et Jérôme SIMÉON. « Subsumption for XML types. ». Dans Jan Van den BUSSCHE et Victor VIANU, éditeurs, *ICDT*, volume 1973 de *Lecture Notes in Computer Science*, pages 331–345. Springer, 2001. 47

[KT03] Pekka KILPELÄINEN et Rauno TUHKANEN. « Regular Expressions with Numerical Occurrence Indicators - preliminary results ». Dans Pekka KILPELÄINEN et Niina PÄIVINEN, éditeurs, *SPLST*, pages 163–173. University of Kuopio, Department of Computer Science, 2003. 52

[KT07] Pekka KILPELÄINEN et Rauno TUHKANEN. « One-unambiguity of regular expressions with numeric occurrence indicators ». *Inf. Comput.*, 205(6):890–916, 2007. 51, 52

[Kus00] Nicholas KUSHMERICK. « Wrapper verification. ». *World Wide Web*, 3(2):79–94, 2000. 47

[Li05] Chen LI, éditeur. *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*. ACM, 2005. . . . . 77, 81

[LMK03] Kristina LERMAN, Steven MINTON, et Craig A. KNOBLOCK. « Wrapper Maintenance: A Machine Learning Approach. ». *J. Artif. Intell. Res. (JAIR)*, 18:149–181, 2003. 47

[LMM00] D. LEE, M. MANI, et M. MURATA. « Reasoning about XML Schema Languages using Formal Language Theory ». Rapport technique, IBM Almaden Research, 2000. 10

[LRWZ04] Laks V. S. LAKSHMANAN, Ganesh RAMESH, Wendy Hui WANG, et Zheng (Jessica) ZHAO. « On Testing Satisfiability of Tree Pattern Queries ». Dans *VLDB*, 2004. . 17

[MAL+05] Robert MCCANN, Bedoor K. ALSHEBLI, Quoc LE, Hoa NGUYEN, Long VU, et An-Hai DOAN. « Mapping Maintenance for Data Integration Systems ». Dans Böhm et al. [BJH+05], pages 1018–1030. . . . . 31, 47

[Mar04] Maarten MARX. « XPath with Conditional Axis Relations ». Dans *EDBT*, 2004. . 17

[MH03] Jayant MADHAVAN et Alon Y. HALEVY. « Composing Mappings Among Data Sources. ». Dans Johann Christoph FREYTAG, Peter C. LOCKEMANN, Serge ABITEBOUL, Michael J. CAREY, Patricia G. SELINGER, et Andreas HEUER, éditeurs, *VLDB*, pages 572–583. Morgan Kaufmann, 2003. 34

[MLMK05] Makoto MURATA, Dongwon LEE, Murali MANI, et Kohsuke KAWAGUCHI. « Taxonomy of XML schema languages using formal language theory. ». *ACM Trans. Internet Techn.*, 5(4):660–704, 2005. 3

[MNSB06] Wim MARTENS, Frank NEVEN, Thomas SCHWENTICK, et Geert Jan BEX. « Expressiveness and complexity of XML Schema ». *ACM Trans. Database Syst.*, 31(3):770–813, 2006. 50

[MRB03] Sergey MELNIK, Erhard RAHM, et Philip A. BERNSTEIN. « Rondo: A Programming Platform for Generic Model Management ». Dans Alon Y. HALEVY, Zachary G. IVES, et AnHai DOAN, éditeurs, *SIGMOD Conference*, pages 193–204. ACM, 2003. 47

[MS94] Alain J. MAYER et Larry J. STOCKMEYER. « Word Problems — This Time with Interleaving ». *Inf. Comput.*, 115(2):293–311, 1994. . . . . 49, 51, 52

[MS03] A. MARIAN et J. SIMÉON. « Projecting XML Documents. ». Dans *VLDB '03*, pages 213–224, 2003. . . . . 11, 12, 21, 22, 38

[MSV03] Tova MILO, Dan SUCIU, et Victor VIANU. « Typechecking for XML transformers ». *J. Comput. Syst. Sci.*, 66(1):66–97, 2003. 17

[OMFB02] D. OLTEANU, H. MEUSS, T. FURCHE, et F. BRY. « XPath: Looking Forward ». Dans *Proc. EDBT Workshop (XMLDM)*, volume 2490 de *LNCS*, pages 109–127. Springer, 2002. 11

[P. 01] P. BLACKBURN AND M. RIJKE. *Modal Logic*. Cambridge University Press, 2001. 71

[PV] Yannis PAPAKONSTANTINOU et Victor VIANU. « Incremental Validation of XML Documents ». Dans *Database Theory - ICDT, International Conference, year = 2003*. 3

[PV00] Yannis PAPAKONSTANTINOU et Victor VIANU. « DTD Inference for Views of XML Data ». Dans *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2000. 3

[qiz] « QizX Free-Engine-3.0 ». `http://www.xmlmind.com/qizx/free_engine.html`. 4

[rel01] « RELAX NG Specification ». The Organization for the Advancement of Structured Information Standards [OASIS], 2001. Committee Specification 3 December 2001. 49, 51, 52

[saxa] « SAX ». http://www.saxproject.org/. 29

[saxb] « Saxon-EE ». `http://www.saxonica.com/`. 4

[SV02] Luc SEGOUFIN et Victor VIANU. « Validating Streaming XML Documents ». Dans *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2002. 3

[SWK+02] A. SCHMIDT, F. WAAS, M. L. KERSTEN, M. J. CAREY, I. MANOLESCU, et R. BUSSE. « XMark: A Benchmark for XML Data Management ». Dans *VLDB '02*, pages 974–985, 2002. 21

[Tat04] Igor TATARINOV. « *Semantic Data Sharing with a Peer Data Management System* ». PhD thesis, University of Washington, 2004. 33

[TBMM04] Henry S. THOMPSON, David BEECH, Murray MALONEY, et Noah MENDELSOHN. « XML Schema Part 1: Structures Second Edition ». Rapport technique, World Wide Web Consortium, Oct 2004. W3C Recommendation. 3, 51

[TH04] Igor TATARINOV et Alon Y. HALEVY. « Efficient Query Reformulation in Peer-Data Management Systems. ». Dans *SIGMOD Conference*, pages 539–550, 2004. . 33, 34

[Ull] Federico ULLIANA. « A Formal Study for a Type System for XQuery Optimization ». 5

[Ull88] Jeffrey D. ULLMAN. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988. 31

[Ull89] Jeffrey D. ULLMAN. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989. 31

[use] « XML Query Use Cases ». `http://www.w3.org/TR/xquery-use-cases/`. 17

[VMP04] Yannis VELEGRAKIS, Renée J. MILLER, et Lucian POPA. « Preserving mapping consistency under schema changes ». *VLDB J.*, 13(3):274–293, 2004. . . . . 46, 47

[xqua] « W3C XML Query (XQuery) ». `http://www.w3.org/XML/Query/`. 4

[XQub] « XQuery 1.0: An XML Query Language ». `http://www.w3.org/xquery`. 3, 32

[xquc] « XQuery 1.0 and XPath 2.0 Formal Semantics ». `http://www.w3.org/TR/xquery-semantics`. 20

[xup] « XQuery Update Facility 1.0 ». `http://www.w3.org/TR/2008/CR-xquery-update-10-20080801`. 4, 22

[YP05] Cong YU et Lucian POPA. « Semantic Adaptation of Schema Mappings when Schemas Evolve ». Dans Böhm et al. [BJH+05], pages 1006–1017. 47