Mapping Maintenance in XML P2P Databases*

Dario Colazzo[1] Carlo Sartiani[2]

[1] LRI - Université Paris Sud - France
dario.colazzo@lri.fr
[2] Dipartimento di Informatica - Università di Pisa - Italy
sartiani@di.unipi.it

Abstract. Unstructured p2p database systems are usually characterized by the presence of *schema mappings* among peers. In these systems, the detection of *corrupted* mappings is a key problem. A corrupted mapping fails in matching the target or the source schema, hence it is not able to transform data conforming to a schema $\mathcal{S}_i$ into data conforming to a schema $\mathcal{S}_j$, nor it can be used for effective query reformulation.
This paper describes a novel technique for maintaining mappings in XML p2p databases, based on a *semantic* notion of mapping correctness.

1 Introduction

The *peer-to-peer* computational model (p2p) is nowadays massively used for sharing and querying data dispersed over the Internet. Peer-to-peer data sharing systems can be classified in two main categories. *Structured* p2p systems [1] [2] distribute data across the network according to a hash function, so to form a distributed hash table (DHT); systems in this class allows for a fast retrieval of data ($O(logn)$, where n is the number of peers in the system), at the price of very limited query capabilities (key lookup queries and, in some systems, range queries). *Unstructured* systems, instead, leave peers free to manage their own data, and feature rich query languages like, for instance, XQuery [3]. Queries are executed by flooding the network and traversing the whole system.

Most unstructured p2p database systems (see [4], [5], and [6] for instance) are characterized by the presence of *schema mappings* among peers. A schema mapping (e.g., a set of Datalog rules) describes how to translate data conforming to a source schema $\mathcal{S}_i$ into data conforming to a target schema $\mathcal{S}_j$ (or to a projection of $\mathcal{S}_j$), and it can be used to reformulate, according to the *Global As View* (GAV) and *Local As View* (LAV) paradigms [7, 8], queries on $\mathcal{S}_i$ into queries over $\mathcal{S}_j$, and *vice versa*. Schema mappings, hence, allow the system to retrieve data that are semantically similar but described by different schemas.

A main problem in mapping-based systems is the maintenance of schema mappings, and, in particular, the detection of *corrupted* mappings. A corrupted mapping fails in matching the target or the source schema, hence it is not able to transform data conforming to a source schema $\mathcal{S}_i$ into data conforming to the target schema $\mathcal{S}_j$. The presence of such mappings can affect query processing: since queries are processed by flooding the network and by repeatedly applying reformulation steps, a corrupted mapping may

* Dario Colazzo was funded by the RNTL-GraphDuce project and by the ACI project "Transformation Languages for XML: Logics and Applications". Carlo Sartiani was funded by the FIRB GRID.IT project and by Microsoft Corporation under the BigTop project.

make the data of some peers unreachable; moreover, optimization techniques based on mapping pre-combination can be vanished by the presence of corrupted mappings.

Nowadays, mapping maintenance is performed manually by the site administrator, and quick responses to sudden mapping corruptions are not possible. To the best of our knowledge, the only proposed technique for automatically maintaining mappings, in the context of XML p2p database systems, has been described in [9]. This technique is based on the use of a type system capable of checking the correctness of a query, in a XQuery-like language [10], 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 [9] 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.

Our Contribution This paper describes a novel technique for maintaining mappings in XML p2p database systems. As for [9], the technique relies on a type system for an XML query language: while in [9] we exploited the type system to check for the correctness of a query wrt a given schema, in the spirit of [10], in this paper we develop a slightly different type system focused on type inference. The main idea is to compare the inferred type of each mapping query with the target schema, so to verify the adherence of the mapping to this schema.

Unlike [9], the proposed technique is independent from queries, does not require a prior query reformulation, and it is *complete*, i.e., any error in a mapping will be discovered.

The paper proposes a *semantic* notion of mapping correctness based on a *simulation-like* form of type projection. Type projection brings the essence of the relational projection to XML, and it can be safely reduced to standard type-checking among *weakened* types, as shown in Section 5. To minimize false-negatives, we provide quite precise type inference techniques, inspired by those proposed in [10].

The proposed technique can be used in unstructured p2p database systems as well as in structured systems, like [11], that combine the DHT paradigm with mappings.

Paper Outline The paper is structured as follows. Section 2 describes a reference scenario for our technique, and briefly introduce the system model and the query language. Section 3, then, defines our notions of mapping validity (no wrong rules wrt the source schema) and mapping correctness (no wrong rules wrt the target schema). Section 4 describes the type system we use for inferring query types. Section 5, next, shows how the definitions of Section 3 can be turned in an operational technique with the assistance of our type system. Section 6, then, discusses some related work. In Section 7, finally, we draw our conclusions.

2 Motivating Scenario

We describe our technique by referring to a sample XML p2p database system inspired by Piazza [4]. The system is composed of a dynamic set of peers, capable of executing queries on XML data, and connected through *sparse point-to-point* schema mappings.

Each peer publishes some XML data (db), that may be empty, in which case the peer only submits queries to the system. Furthermore, each peer has two distinct schema descriptions. The first one, $\mathcal{U}$ (the *peer schema*), describes how local data are organized. The second one, $\mathcal{V}$ (the *peer view*), is a view over $\mathcal{U}$, and has a twofold role. First, it works as input interface for the peer, so that queries sent to peer p_i should respect p_i view of the world. Second, it describes the peer *view* of the world, i.e., the virtual view against which queries are posed: each peer poses queries against its peer view, since it assumes that the outer world adopts this schema.

The peer schema and the peer view 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 GAV mappings, where the target schema is described in terms of the source schema; nevertheless, this approach applies to LAV mappings too, since, as noted in [12], a LAV mapping from p_i to p_j can be interpreted as a GAV mapping from p_j to p_i.

In addition to (possibly empty) data and schema information, each peer contains a set, possibly a singleton, of *peer mappings* $\{m_{ij}\}_j$. A peer mapping m_{ij} from peer p_i to peer p_j is a set of queries that show how to translate data belonging to the view of p_i ($\mathcal{V}_i$) into data conforming to a projection of the view of p_j ($\mathcal{V}_j$).

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, and will be described in Section 3. These mappings link peers together, and form a sparse graph; queries are then executed by exploring the transitive closure of such mappings.

Systems conforming to this architecture rely on schema mappings to process and execute queries. 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 system, namely the connection of new nodes and the disconnection of existing nodes, as well as the changes in peer data and 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 p2p systems, such as the mapping composition approach described in [13].

The following Example illustrates the basic concepts of the query language, provides an intuition of the mapping correctness notion (described in Section 3), and shows how mapping incorrectness can reflect on query answering.

Example 1. Consider a bibliographic data sharing system, whose topology is shown in Figure 1.

Assume that Pisa and New York use the following views.

```
PisaBib = bib[(Author)*]
Author = author[Name, Affiliation, Paper*]
Name = name[String]
```

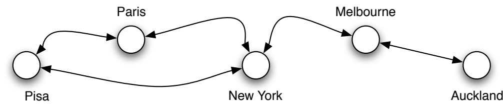

Fig. 1. Bibliographic p2p network.

```
Affiliation = affiliation[String]
Paper = paper[Title, Year]
Title = title[String]
Year = year[Integer]

NYBib = bib[(Article|Book)*]
Article = article[Author*,Title,Year, RefCode]
Author = author[String]
Title = title[String]
Year = year[Integer]
Book = book[Author*,Title,Year, RefCode]
RefCode = refCode[String]
```

Suppose now that Pisa uses the following queries to map its view into the view of New York.

```
NYBibliography <-
Q1($input): for $y in $input/year return $y
Q2($input): for $t in $input/title return $t
Q3($input): for $p in $input//paper,
                  $t in $p/title
            return article[ Q2($p), Q1($p),
                            for $aut in $input/author,
                                $pap in $aut/paper
                                $title in $pap/title
                            where $title = $t
                            return author[$aut/name/text()]]
Q4($input): for $bib in /bib return bib[Q3($bib)]
```

This mapping transforms data conforming to a large fragment of the `PisaBib` schema (only affiliation elements are discarded) into data conforming to a fraction of the `NYBib` schema. This is a quite common situation in data integration and p2p data sharing systems, since usually only a fraction of semantically related heterogeneous schemas can be reconciled. Since this mapping is not a function from `PisaBib` to `NYBib` (it does not produce `refCode` elements), standard result analysis based on subtyping cannot be used to check its correctness.

Consider query Q_3 in the Pisa $\rightarrow$ NY mapping. The outer `for` clause iterates over paper element, and binds the $\$p$ and $\$t$ variables to paper and title elements respectively. The outer `return` clause produces the results of the query; in this case, a nested query changing the nesting of author and paper elements is invoked. The correlation of the nested query with the outer query is given by the inner `where` clause, which filters the variable bindings of the inner query.

As it can be noted, this mapping is *correct* since it transforms a data instance conforming to `PisaBib` into a data instance conforming to a projection of `NYBib`.

Assume now that New York slightly changes its view: in particular, the site administrator changes the way author names are represented: instead of a simple author

element, information about author's first name and second name is inserted into the author element: `Author = author[first[String],second[String]]`.

This change in the target schema makes the Pisa → NY mapping incorrect. Indeed, a `PisaBib` data instance is transformed in a data instance having *simple content* author elements, while the new New York view requires more complex author elements.

The incorrectness of the Pisa → NY schema mapping reflects on query answering. Indeed, consider the query shown in Figure 2 (a). This query, submitted by a user in Pisa, asks for all articles written by Mary F. Fernandez. The query is first executed locally in Pisa. Then, the system reformulates the query so to match New York view; this reformulation is performed by directly composing the query with the mapping from Pisa to New York, relying again on standard algorithms for query unfolding [14, 13] [1].

At the end of the reformulation process, the reformulated query, shown in Figure 2 (b), is then sent to the New York site. Unfortunately, the transformed query does not match the new view of New York, so the Pisa user cannot gather results from the New York site.

```
articles_Fernandez[
for $aut in $bib/author,
    $pap in $aut/paper,
    $t in $pap/title,
    $n in $aut/name
let $mf := ''Mary F. Fernandez''
where $n = $mf
return article[$t, $pap/year]
```

(a) Pisa user query.

```
articles_Fernandez[
for $a in $bib/article,
    $aut in $a/author
let $mf := ''Mary F. Fernandez''
where $aut = $mf
return article[$a/title, $a/year]
```

(b) Transformed Pisa user query.

Fig. 2. Reformulation of a user query.

3 Mapping Validity and Correctness

In this Section we describe 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, as shown in Section 5.

To define mapping properties, we have to formally present the query language used for expressing both user queries and mapping rules, as well as the type language used for describing schemas and views.

3.1 Query Language

User queries and mapping rules are expressed in the μXQ query language [10], whose grammar is shown in Table 3.1. μXQ is a minimal core language for XML data, roughly equivalent to the FLWR core of XQuery. We impose two further restrictions wrt this grammar: first, we forbid the navigation of the result of a nested query by the outer query; second, we restrict the predicate language to the conjunction, disjunction, or negation of variable comparisons. These restrictions, also present in Piazza, allow for a better handling of errors at the price of a modest decrease in the expressive power of the language.

[1] We show a minimal transformed query, obtained by minimizing the original transformed query and by deleting all redundant subqueries.

The semantics of the language and the required auxiliary functions are shown in Tables 3.2 and 3.3. There, ρ is a substitution assigning a forest to each free variable in the query; also, `dos` is a shortcut for descendant-or-self. All the rest is self explicative.

Note that our data model is unordered, so that we consider a tree $l[f_1, f_2]$ as equivalent to $l[f_2, f_1]$. As in Piazza, this assumption is motivated by the non feasibility of imposing a global document order over XML data dispersed over a p2p network.

Table 3.1. *μXQ grammar*

$$
\begin{array}{lll}
Q & ::= & () \mid b \mid l[Q] \mid Q,Q \mid \overline{x}\ \texttt{child} :: \mathit{NodeTest} \mid \overline{x}\ \texttt{dos} :: \mathit{NodeTest} \\
 & & \mid \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{return}\ Q \mid \texttt{let}\ x ::= Q\ \texttt{return}\ Q \\
 & & \mid \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{where}\ P\ \texttt{return}\ Q \mid \texttt{let}\ x ::= Q\ \texttt{where}\ P\ \texttt{return}\ Q \\
\mathit{NodeTest} & ::= & \texttt{l} \mid \texttt{node()} \mid \texttt{text()} \\
P & ::= & \texttt{true} \mid \chi\ \delta\ \chi \mid \texttt{empty}(\chi) \mid P\ \texttt{or}\ P \mid \texttt{not}\ P \mid (P) \\
\chi & ::= & \overline{x} \mid x \\
\delta & ::= & = \mid <
\end{array}
$$

Table 3.2. *μXQ semantics*

$$
\begin{array}{llll}
[\![b]\!]_\rho & \triangleq b & [\![x]\!]_\rho & \triangleq \rho(x) \\
[\![\overline{x}]\!]_\rho & \triangleq \rho(\overline{x}) & [\![()]\!]_\rho & \triangleq () \\
[\![Q_1, Q_2]\!]_\rho & \triangleq [\![Q_1]\!]_\rho, [\![Q_2]\!]_\rho & [\![l[Q]]\!]_\rho & \triangleq l[[\![Q]\!]_\rho]
\end{array}
$$

$$[\![\overline{x}\ \texttt{child} :: \mathit{NodeTest}]\!]_\rho \triangleq \mathit{childr}([\![\overline{x}]\!]_\rho) :: NodeTest$$

$$[\![\overline{x}\ \texttt{dos} :: \mathit{NodeTest}]\!]_\rho \triangleq \mathit{dos}([\![\overline{x}]\!]_\rho) :: NodeTest$$

$$[\![\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2]\!]_\rho \triangleq [\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho}$$

$$[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2]\!]_\rho \triangleq \prod_{t \in trees([\![Q_1]\!]_\rho)} [\![Q_2]\!]_{\rho, \overline{x} \mapsto t}$$

$$[\![\texttt{let}\ x ::= Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2]\!]_\rho \triangleq \text{if } P(\rho, x \mapsto [\![Q_1]\!]_\rho) \text{ then } [\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho} \text{ else } ()$$

$$[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2]\!]_\rho \triangleq \prod_{t \in trees([\![Q_1]\!]_\rho)} (\text{if } P(\rho, \overline{x} \mapsto t) \text{ then } [\![Q_2]\!]_{\rho, \overline{x} \mapsto t} \text{ else } ())$$

3.2 Type Language

We adopt, essentially, XDuce's type language [15], with two exceptions. First, we exclude (vertical) recursive types. This is motivated by the fact that FLWR queries are not powerful enough to transform trees with arbitrary depth, hence we can restrict the type language to types that describe trees with limited and finite depth. As we will see, this restriction will allow us to introduce rather precise type-inference techniques, that will minimize false negatives returned while checking for mapping correctness.

Second, we consider commutative product types. In other words, we do not assume any order on sequence types, so that $T, U \sim U, T$. This is motivated by the fact that in distributed environments is almost impossible to reach a common agreement about

Table 3.3. *Auxiliary functions*

$dos(b)$	$\triangleq b$	$childr(b)$	$\triangleq ()$
$dos(l[f])$	$\triangleq l[f], dos(f)$	$childr(l[f])$	$\triangleq f$
$dos(())$	$\triangleq ()$	$dos(f, f')$	$\triangleq dos(f), dos(f')$
$b :: l$	$\triangleq ()$	$l[f] :: l$	$\triangleq l[f]$
$() :: l$	$\triangleq ()$	$(f, f') :: l$	$\triangleq f :: l, f' :: l$
$m[f] :: l$	$\triangleq ()$ $\quad m \neq l$		
$b :: node()$	$\triangleq ()$	$() :: node()$	$\triangleq ()$
$m[f] :: node()$	$\triangleq m[f]$	$(f, f') :: node()$	$\triangleq f :: node(), f' :: node()$
$b :: text()$	$\triangleq b$	$() :: text()$	$\triangleq ()$
$m[f] :: text()$	$\triangleq ()$	$(f, f') :: text()$	$\triangleq f :: text(), f' :: text()$

ordering, so some peer may assume that title elements precede author elements in the document order, while other peers may assume the contrary. Hence, we must adopt types that abstract from ordering. This aspect will affect the notions of type projection as well.

Following XDuce notation, types are defined as follows:

$$\textbf{Types} \quad T ::= () \mid B \mid l[T] \mid T,T \mid T \mid T \mid T*$$
$$\textbf{Base Type} \quad B ::= \texttt{String}$$

Here, $()$ 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, while $T*$ is the type for repetition. In the following, an element type with empty content $l[()]$ will always be abbreviated as $l[]$.

Type semantics is standard: $[\![_]\!]$ is the minimal function from types to sets of forests that satisfies the following monotone equations:

$$[\![()]\!] \triangleq \{()\} \qquad [\![B]\!] \triangleq \{b \mid b \textit{ is a string}\} \qquad [\![l[T]]\!] \triangleq \{l[f] \mid f \in [\![T]\!]\}$$
$$[\![T_1 \mid T_2]\!] \triangleq [\![T_1]\!] \cup [\![T_2]\!] \qquad [\![T_1, T_2]\!] \triangleq \{f_1, f_2 \mid f_i \in [\![T_i]\!]\} \qquad [\![T*]\!] \triangleq [\![T]\!]^*$$

In the following we will use $f : T$ as shortcut for $f \in [\![T]\!]$. Type semantics induces the following subtyping relation:

$$T < U \;\Leftrightarrow_{\text{def}}\; [\![T]\!] \subseteq [\![U]\!]$$

3.3 Correctness of Schema Mappings

In this Section, we introduce and formalize our notion of mapping correctness. The notion is *semantic* and is not related to any particular type system.

Definition 1 (Mapping). *A mapping m from the peer view of p_i to the peer view of p_j is a set of queries $m = \{q_k\}_k$ on data (possibly) conforming to p_i's view and returning data (possibly) conforming to p_j's view.*

The previous definition states that a mapping is just a set of queries that may match the source and/or the target schema. Unlike [16], where mappings must match both the target and the source schema, we do not impose constraints on mappings. This allows for capturing mappings that are imprecise or that become incorrect because of a change in the system status.

The following definition introduces the notion of mapping validity.

Definition 2 (Mapping validity). *A mapping $m = \{q_k\}_k$ from p_i's view to p_j's view ($\mathcal{V}_i \rightarrow \mathcal{V}_j$) is* valid *if and only if, for each query q_k, q_k is correct wrt $\mathcal{V}_i$, in the sense that, for each non-empty subquery q of q_k, there exists a data instance d of $\mathcal{V}_i$ such that, when evaluated on d, q will return a non-empty result.*

Mapping validity implies that a valid mapping must be correct wrt the source schema, i.e., it matches the structure and the constraints of the source schema. We adopt the query correctness notion described in [18, 10] and [9]. Mapping validity [2] 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 the detection of errors wrt the target schema.

Definition 3 (Mapping correctness). *A mapping $m = \{q_k\}_k$ from p_i's view to p_j's view ($\mathcal{V}_i \rightarrow \mathcal{V}_j$) is* correct *if and only if, for each query q_k, for each data instance d_h conforming to $\mathcal{V}_i$, there exists a data instance d_l conforming to $\mathcal{V}_j$, such that, $q_k(d_h) \lesssim d_l$, where $\lesssim$ is defined as shown in Definition 4.*

Definition 4 (Value projection). *The value projection relation $\lesssim$ is the minimal relation such that:*

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

The above definitions state that a mapping from $\mathcal{V}_i$ to $\mathcal{V}_j$ is correct if and only, for each rule in the mapping, the result of each query on $\mathcal{V}_i$ is mapped, according to the $\lesssim$ relation, into a value conforming to $\mathcal{V}_j$. $\lesssim$ is an *injective* simulation relation among values, inspired by the projection operator of the relation data model. Intuitively, $d_1 \lesssim d_2$ if there exists a subterm d_3 in d_2 such that d_3 matches d_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. This notion of projection for XML trees is a generalization of that introduced in [17], where leaf values are taken into account too.

Our correctness notion is *semantic*, in the sense that it depends on the semantics of queries and types rather than on a set of type-checking rules; this implies that errors are independent from the type-checking rules, so that our correctness notion can be adopted in any context and with any type language.

4 Type System

Our type system is a variation of the type systems shown in [18][10][9]. While those type systems focus on the detection of errors in a query wrt a source schema, this type system focuses on type inference.

[2] Validity can be checked by algorithms proposed in [18][10][9]; these algorithms are polynomial in most practical cases.

4.1 Judgments and Type Rules

To infer the output type of a μXQ query, we adopt rules, shown in Tables 4.1 and 4.2, that prove judgments of the form $\Gamma \vdash Q : T$, where the environment Γ provides information about the types of Q free variables, while T is an upper bound for all possible values returned by Q, when evaluated under a valid substitution, that is an assignment of free variables that respects type constraint in Γ . Variable environments and valid substitutions are defined below.

$$\textbf{Variable Environments} \quad \Gamma ::= () \mid x : T, \Gamma \mid \overline{x} : T, \Gamma$$

A variable environment Γ is well-formed if no variable is defined twice, and if every for-variable $\overline{x}$ (i.e., a variable bound by a `for` clause) is associated to a tree type ($l[T']$ or B).

Definition 5 (Valid substitutions $\mathcal{R}(\Gamma)$). *For any well-formed environment Γ, we define the set of valid substitutions wrt Γ as follows:*

$$\mathcal{R}(\Gamma) = \{\rho \mid \chi \mapsto f \in \rho \Rightarrow (\chi : T \in \Gamma \wedge f \in [\![T]\!])\}$$

A first basis for a good level of precision is given by a particular technique we use to infer types for `for` queries. Given a query `for` $\overline{x}$ `in` Q_1 `return` Q_2, in order to infer a type for it, the rules first infer a type T_1 for Q_1, and then they simulate a sort of abstract iteration over T_1, in order to type the body Q_2. This is done by means of the auxiliary judgment $\Gamma \vdash \overline{x}\ \texttt{in}\ T_1 \rightarrow Q_2 : T_2$.[3]

For example, if $T_1 = S_1, S_2$, to type `for` $\overline{x}$ `in` Q_1 `return` Q_2 we recursively prove $\Gamma \vdash \overline{x}\ \texttt{in}\ S_i \rightarrow Q_2 : S'_i$, for $i = 1, 2$, and then combine the results to obtain $T_2 = S'_1, S'_2$. The recursive process is still purely structural for union and * types, and stops when a tree type is finally encountered.

Similar comments hold for queries with `where` conditions, where we use an auxiliary judgment $\Gamma \vdash \overline{x}\ \texttt{in}\ T \rightarrow Q\ \texttt{where}\ P : T$. Type correctness of `where` clauses is proved by rules over judgments $\Gamma \vdash P$ which are quite standard (and omitted in this abstract for reasons of space).

More in details, case analysis for iterations is performed by (TYPEIN_) rules. In particular, termination is ensured by rule (TYPEINTREE), which stops the case-analysis, since a tree type T=B or T=$m[T']$ is reached, inserts the assumption $\overline{x} : T$ in Γ, starts the analysis of the `where` condition P, and falls back to standard type-checking. Observe here that we use an operator *Split*(T); for the moment just assume that *Split*$(T) = \{T\}$. Later we will modify this operator in order to improve precision of type inference. Rule (TYPELETSPLITTING) is standard, since we are assuming that *Split*$(T) = \{T\}$.

Rule (TYPECHILD) requires the type of $\overline{x}$ to be a tree type $m[T']$, and uses $\vdash T' ::$ *NodeTest* $\Rightarrow U$ to restrict the content type T' to the tree types with structure satisfying *NodeTest*. Rules to prove judgments $\vdash T' ::$ *NodeTest* $\Rightarrow U$ are straightforward, and their meaning is stated in the following lemma.

Lemma 1 (Type Filtering Checking). *For any T :*

$$\vdash T :: \textit{NodeTest} \Rightarrow U \Leftrightarrow [\![U]\!] = \{f :: \textit{NodeTest} \mid f \in [\![T]\!]\}$$

[3] This technique was first formalized in [19], where no properties about the system were proved.

Rule (TYPEDOS) is similar, and is strictly inspired by the technique adopted in the current W3C XQuery type system. Instead of using the content type T', it extracts all the node types $\{U_1, \ldots, U_n\}$ that are reachable from T, using the function Trees(T) defined later, and defines a new type $U' = (U_1 \mid \ldots \mid U_n)*$. U' is the type of any forest that contains only nodes whose type is one of the U_i's, hence is an appropriate type for the forest of all descendants of a tree of type T. The type of $\overline{x}$ `dos` :: *NodeTest* is obtained by restricting U' to the tree types with structure satisfying *NodeTest*.

We can now define the auxiliary function Trees(T):

Definition 6 (Subtrees Type Extraction).

$$
\begin{array}{llll}
\mathit{Trees}(()) & \triangleq \emptyset & \mathit{Trees}(T, U) & \triangleq \mathit{Trees}(T) \cup \mathit{Trees}(U) \\
\mathit{Trees}(B) & \triangleq \{B\} & \mathit{Trees}(T*) & \triangleq \mathit{Trees}(T) \\
\mathit{Trees}(l[T]) & \triangleq \{l[T]\} \cup \mathit{Trees}(T) & \mathit{Trees}(T \mid U) & \triangleq \mathit{Trees}(T) \cup \mathit{Trees}(U)
\end{array}
$$

Table 4.1. *Query Type Rules*

(TYPEEMPTY)
$$\frac{\mathit{WF}(\Gamma \vdash () : ())}{\Gamma \vdash () : ()}$$

(TYPEATOMIC)
$$\frac{\mathit{WF}(\Gamma \vdash b : B)}{\Gamma \vdash b : B}$$

(TYPEVAR)
$$\frac{\chi : T \in \Gamma \quad \mathit{WF}(\Gamma \vdash \chi : T)}{\Gamma \vdash \chi : T}$$

(TYPEELEM)
$$\frac{\Gamma \vdash Q : T}{\Gamma \vdash l[Q] : l[T]}$$

(TYPEFOREST)
$$\frac{\Gamma \vdash Q_i : T_i \quad i = 1, 2}{\Gamma \vdash Q_1, Q_2 : T_1, T_2}$$

(TYPELETWHERESPLITTING)
$$\frac{\Gamma \vdash Q_1 : T_1 \quad \mathit{Split}(T_1) = \{A_1, \ldots, A_n\} \qquad \Gamma,\, x : A_i \vdash P \quad \Gamma,\, x : A_i \vdash Q_2 : U_i \quad i = 1 \ldots n}{\Gamma \vdash \texttt{let}\ x ::= Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2 : U_1 \mid \ldots \mid U_n \mid ()}$$

(TYPEFORWHERE)
$$\frac{\Gamma \vdash Q_1 : T_1 \quad \Gamma \vdash \overline{x}\ \texttt{in}\ T_1 \rightarrow Q_2\ \texttt{where}\ P : T_2}{\Gamma \vdash \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2 : T_2 \mid ()}$$

(TYPELETSPLITTING)
$$\frac{\Gamma \vdash Q_1 : T_1 \quad \mathit{Split}(T_1) = \{A_1, \ldots, A_n\} \qquad \Gamma,\, x : A_i \vdash Q_2 : U_i \quad i = 1 \ldots n}{\Gamma \vdash \texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2 : U_1 \mid \ldots \mid U_n}$$

(TYPEFOR)
$$\frac{\Gamma \vdash Q_1 : T_1 \qquad \Gamma \vdash \overline{x}\ \texttt{in}\ T_1 \rightarrow Q_2\ \texttt{where}\ \texttt{true} : T_2}{\Gamma \vdash \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : T_2}$$

Lemma 2 (Soundness of DOS). *For any* T *:*

$$\{U_1, \ldots, U_n\} = \mathit{Trees}(T) \wedge U = (U_1 \mid \ldots \mid U_n)* \Rightarrow \forall f \in [\![T]\!].\ \mathit{dos}(f) \in [\![U]\!]$$

4.2 Soundness of the Type System

We provisionally assumed that *Split*$(T) = \{T\}$, which results in a completely standard LET-RETURN type rule. This is sufficient to obtain the canonical 'soundness' property (Theorem 1): types are upper bounds for the set of all possible results.

Table 4.2. *Query Type Rules: Rules for Iteration, Child and Dos.*

(TYPEINEMPTY)

$$\frac{WF(\Gamma \vdash \overline{x}\ \texttt{in}\ () \rightarrow Q\ \texttt{where}\ P : ())}{\Gamma \vdash \overline{x}\ \texttt{in}\ () \rightarrow Q\ \texttt{where}\ P : ()}$$

(TYPEINUNION)

$$\frac{\Gamma \vdash \overline{x}\ \texttt{in}\ T_i \rightarrow Q\ \texttt{where}\ P : T'_i \quad i = 1,2}{\Gamma \vdash \overline{x}\ \texttt{in}\ T_1 \mid T_2 \rightarrow Q\ \texttt{where}\ P : T'_1 \mid T'_2}$$

(TYPEINTREE)

$$\frac{\begin{array}{l}(T = m[T'] \vee T = B) \quad \mathit{Split}(T) = \{A_1, \ldots, A_n\} \\ \Gamma,\ \overline{x} : A_i \vdash P \quad \Gamma,\ \overline{x} : A_i \vdash Q : U_i \quad i = 1 \ldots n\end{array}}{\Gamma \vdash \overline{x}\ \texttt{in}\ T \rightarrow Q\ \texttt{where}\ P : U_1 \mid \ldots \mid U_n}$$

(TYPEINCONC)

$$\frac{\Gamma \vdash \overline{x}\ \texttt{in}\ T_i \rightarrow Q\ \texttt{where}\ P : T'_i \quad i = 1,2}{\Gamma \vdash \overline{x}\ \texttt{in}\ T_1, T_2 \rightarrow Q\ \texttt{where}\ P : T'_1, T'_2}$$

(TYPEINSTAR)

$$\frac{\Gamma \vdash \overline{x}\ \texttt{in}\ T \rightarrow Q\ \texttt{where}\ P : U}{\Gamma \vdash \overline{x}\ \texttt{in}\ T* \rightarrow Q\ \texttt{where}\ P : U*}$$

(TYPECHILD)

$$\frac{\begin{array}{l}WF(\Gamma \vdash \overline{x}\ \texttt{child} :: \mathit{NodeTest} : U) \\ \overline{x} : T \in \Gamma \wedge (T = m[T''] \vee T = B) \\ T' = \text{ if } T = m[T''] \text{ then } T'' \text{else } () \\ \vdash T' :: \mathit{NodeTest} \Rightarrow U\end{array}}{\Gamma \vdash \overline{x}\ \texttt{child} :: \mathit{NodeTest} : U}$$

(TYPEDOS)

$$\frac{\begin{array}{l}WF(\Gamma \vdash \overline{x}\ \texttt{dos} :: \mathit{NodeTest} : U) \\ \overline{x} : T \in \Gamma \wedge (T = m[T''] \vee T = B) \\ \{U_1, \ldots, U_n\} = \text{Trees}(T) \\ U' = (U_1 \mid \ldots \mid U_n)* \\ \vdash U' :: \mathit{NodeTest} \Rightarrow U\end{array}}{\Gamma \vdash \overline{x}\ \texttt{dos} :: \mathit{NodeTest} : U}$$

Theorem 1 (Upper Bound). *For any well-formed Γ and query Q:*

$$\Gamma \vdash Q : U \wedge \rho \in \mathcal{R}(\Gamma) \Rightarrow [\![Q]\!]_\rho \in [\![U]\!]$$

The proof of this theorem is essentially the same as the one given in [18] [10], since considered XPath-like paths do not match the horizontal structure of sequences, so their typing does not depend on ordering.

This theorem is crucial to guarantee soundness of mapping correctness checking. Indeed, if Q is a mapping from $\mathcal{S}_i$ to $\mathcal{S}_j$, and $\Gamma \vdash Q : U$, then thanks to the above theorem, we can compare U wrt $\mathcal{S}_j$ in order to verify whether the semantics of Q conforms to $\mathcal{S}_j$. In the next section we will formalize how this comparison can be done in order to agree to the notion of mapping correctness (Definition 3).

The system cannot be made complete: as for any type system based on regular expression types, the presence of queries that may produce sets of trees that are not regular languages makes completeness impossible. However, we will see later how the precision of the type system may be improved, and why more precision is desirable in our context.

5 Correctness Checking

Definitions 3 and 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.

Definition 7 (Type projection). *Given two type 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 semantics, 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.

To use type projection in mapping correctness checking, we must correlate type projection and mapping correctness. To this aim, we can rely on the result type of a query as inferred by our type system, as shown in the following theorem.

Theorem 2 (Completeness of type projection). *Given a mapping $m = \{q_k\}_k$ from $\mathcal{V}_i$ to $\mathcal{V}_j$, m is correct if $\forall q_k.\ \Gamma \vdash\ q_k : T$ and $T \lesssim \mathcal{V}_j$, where Γ is an environment obtained from $\mathcal{V}_i$.*

The previous theorem states that, 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.

The type projection relation is still not operational, since its definition involves a universal quantification on the data instances of the source schema. To overcome this problem and obtain a practical way of checking type projection, we introduce the 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 unordered types, whose decidability has been proved in [20].

5.1 Type Approximation

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

Definition 8 (Type approximation). *Given a type U, we indicate with $U^{\lhd}$ 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*

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

It is easy to prove that $T < T^{\lhd}$. To prove the main results about type approximation, we have to introduce the notion of *contexts*, whose grammar is shown below.

$$\textbf{Contexts}\ C ::= x \mid () \mid C, C \mid l[C] \mid b$$

A context is a partially specified forest, where variables indicate arbitrary forests. Variables are always assumed to be unique, and context instantiation is indicated as C_{ρ}, where ρ is a set of variable assignments $x \mapsto f$. We indicate with $C_{()}$ the forest obtained by C by replacing each variable with the empty sequence.

If we indicate with $f \simeq f'$ the fact that the two forests are equal up to ordering among children and values at leafs, we can state the following lemma.

Lemma 3. *Given two forests f_1 and f_2, the following relation holds:*

$$f_1 \lesssim f_2 \Leftrightarrow \exists C.\exists \rho.\ C_{()} \simeq f_1 \wedge f_2 \simeq C_{\rho}$$

The following theorem correlates $T^{\lhd}$ with T.

Theorem 3.

$$T^{\lhd} \lesssim T$$

Lemma 4. *For each type U:*

$$1.\ \forall f : U^{\lhd}.\ (\ f \neq ()\ \Rightarrow\ \exists C,\ \rho,\ f' : U.\ C_{()} = f\ \wedge\ f' = C_{\rho})$$
$$2.\ \forall f : U, C, \rho.\ (f = C_{\rho}\ \Rightarrow\ C_{()} : U^{\lhd})$$
$$3.\ \forall C.\ (C_{()} \neq ()\ \wedge\ C_{()} : U^{\lhd}\ \Rightarrow \exists f : U. \exists \rho.\ f = C_{\rho})$$

The previous lemma serves to prove the following main theorem.

Theorem 4 (Type projection as sub-typing).

$$T \lesssim U \Leftrightarrow T < U^{\lhd}$$

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 $U^{\lhd}$. This theorem proves the decidability of type projection, since decidability of subtyping for a superset of our type language has been proved in [20]. For what concerns the complexity of type projection, we recently proved that, for our type language, type projection can be checked in polynomial type, hence making our maintenance approach more effective.

5.2 Improving Precision of Type Inference

As already observed, inferred types cannot precisely capture query semantics. However, there is some space for gaining more precision, which implies less false-negative in checking mapping correctness. This is typical of every approach based on result analysis, including those of languages of the XDuce family.

As shown in [18][10], by tuning the operator *Split*(T), we may improve the precision of the type system. Under the assumption *Split*$(T) = \{T\}$, the presented type system is not precise enough when, for example, there are variables that occur more than once (*non-linear* variables) and with a union type. For example, consider the (artificial) type $X = data[mbl[]+ \mid phn[]+]$, and the sequence query $(x/mbl, x/phn)$. When x has type X, this query yields either a sequence of elements $mbl[]$ or a sequence of elements $phn[]$. Instead, as in XQuery, our type system infers a type $(mbl[]*, phn[]*)$, which also contains sequences with both $mbl[]$ and $phn[]$ elements. If this type is compared with $(mbl[]* \mid phn[]*)$, in order to check whether the query output conforms to this expected type, the checking will fail thus producing a false negative.

We solve these problems by using in the rules a finer *Split*() function, which produces more precise types. For example, if the input type $X = data[mbl[]+ \mid phn[]+]$ is split in the two types $data[mbl[]+]$ and $data[phn[]+]$, and, then, two separate analysis are performed, we obtain the types $data[mbl[]*]$ and $data[phn[]*]$. Then the query type is the union of these two types, and thus a subtype of the previous expected type, thus avoiding a false negative.

The definition of *Split*(T) is non-trivial in the presence of recursive types. In [18][10] we propose a solution that works under a mild restrictions over the use of recursion. Here, we propose the same definitions without making any restriction as recursive types have already been excluded.

Definition 9 **(*Split*(T)).**

$$\begin{array}{llll} \mathit{Split}(()) & \triangleq \{()\} & \mathit{Split}(T \mid U) & \triangleq \mathit{Split}(T) \cup \mathit{Split}(U) \\ \mathit{Split}(B) & \triangleq \{B\} & \mathit{Split}(l[T]) & \triangleq \{l[A] \mid A \in \mathit{Split}(T)\} \\ \mathit{Split}(U*) & \triangleq \{U*\} & \mathit{Split}(T, U) & \triangleq \{(A, B) \mid A \in \mathit{Split}(T) \wedge B \in \mathit{Split}(U)\} \end{array}$$

Splitting stops when a *-type is met. As shown in [10], this ensures acceptable complexity for a very wide class of cases, while ensuring good precision at the same time, as in schemas most union types are the form $(T \mid U)*$, which are not split.

To have an idea of the precision that we gain by splitting, we have that a query Q without `where` conditions always evaluates to $()$, under well-typed substitutions, if and only if its inferred type is $()$; as shown in [10], this does not hold without splitting. As a second example, the reader can run the rules over Example 1, and realize that the inferred type is quite precise and is a projection of the target type.

To conclude, since we are considering non recursive types, we believe that an alternative typing for $\overline{x}$ `dos` :: *NodeTest* expressions, based on the abstract execution of the descendant-or-self operator over the type bound to $\overline{x}$, by possibly using splitting, may further improve precision. We leave this issue as future work.

6 Related Work

To the best of our knowledge, the only alternative technique for detecting corrupted mappings in XML p2p systems is the one described in [9]. We have already discussed differences between the present approach and that work. Other works on p2p systems [16] [5] do not address the problem of checking mapping correctness: they always assume mappings to be correct, with a correctness notion very close to our semantic correctness. Starting from correct mappings, [16] proposes a correct and complete query answering algorithm for p2p data integration systems.

Our type system is a variation of the type systems of [10] and [9], obtained by dropping error-checking in favor of a better precision in type inference. In these works we have already outlined advantages of these type systems wrt to the W3C XQuery type system [21].

7 Conclusions and Future Work

This paper presented a novel technique for detecting corrupted mappings in XML p2p 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.

To check mapping correctness, we introduced a notion of type projection for XML types. By reducing type projection to standard subtyping among weakened types, it follows that type projection is decidable [20]. We recently proved that type projection can be checked in polynomial time.

We proved that mapping correctness can be reduced to type projection between the inferred result type of the mapping and the target schema, and showed that our approach is *complete*, i.e., all errors will be detected. To decrease false negatives, we augment the precision of type inference through type splitting.

Although this work is not in its infancy, much work remains to do as it forms the basis for a massive future work. In particular, we plan, in the near future, to implement this technique in a centralized, logical p2p system, so to verify its applicability in a background maintenance activity. Finally, we plan to enrich our approach with some form of *self-healing* technique, so to suggest to the user possible corrections for any detected wrong mapping.

References

1. Dabek, F., Brunskill, E., Kaashoek, M.F., Karger, D.R., Morris, R., Stoica, I., Balakrishnan, H.: Building peer-to-peer systems with chord, a distributed lookup service. In: HotOS. (2001) 81–86
2. : (The FreePastry System. www.cs.rice.edu/cs/systems/pastry/freepastry/)
3. Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium (2003) W3C Working Draft.
4. Halevy, A.Y., Ives, Z.G., Mork, P., Tatarinov, I.: Piazza: data management infrastructure for semantic web applications. In: Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003, ACM (2003) 556–567
5. Franconi, E., Kuper, G.M., Lopatenko, A., Zaihrayeu, I.: Queries and updates in the codb peer to peer database system. In: VLDB. (2004) 1277–1280
6. Goasdoué, F., Rousset, M.C.: Answering queries using views: A krdb perspective for the semantic web. ACM Trans. Internet Techn. **4** (2004) 255–288
7. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press (1988)
8. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume II. Computer Science Press (1989)
9. Colazzo, D., Sartiani, C.: Typechecking Queries for Maintaining Schema Mappings in XML P2P Databases. In: Proceedings of the 3th Workshop on Programming Language Technologies for XML (Plan-X), in conjunction with POPL 2005. (2005)
10. Colazzo, D., Ghelli, G., Manghi, P., Sartiani, C.: Types for Path Correctness of XML Queries. In: Proceedings of the 2004 International Conference on Functional Programming (ICFP), Snowbird, Utah, September 19-22, 2004. (2004)
11. Abiteboul, S., Manolescu, I., Preda, N.: Sharing Content in Structured P2P Networks. Technical report, INRIA (2005)
12. Tatarinov, I.: Semantic Data Sharing with a Peer Data Management System. PhD thesis, University of Washington (2004)
13. Tatarinov, I., Halevy, A.Y.: Efficient query reformulation in peer-data management systems. In: SIGMOD Conference. (2004) 539–550
14. Madhavan, J., Halevy, A.Y.: Composing mappings among data sources. In: VLDB. (2003) 572–583
15. Hosoya, H., Pierce, B.C.: XDuce: An XML Processing Language (1999) Preliminary Report.
16. Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Logical foundations of peer-to-peer data integration. In: PODS. (2004) 241–251
17. Marian, A., Siméon, J.: Projecting xml documents. In: VLDB. (2003) 213–224
18. Colazzo, D.: Path Correctness for XML Queries: Characterization and Static Type Checking. PhD thesis, Dipartimento di Informatica, Università di Pisa (2004)
19. Fernandez, M., Siméon, J., Wadler, P.: A Semi-monad for Semi-structured Data. In: ICDT. (2001) 263–300
20. Dal-Zilio, S., Lugiez, D., Meyssonnier, C.: A logic you can count on. In Jones, N.D., Leroy, X., eds.: POPL, ACM (2004) 135–146
21. Draper, D., Fankhauser, P., Fernandez, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium (2005) W3C Working Draft.
22. Benzaken, V., Castagna, G., Frisch, A.: Cduce: an xml-centric general-purpose language. In: ICFP. (2003) 51–63