Typechecking Queries for Maintaining Schema Mappings in XML P2P Databases*

Dario Colazzo
Laboratoire de Recherche en Informatique (LRI)
Bat 490 Université Paris Sud
91405 Orsay Cedex France
dario.colazzo@lri.fr

Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
Via F. Buonarroti 2 - 56127 - Pisa - Italy
sartiani@di.unipi.it

Abstract

Peer data management systems (PDMSs) for XML data allow the user to easily share and query XML data dispersed over multitudes of sites without the complex and heavy administrative tasks that characterize traditional distributed database systems. Most of them are based on a common model, where nodes in the system are connected through sparse *point-to-point* mappings, and queries are executed with decentralized versions of GAV and LAV query reformulation algorithms.

The presence of links among peers poses new problems related to the intrinsically dynamic nature of p2p systems, namely the fact that peers may join and leave the network at any time, as well as locally and independently change their data and schemas. As a consequence, mappings may suddenly become corrupted, hence greatly affecting the quality of results retrieved by the system. At this time, both the detection of problems in the current set of mappings and the maintenance of these mappings are performed manually by each site owner/user, and systems have no way to automatically warn the user about emerging issues in the mapping chain.

In this paper we present an automatic technique for identifying corrupted as well as imprecise links in XML p2p database systems. Our technique, based on static type analysis of XQuery like queries, employs an enhanced version of the μXQ type system [5], that allows for a precise location of errors in queries wrt schema definitions.

1 Introduction

The last few years have seen the rapid emerging of two new Internet-related technologies. The first one, the eXtensible Markup Language (XML), was designed in an effort to make WWW documents cleaner and machine understandable, and has become the standard format for exchanging data between different data sources. The second technology, the *peer-to-peer* (p2p) computational paradigm, started as a way to easily share files over the Internet, and affirmed as a low-cost, scalable and flexible evolution of client-server and distributed systems.

In the field of database systems, these technologies combine to form to a new family of data management systems, sometimes called PDMSs (Peer Data Management Systems), that allow the user to easily share and query XML data dispersed over multitudes of sites without the complex and heavy administrative tasks that characterize traditional distributed database systems. Most current p2p database systems for XML data [10] [15] [8] [2] are based on a common model, where nodes in the system are connected through sparse *point-to-point* mappings, and queries are executed with decentralized versions of GAV and LAV query reformulation algorithms [17, 18]. PDMSs can be seen as a mixture of data integration an p2p technologies, hence resulting in adaptation of prior data integration techniques to a widely distributed and someway unstable environment.

The presence of mappings among peers, while allowing for a more precise and less expensive query routing than *flood-based* p2p systems (e.g., Gnutella [1]), poses new problems related to the intrinsically dynamic nature of p2p systems, namely the fact that peers may join and leave the network at any time, as well as locally and independently change their data and schemas: for instance, when a peer changes the structure of the data it is sharing, its mappings with other peers may become corrupted. The introduction

*Dario Colazzo was funded by the RNTL-GraphDuce project. Carlo Sartiani was funded by the FIRB GRID.IT project.

of corrupted mappings in the system greatly affects the quality and the quantity of the results that can be retrieved in response to a query. Indeed, queries are usually evaluated by traversing a chain of peers and by exploiting their mappings in the query reformulation process: as a consequence, the presence of a corrupted or imprecise mapping from peer p_i to peer p_{i+1} may make peer p_{i+1} unreachable, and may influence the reachability of other peers.

At this time, both the detection of problems in the current set of mappings and the maintenance of these mappings are performed manually by each site owner/user, and systems have no way to automatically warn the user about emerging issues in the mapping chain.

Our Contribution In this paper we present an automatic technique for identifying corrupted as well as imprecise schema mappings in XML p2p database systems. Our technique, based on the typechecking of XQuery queries, employs an enhanced version of the XQuery type system [5], that allows for a precise location of errors in queries wrt schema definitions. By relying on this technique, the p2p system can promptly warn the user about errors in mapping definitions and give her hints about the location of the errors in the mapping chain, as well as about the schema fragments whose mappings should be improved. The approach we are proposing can be used for maintaining existing links among peers, as well as for assisting the site owners in the design of new mappings.

The proposed technique can be safely combined with traditional query processing algorithms based on the repeated applications of GAV and LAV algorithms over the transitive closure of schema mappings.

Paper Outline The paper is organized as follows. Section 2 shows the scenario for this work and, in particular, introduces the formalism we use for modeling XML p2p database systems. Section 3, then, sketches the type system we are using and its differences wrt the standard type system of XQuery. Section 4, then, describes the main properties of the type system. Section 5, next, shows how the typechecking algorithm of the type system can be used to locate, if any, errors in the mapping chain. In Sections 6 and 7, finally, we discuss some related work and draw our conclusions.

2 Motivating Scenario

We describe our technique by referring to a sample XML p2p database system inspired by Piazza [10]. 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.

The state of the system is modeled as a dynamic set $\{p_i\}_i$ of peers, where each peer is represented as shown by Definition 2.1.

Definition 2.1 *A peer is a tuple* $p_i = (id, db, \mathcal{T}, \mathcal{V}, \{\rho_{ij}\}_j)$*, where:*

- *id is the unique identity of* p_i*;*
- *db are the data published by* p_i*;*
- $\mathcal{T}$ *is the schema of the data of* p_i*;*
- $\mathcal{V}$ *is the world view of* p_i*, i.e., the view against which* p_i *queries are posed;*
- *and* $\{\rho_{ij}\}_j$ *is a set of* bidirectional *point-to-point mappings from* p_i *view to the views of other peers.*

As pointed out by the definition, each peer publishes some XML data (*db*), that are described according to the data model shown below; as it can be seen, published data may be empty, in which case the peer only submits queries to the system.

$$\begin{array}{lcl} f & ::= & () \mid t \mid f, f \\ t & ::= & b \mid l[f] \end{array}$$

Each peer has two distinct schema descriptions. The first one, $\mathcal{T}$ (the *peer schema*), describes how local data are organized; these data may be empty, as it happens in most p2p file sharing systems, in which case $\mathcal{T}$ is just the empty type. The second one, $\mathcal{V}$ (the *peer view*), is a view over $\mathcal{T}$, 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; this allows a peer to limit the fraction of its local data accessible from the outer world (e.g., the peer may make visible only unclassified data), and to export *computed* or *aggregated* data not really present in the database (i.e., the number of author of a given paper instead of their names). Second, it describes the peer *view* of the world, i.e., the virtual view against which queries are posed. $\mathcal{V}$ can be seen as the schema that the peer assumes to be adopted by the rest of the world; so, each peer poses queries against its peer view, since it assumes that the outer world adopts this schema.

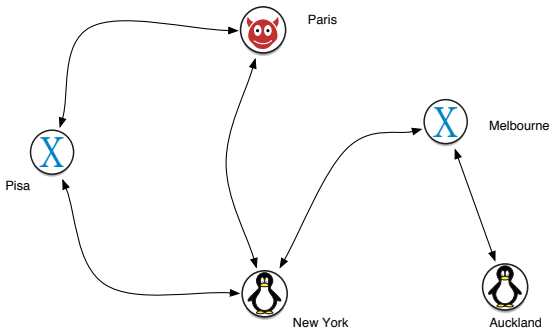

Figure 1: Bibliographic p2p network.

```
PisaBib = bib[(Articolo)*]
Articolo = articolo[Autore*,Titolo,Anno]
Autore = autore[String]
Titolo = titolo[String]
Anno = anno[Integer]
```

Figure 2: Pisa view.

The peer schema and the peer view are connected through a schema mapping, that shows how to translate a query expressed on the peer view into a query against the peer schema (in the following we will use the "schema mapping" to denote any mapping between types). The mapping can be defined according to the *Global As View* approach, or to the *Local As View* approach. In the first case, the view is defined in terms of the schema, so that any query posed against the view can be executed by just composing it with the view definition (*query unfolding*); on the contrary, in the latter case the schema is defined in terms of the view, so that query execution requires the system to perform complex rewritings on the query. The pros and cons of the GAV and LAV approach are extensively discussed in the literature; our distributed typechecking algorithm is independent from the view definition approach, so we can abstract from the specific view definition policy.

Since $\mathcal{V}$ plays the role of view of the world too, it can be loosely connected to $\mathcal{T}$: in particular, when $\mathcal{T}$ is empty, $\mathcal{V}$ is fully independent from $\mathcal{T}$:

The presence of two distinct schema descriptions has been introduced in the GLAV data integration approach [9], and it allows one to use both the LAV and GAV query rewriting approaches. $\mathcal{T}$ and $\mathcal{V}$ are systems of type equations conforming to the type system we will describe in the next Section. The following Example shows the views used in a sample p2p database.

Example 2.2 Consider a bibliographic data sharing system, where each node publishes its bibliographic references encoded as XML data, and accesses the references of other nodes. Assume that the system is composed of five nodes as shown in Figure 1.

As suggested in Figure 1, the peer in Pisa is *directly* connected with peers in Paris and in New York, and interacts with other peers by means of the view shown in Figure 2.

This view tells that users in Pisa see the p2p database as formed by elements tagged Articolo, each one containing the most relevant information about an article. Moreover, this view acts as an access controller to Pisa data, since, as shown by the Pisa schema of Figure 3, it blocks the access to authors addresses. New York view, shown in Figure 4, differs from that of Pisa since New York users can access not only articles, as Pisa users, but also books in the database.

■

```
PisaBib = bib[(Articolo)*]
Articolo = articolo[Autore*,Titolo,Anno]
Autore = autore[Nome,Indirizzo]
Nome = nome[String]
Indirizzo = indirizzo[String]
Titolo = titolo[String]
Anno = anno[Integer]
```

Figure 3: Pisa view.

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

Figure 4: New York view.

In addition to (possibly empty) data and schema information, each peer contains a set, possibly a singleton, of *peer mappings* $\{\rho_{ij}\}_j$. A peer mapping ρ_{ij} from peer p_i to peer p_j is a pair of type correspondences $(\mathcal{V}_j \leftarrow q_{1j}, \mathcal{V}_i \leftarrow q_{2j})$ that map the view of p_i ($\mathcal{V}_i$) into the view of p_j ($\mathcal{V}_j$), and vice versa. More precisely, the mapping shows how to transform the extension of $\mathcal{V}_i$ into the extension of $\mathcal{V}_j$, the transformation being expressed through *queries*. These queries are then used to reformulate user queries against $\mathcal{V}_j$ into queries over $\mathcal{V}_j$.

Queries are expressed in the same query language used for posing general queries, as shown in Figure 5 and in Table 2.1. These mappings link peers together, and form a sparse graph; queries are then executed by exploring the transitive closure of such mappings.[1]

For the sake of simplicity, we use bidirectional mappings, while real systems use mostly unidirectional mappings, the reverse mapping being obtained at run-time by applying LAV processing techniques, as described in [16].

Queries are expressed in an XQuery-like language, called μXQ, that is roughly equivalent to the FLWR core of XQuery, with two exceptions: 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 allow for a better handling of errors at the price of a modest decrease in the expressive power of the language: indeed, most nested queries are used without any further navigation of their results; moreover, a comparison between a variable and a constant can be simulated by binding the constant to a *let* variable in the binding section of the query.

The following Example illustrates the basic concepts of μXQ [2]

Example 2.3 Consider the following query Q posed against the Pisa view.

```
nuovaBib[
for $aut in $bib//autore
let $pap := for $a in $bib/articolo
            let $a_list := $a/autore
            where $aut isin $a_list
            return lavoro[$a/titolo, $a/anno]
return item[$aut, $pap]]
```

Note that, even though μXQ does not feature an `isin` predicate, the above query can be coded in μXQ by simply expanding the fragment '`where  $aut isin $a_list`', as shown below:

```
let $aut_list := for $aa in $a_list
                 let $aa_text := $aa/text(),
                     $aut_text := $aut/text()
                 where $aut_text = $aa_text
                 return $aa
where not empty($aut_list)
```

The query Q returns the list of authors in the database, together with the basic information about the papers they wrote. The clause `for $aut ... autore` iterates over the set of `autore` elements and binds the $\$aut$ variable to each `autore` node, hence returning a set of variable-to-node bindings. The `let` clause evaluates a nested query, whose result is a sequence of XML nodes, and binds the whole sequence to the $\$pap$ variable, hence producing a single variable binding. The nested query contains a `where` clause that checks whether the node bound to $\$aut$ in the outer query is in the set returned by the evaluation of the path expression $\$a/autore$. ■

[1] We assume that ρ_{i0} specifies how to map the peer schema of p_i into its peer view, and vice versa.

[2] For the sake of simplicity, in the rest of he paper we will use / and // in place of *child* :: and *dos* ::, respectively.

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

Figure 5: μXQ grammar

Table 2.1. *μXQ semantics*

$[\![b]\!]_\rho \triangleq b$ $\quad [\![x]\!]_\rho \triangleq \rho(x)$

$[\![\overline{x}]\!]_\rho \triangleq \rho(\overline{x})$ $\quad [\![()]\!]_\rho \triangleq ()$

$[\![Q_1, Q_2]\!]_\rho \triangleq [\![Q_1]\!]_\rho, [\![Q_2]\!]_\rho$ $\quad [\![l[Q]]\!]_\rho \triangleq l[[\![Q]\!]_\rho]$

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

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

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

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

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

$\mathit{dos}(b) \triangleq b$ $\quad \mathit{childr}(b) \triangleq ()$

$\mathit{dos}(l[f]) \triangleq l[f], \mathit{dos}(f)$ $\quad \mathit{childr}(l[f]) \triangleq f$

$\mathit{dos}(()) \triangleq ()$ $\quad \mathit{dos}(f, f') \triangleq \mathit{dos}(f), \mathit{dos}(f')$

$b :: l \triangleq ()$ $\quad l[f] :: l \triangleq l[f]$

$() :: l \triangleq ()$ $\quad (f, f') :: l \triangleq f :: l, f' :: l$

$m[f] :: l \triangleq ()$ $\quad m \neq l$

$b :: \mathit{node}() \triangleq ()$ $\quad () :: \mathit{node}() \triangleq ()$

$m[f] :: \mathit{node}() \triangleq m[f]$ $\quad (f, f') :: \mathit{node}() \triangleq f :: \mathit{node}(), f' :: \mathit{node}()$

$b :: \mathit{text}() \triangleq b$ $\quad () :: \mathit{text}() \triangleq ()$

$m[f] :: \mathit{text}() \triangleq ()$ $\quad (f, f') :: \mathit{text}() \triangleq f :: \mathit{text}(), f' :: \mathit{text}()$

```
NYBibliography <-
Q1($input):  for $ed in $input/editore
             return publisher[$ed/text()]
Q2($input):  for $an in $input/anno
             return year[$an/data()]
Q3($input):  for $t in $input/titolo
             return title[$t/text()]
Q4($input):  for $aut in $input/autore
             return author[$aut/text()]
Q5($input):  for $art in $input/articolo
             return article[Q4($art), Q3($art),Q2($art),Q6($art)]
Q6($input):  refCode['xxx-Pisa-xxx']
Q7($input):  for $bib in /pisaBib
             return bib[Q5($bib)]
```

Figure 6: Pisa → New York mapping.

```
PisaBib <-
Q1($input):  for $ed in $input/editore
             return editore[$ed/text()]
Q2($input):  for $an in $input/anno
             return anno[$an/data()]
Q3($input):  for $t in $input/titolo
             return titolo[$t/text()]
Q4($input):  for $aut in $input/autore
             let $nome := $aut/nome
             return autore[$nome/text()]
Q5($input):  for $art in $input/articolo
             return articolo[Q4($art), Q3($art),Q2($art)]
Q6($input):  for $bib in /pisaBib
             return bib[Q5($bib)]
```

Figure 7: Mapping from the Pisa schema into the Pisa view.

Systems conforming to this architecture rely on schema mappings to process and execute queries; in particular, the cost of query execution, together with the quality of the results returned by the system, are deeply connected to the quality of schema mappings. Unfortunately, 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. In particular, the dynamicity of both the topology and the schema/content of nodes can lead to the corruption of existing mappings, which can significantly affect the quality of results returned by the system. Furthermore, existing optimization techniques for p2p systems, such as the mapping composition approach described in [16], can be vanished by mapping changes; this, in turn, reflects on the result quality as well as on the query processing cost (as shown in [16] and [14], mapping composition algorithms have unbounded space complexity).

As a consequence , the ability to detect corrupted mappings between alive nodes as soon as possible becomes very important, as shown in the following Example.

Example 2.4 Consider the bibliographic data sharing system shown in Example 2.2, and suppose that Pisa uses the query of Figure 6 to map its view into the view of New York. Pisa also uses the query of Figure 7 to map its view into its schema.

Consider now the query shown in Figure 8. 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; since it expressed in terms of the Pisa view, the system rewrites the query by using the mapping of Figure 7, so to obtain a query posed against the Pisa schema. In particular, as the mapping of Figure 7 describes the Pisa view in terms of the Pisa schema, the system has to *invert* this mapping, so to obtain a mapping from the view into the schema, and then to compose the newly obtained mapping with the query. This rewriting is performed by relying on standard algorithms for rewriting query over views [14, 16].

Once the query is locally executed, 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, 16]. To illustrate how these algorithms work, consider the first `for` clause of the query; the clause searches for articolo elements nested into elements

```
articoli_Fernandez[
for $a in $bib/articolo,
    $aut in $a/autore
let $mf := ‘‘Mary F. Fernandez’’
where $aut() = $mf
return $a]
```

Figure 8: Pisa user query.

```
articoli_Fernandez[
for $a in $bib/article,
    $aut in $a/author
let $mf := ‘‘Mary F. Fernandez’’
where $aut() = $mf
return $a]
```

Figure 9: Transformed Pisa user query.

bound to $*bib*. To reformulate this clause, the system matches the path with the mapping definition, hence discovering that the path is mapped by query Q_5. By looking into the chosen mapping fragment, the system knows that the path must be rewritten as `$bib/article`, and that the remaining paths must be reformulated by using queries Q_4, Q_3, Q_2, and Q_6.

At the end of the reformulation process, the reformulated query, shown in Figure 9, is then sent to the New York site; when this query arrives at New York, it is successfully executed since it is compatible with New York peer view.

Assume now that New York slightly changes the way author names are represented, and that this schema change reflects on the peer view; instead of a simple author element, detailed information about author's first name and second name are represented: `Author = author[first[String],second[String]]`.

Now, when the Pisa user runs again her query, she is not obtaining results from New York, since the rewritten query does not match the new view of New York. Unfortunately, New York site just returns an empty sequence as result of the query, so Pisa has no way to distinguish between the error and an unsatisfied predicate. By typechecking the transformed query against the new view, the system would inform Pisa that an error is present in the query, which implies that the mapping is corrupted. ■

3 Type System

The central point in the proposed approach is the use of a type system capable of capturing incoherences between the schema specification and both the twigs and the predicates contained in a query. By relying on this feature, our technique can identify discrepancies between the transformed query and the target peer view, and provide detailed information about them.

Our type system is capable of precisely identifying type-wrong fragments of a query produced by mappings. Hence, mappings involved in the production of that wrong part (corrupted mappings) can easily be identified, as, starting from a sub-query, it is possible to retrieve the mapping that has produced that part. Also, we provide type information about the part of the database that cannot be matched by the wrong query fragment. Starting from this type information, it is possible to find an alternative and correct definition of the identified corrupted mapping.

The type system is an extension of the one presented in [5]. For reason of space, here we formalize input/output and the main properties of the type system, and explain the role these properties play in the detection of p2p corrupted mappings (some main definitions of the type system are in Appendix A).

The proposed type system is designed to statically detect the presence of two kinds of errors: sub-query emptiness and wrong comparisons in the *where* clause. It differs from that of [5] in two key points: unlike [5], we extend the check for correctness to the *where* clause and introduce a proper treatment of conditions; moreover, error messages returned by the type system contain not only the wrong sub-query, as in [5], but also the types of the schema involved in the error. We provide this detailed information since it can be very useful during mapping maintenance or debugging.

Detecting sub-query emptiness means discovering, at static time, the presence of sub-queries that, at run time, will evaluate to the empty sequence, in each valid evaluation of the query (a query evaluation is valid if it is done wrt a variable substitution which is valid wrt types of variables in the substitution). Essentially, these sub-queries consist of path expressions that never match the input data, and, therefore,

are not correct wrt to the types of input data. Detecting wrong where-comparisons, instead, means to discover, at static time, the presence of comparisons between values of different types

Discovering the presence of such errors is crucial to discover incompatibilities between the query structural requirements and input data structural specifications (input types). While type rules of the type system are omitted in this paper, these errors are characterized quite rigorously in terms of query semantics. This characterization is then used to measure the accuracy of the type analysis.

3.1 Query Correctness

The notion of correctness we propose is an extension of the notion of FE-correctness proposed in [5, 4] (where FE stands for *For each - Exists*).

As already stated, we adopt a notion of correctness which is existential, in the sense that it deems a query as correct if *for each* part of the query *there exists* a valid evaluation under which that part behaves well. Essentially, a sub-query behaves well if the action it specifies is coherent with respect to the type of the operand of that sub-query. For instance, a sub-query `for` $\overline{x}$ `in` y `return` Q' is correct if the type environment allows y to be assigned to a not empty sequence in at least one evaluation (there is something to iterate on), as well as $\overline{x}$ `child` :: l, which is an instance of $\overline{x}$ `child` :: *NodeTest*, is correct if the type environment is such that $\overline{x}$ can be assigned to a node with at least a child labeled as l, again, in at least one evaluation. In both cases, the sub-queries yield a not empty sequence for at least one evaluation. For this reason we call *Not-Empty correctness* this kind of correctness, briefly NE-correctness.

Our notion of correctness involves *where* comparisons as well. A comparison $x = y$ is correct if the type environment allows both variables to be assigned to two base values. We call *WHERE-correctness* this kind of correctness.

As an example of NE-correct query, consider the following query:

$$\$x/phone, \$x/mobile$$

where $x is of type $(data[phone[B] \mid mobile[B]])*$. This query is considered as correct, as each sub-query has a match for at least one instance of $x. Observe that queries like this are quite common in practice, and they would be discarded by a standard, universal notion of correctness (typical of traditional programming languages), according to which an expression is correct if each part of it behaves well under each possible evaluation. Indeed, here, the sub-query $x/phone is always empty in the case the database only contains numbers of mobile phones, and for this reason the whole query would be considered as incorrect under universal correctness.

For WHERE-correctness, we adopt an existential approach as well. Before seeing a motivating example, recall that, in standard programming languages, a binary comparison involved in a logical condition is correct if the two involved arguments have the same type. Here we have to relax this notion, thus resorting to an existential notion of correctness, as otherwise common queries like the following one would be discarded.

```
for $a in $bib/article
let $n := $a/author/text()
let $m := $a/author/second/text()
where $n = 'Fernandez' or $m = 'Fernandez'
return $a/title
```

with $bib of type

```
NYBib = bib[(Article|Book)*]
Article = article[Author*,Title,Year, RefCode]
Author = author[String | first[String],second[String]]
Title = title[String]
....
```

Note that author content may be a simple string or a complex value composed of two elements. Above, if in the query input all author names are complex values, the first *where* condition compares two incompatible values, and similarly for the second condition when all names are simple strings. However, when both kinds of author name appear, both comparisons make sense.

In our system a comparison is correct if there is at least one evaluation under which two compatible values are compared, as in the case above. This corresponds to consider a comparison as correct if the types of the two arguments are both super type of the base type, according to the standard notion of XML subtyping [11].

To characterize and to check correctness we have to formally assign an identifier to each query sub-expression. To this end, we will need the operation $(Q)_{|\beta}$, which, for any query Q and *location* β, locates the corresponding sub-query. The location β is just a path of 0's and 1's, and the function $(Q)_{|\beta}$ follows

β in a walk down the syntax tree of Q, without considering possible *where* clauses. As we will see, this in this way we will be also able to precisely locate *where* conditions.

Definition 3.1 (**$(Q)_{|\beta}$**) *$(Q)_{|\beta}$ denotes the sub-term of the query Q located by the location β, which is a sequence of 0's and 1's:*

$$
\begin{array}{lcll}
(Q)_{|\epsilon} & \triangleq & Q & \\
(l[Q])_{|0.\beta} & \triangleq & (Q)_{|\beta} & \\
(Q_0, Q_1)_{|i.\beta} & \triangleq & (Q_i)_{|\beta} & i \in \{0,1\} \\
(\texttt{for } \overline{x} \texttt{ in } Q_0 \texttt{ return } Q_1)_{|i.\beta} & \triangleq & (Q_i)_{|\beta} & i \in \{0,1\} \\
(\texttt{let } x ::= Q_0 \texttt{ return } Q_1)_{|i.\beta} & \triangleq & (Q_i)_{|\beta} & i \in \{0,1\} \\
(\texttt{for } \overline{x} \texttt{ in } Q_0 \texttt{ where } P \texttt{ return } Q_1)_{|i.\beta} & \triangleq & (Q_i)_{|\beta} & i \in \{0,1\} \\
(\texttt{let } x ::= Q_0 \texttt{ where } P \texttt{ return } Q_1)_{|i.\beta} & \triangleq & (Q_i)_{|\beta} & i \in \{0,1\} \\
(Q)_{|\beta} & \triangleq & \bot & \textit{otherwise}
\end{array}
$$

We also define $Locs(Q) = \{\beta \mid (Q)_{|\beta} \neq \bot\}$.

In order to define NE-correctness, we first define the set $CriticalLocs(Q)$ of the locations of Q where we will look for pieces of wrong paths.

$$
\begin{array}{rcl}
CriticalLocs(Q) & \triangleq & \{\beta \mid ((Q)_{|\beta} = (\overline{x}\ \texttt{child} :: NodeTest) \ \vee \ (Q)_{|\beta} = (\overline{x}\ \texttt{dos} :: NodeTest))\} \ \cup \\
 & & \{\beta.0 \mid (Q)_{|\beta} = \texttt{for } \overline{x} \texttt{ in } Q_0 \texttt{ return } Q_1\}
\end{array}
$$

$CriticalLocs(Q)$ does not coincide with $Locs(Q)$ because, at least, all locations that reach a sub-query that is () must not be tested for non-emptiness. But we can also observe that a `let` sub-query evaluates to () if and only if the `return` sub-query does, hence, once we have indicated that the `return` sub-query has a problem, the same information about the whole `let` sub-query is redundant. A similar consideration holds for a Q_0, Q_1 sub-query: once the sub-queries Q_0 and Q_1 have been checked, any information about the fact that the whole Q_0, Q_1 evaluates to () is redundant. After a complete analysis, one realizes that only errors located in sub-queries from which the programmer explicitly started a `child`/`dos` navigation or a `for` iteration should be considered.

To formalize our notion of correctness, we define $Ext(\rho, Q, \beta)$ as the set of all valid substitutions that will be used to evaluate the sub-query $(Q)_{|\beta}$ when Q is evaluated under ρ. These substitutions correspond to ρ extended with the bindings introduced by each traversed `let` or `for`. $Ext(\rho, Q, \beta)$ is not just a singleton since each sub-query in the scope of a `for` $\overline{x}$ `in` Q_0 is evaluated once for each tree in the forest f returned by Q_0 under ρ. As f may be the empty forest, $Ext(\rho, Q, \beta)$ may be empty as well.

In calculating $Ext(\rho, Q, \beta)$ we abstract away from *where* conditions, since we want a notion of correctness that only involves structural type information about input and actions specified by sub-queries. We do not consider as incorrect an expression which always evaluates to () because of a *where* condition which is *always* false, as in

`for $x in a[] where` *Condition* `return $x`

This query is always empty if and only if the expression *Condition* is always false. Hence, we have an NE-error iff *Condition* is always false.

The problem of detecting the presence of conditions that are *always* false is, in general, strictly connected to a particular content of the database, and in general out of the scope of a type system. Here, our aim is to inform the user of a p2p system only about conformance between the structural requirements of the query and the type structure of the query inputs.

As our notion of NE-correctness is up-to the filtering imposed by the *where* clause, we will use $\{\!|Q|\!\}_\rho$ to denote the semantics of Q up to the *where* clause; $\{\!|Q|\!\}_\rho$ can be defined as $[\![\textit{drop-where}(Q)]\!]_\rho$, where *drop-where*$(Q)$ is the query obtained from Q by just dropping all its *where* sub-expressions; as a consequence, $\{\!|Q|\!\}_\rho$ always contains, in some sense, $[\![Q]\!]_\rho$. Over $\{\!|\cdot|\!\}_\rho$ we can define a query result containment relation, based on the following structural containment relation among forests.

Definition 3.2 (Forest containment) *Over forests, we define the containment relation $\ll$ as follows*

$$
\begin{array}{lcl}
b \ll b & & \\
() \ll f & & \\
f' \ll f'' & \Rightarrow & f, f' \ll f, f'' \\
f' \ll f'' & \Rightarrow & f', f \ll f'', f \\
f' \ll f'' & \Rightarrow & l[f'] \ll l[f''] \\
f' \ll f \ \wedge f \ll f'' & \Rightarrow & f' \ll f''
\end{array}
$$

Moreover, we extend $\ll$ to substitutions ρ in the obvious way:

$$\rho \ll \rho' \;\Leftrightarrow\; (\chi \mapsto f \in \rho \;\Rightarrow\; \chi \mapsto f' \in \rho' \wedge f \ll f')$$

Proposition 3.3 (Query result containment) *If $\rho \ll \rho'$ then $[\![Q]\!]_\rho \ll \{\!|Q|\!\}_\rho$*

Definition 3.4 (Substitution Extension)

$$\begin{aligned}
&Ext(\rho,\, Q,\, \epsilon) && \triangleq \{\rho\}\\
&Ext(\rho,\, \texttt{let}\ x ::= Q_0\ \texttt{return}\ Q_1,\, 1.\beta) \\
&\qquad && \triangleq Ext((\rho, x \mapsto \{\!|Q_0|\!\}\rho),\, Q_1,\, \beta)\\
&Ext(\rho,\, \texttt{for}\ \overline{x}\ \texttt{in}\ Q_0\ \texttt{return}\ Q_1,\, 1.\beta) \\
&\qquad && \triangleq \textstyle\bigcup_{t \in trees(\{\!|Q_0|\!\}\rho)} Ext((\rho, \overline{x} \mapsto t),\, Q_1,\, \beta)\\
&Ext(\rho,\, \texttt{let}\ x ::= Q_0\ \texttt{where}\ P\ \texttt{return}\ Q_1,\, 1.\beta) \\
&\qquad && \triangleq Ext((\rho, x \mapsto \{\!|Q_0|\!\}\rho),\, Q_1,\, \beta)\\
&Ext(\rho,\, \texttt{for}\ \overline{x}\ \texttt{in}\ Q_0\ \texttt{where}\ P\ \texttt{return}\ Q_1,\, 1.\beta) \\
&\qquad && \triangleq \textstyle\bigcup_{t \in trees(\{\!|Q_0|\!\}\rho)} Ext((\rho, \overline{x} \mapsto t),\, Q_1,\, \beta)\\
&\textit{otherwise: } (Q)_{|i} \neq \bot \Rightarrow Ext(\rho,\, Q,\, i.\beta) && \triangleq Ext(\rho,\, (Q)_{|i},\, \beta)
\end{aligned}$$

NE-correctness can be formally captured in terms of substitution extension. A non-() sub-query $(Q)_{|\beta}$ is correct if there exist $\rho \in \mathcal{R}$ and $\rho' \in Ext(\beta,\, Q,\, \rho)$ such that $\{\!|(Q)_{|\beta}|\!\}\rho' \neq ()$. Indeed, if such a substitution cannot be found, $(Q)_{|\beta}$ is useless to the whole query, and is hence incorrect.

We can now formalize query correctness. We first define NE-correctness and then WHERE-correctness.

Definition 3.5 (NE-correctness of Q w.r.t. $\mathcal{R}$) *Let $\mathcal{R}$ be a set of substitutions for the free variables of a query Q. Q is correct w.r.t. $\mathcal{R}$ iff:*

$$\forall \beta \in CriticalLocs(Q).\quad \exists \rho \in \mathcal{R}.\ \exists \rho' \in Ext(\rho,\, Q,\, \beta).\ \{\!|(Q)_{|\beta}|\!\}\rho' \neq ()$$

Dually, Q has a NE error at path $\beta \in CriticalLocs(Q)$ w.r.t. $\mathcal{R}$ iff:

$$\forall \rho \in \mathcal{R}.\ \forall \rho' \in Ext(\rho,\, Q,\, \beta).\ \{\!|(Q)_{|\beta}|\!\}\rho' = ()$$

(Observe that $Ext(\rho,\, Q,\, \beta) = \emptyset$ implies that Q has an error at β.)

The machinery we have introduced to characterize NE-correctness can be used to define WHERE-correctness as well. To this end, we only need a couple of (quite intuitive) definitions, and to observe that, if $(Q)_{|\beta} = \texttt{for}\ \overline{x}\ \texttt{in}\ Q_0\ \texttt{where}\ P\ \texttt{return}\ Q_1 \mid \texttt{let}\ x ::= Q_0\ \texttt{where}\ P\ \texttt{return}\ Q_1$, then the set of possible substitutions, up-to *where* filtering, under which each condition of P is evaluated, is $Ext(\rho,\, Q,\, \beta.1)$, the same set of substitutions for Q_1.

Definition 3.6 *Given a query Q containing a condition $\chi_1\ \delta\ \chi_2$ or $\texttt{empty}(\chi)$, we say that the condition is a β-condition in Q if β is such that*

$$\begin{array}{lcll}
(Q)_{|\beta} & = & \texttt{for}\ \overline{x}\ \texttt{in}\ Q_0\ \texttt{where}\ P\ \texttt{return}\ Q_1 & \textit{or}\\
(Q)_{|\beta} & = & \texttt{let}\ x ::= Q_0\ \texttt{where}\ P\ \texttt{return}\ Q_1 &
\end{array}$$

with P containing the condition $\chi_1\ \delta\ \chi_2$, or $\texttt{empty}(\chi)$.

Definition 3.7 (WHERE-correctness of Q w.r.t. $\mathcal{R}$) *Let $\mathcal{R}$ be a set of substitutions for the free variables of a query Q. Q is WHERE-correct w.r.t. $\mathcal{R}$ iff for all β:*

$$\begin{array}{rl}
\forall \beta\textit{-condition } \chi_1\ \delta\ \chi_2. & \exists \rho \in \mathcal{R}.\ \exists \rho' \in Ext(\rho,\, Q,\, \beta.1).\quad (\rho'(\chi_1) = b_1\ \wedge\ \rho'(\chi_2) = b_2)\\
\forall \beta\textit{-condition } \texttt{empty}(\chi). & \exists \rho \in \mathcal{R}.\ \exists \rho' \in Ext(\rho,\, Q,\, \beta.1).\quad (\rho'(\chi) \neq b)
\end{array}$$

Dually, w.r.t. $\mathcal{R}$, the query Q has a WHERE-error

- *in the β-condition $\chi_1\ \delta\ \chi_2$*

$$\forall \rho \in \mathcal{R}.\ \forall \rho' \in Ext(\rho,\, Q,\, \beta.1).\quad \nexists b_1, b_2.(\rho'(\chi_1) = b_1\ \wedge\ \rho'(\chi_2) = b_2)$$

- *in the β-condition* $\texttt{empty}(\chi)$*:*

$$\forall \rho \in \mathcal{R}.\ \forall \rho' \in Ext(\rho,\, Q,\, \beta.1).\quad (\rho'(\chi) = b)$$

The last case says that, w.r.t. $\mathcal{R}$, it is never the case that the actual value for χ in the β-comparison $\texttt{empty}(\chi)$ *is a list, hence that condition is an error.*

The notion of NE-correctness provided here perfectly coincides with the notion of FE-correctness given in [5], while the notion of WHERE-correctness is completely new, as in [5] *where* clauses were not considered.

As testified by our motivating scenario, the proposed notions of correctness play a crucial role in characterizing correctness of p2p mappings, as correctness ensures that there exists a strong adherence between structural specifications in the mapped query and the peer view or schema.

3.2 Type Environments and Types

We adopt, essentially, XDuce's type language [12]. Types and type environments are defined as follows:

Types	T	::=	$()$	*empty forest type*
		|	B	*base type*
		|	T, T	*product type*
		|	$T \mid T$	*union type*
		|	$l[T]$	*element type*
		|	$T*$	*repetition type*
		|	X	*type variable*
Environments	E	::=	$()$	
		|	$X = T,\ E$	

Base Types B ::= `String`

An element type with empty content $l[()]$ will always be abbreviated as $l[]$. A type environment E is a sequence of type definitions of the form $X = T$, where no type variable is bound to two distinct types; $E(X)$ denotes the type bound to X by E.

We restrict to $l[]$-guarded type environments, that are environments where only $l[]$-guarded vertical recursion is allowed, as in $X = l[X \mid ()]$, for instance; we forbid equations like $X = X \mid ()$ and $X = X, Y$. The lack of horizontal recursion is counterbalanced by the presence of the Kleene star operator $*$. This restriction is canonical, and makes the type language as expressive as regular tree languages [13, 6].

Type semantics is standard: $[\![\_]\!]_E$ is the minimal function from types to sets of forests that satisfies the following monotone equations (the function is well-defined by Knaster-Tarski theorem):

$$\begin{array}{lcllcl}
[\![()]\!]_E & \triangleq & \{()\} & [\![B]\!]_E & \triangleq & \{b\} \\
[\![l[T]]\!]_E & \triangleq & \{l[f] \mid f \in [\![T]\!]_E\} & [\![X]\!]_E & \triangleq & [\![E(X)]\!]_E \\
[\![T, T']\!]_E & \triangleq & \{f, f' \mid f \in [\![T]\!]_E,\ f' \in [\![T']\!]_E\} & [\![T \mid T']\!]_E & \triangleq & [\![T]\!]_E \cup [\![T']\!]_E \\
[\![T*]\!]_E & \triangleq & \{(), f_1, \ldots, f_n \mid n \geq 0,\ f_i \in [\![T]\!]_E\} & & &
\end{array}$$

An environment E is well-formed only if it is $l[]$-guarded and defines type with non-empty semantics, i.e. empty-type definitions like $X = l[X]$ are not allowed. A type T is well-formed in an environment E if every variable in T is defined in E.

The type assignments for the free variables of a query are defined by means of *variable environments* Γ of the form:

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

A variable environment Γ is well-formed, w.r.t. an environment E, if no variable is defined twice, if every type is well-formed in E, and if every for-variable $\overline{x}$ is associated to a tree type ($l[T']$ or B).

Our type rules prove judgments of the form :

$$\textbf{Judgments} \quad J ::= E;\ \Gamma \vdash_\beta Q : (T;\ \mathcal{S};\ \mathcal{W})$$

In $E;\ \Gamma \vdash_\beta Q : (T;\ \mathcal{S};\ \mathcal{W})$, the type T is the result type of Q, and defines an upper bound for the actual set of values for Q.

In order to compute NE-errors, our typing judgments also return an error set $\mathcal{S}$, which contains a set of pairs $(\beta.\alpha, \mathcal{T})$, where $\mathcal{T}$ is a set of types, such that:

- the sub-query of Q at α is not NE-correct;

- types that have failed to match the sub-query at α are in $\mathcal{T}$.

Therefore, we aim at signaling both positions of wrong sub-queries and parts of the schema to which a wrong sub-query has failed to match. In this way, the programmer knows what part of the query is wrong and why, as she also has type information pertinent to the wrong sub query, thus allowing faster and more accurate query correction. Of course, to give meaning to type components in $\mathcal{T}$, the global type environment E is needed.

Similar considerations hold for the error-set $\mathcal{W}$ in $E;\ \Gamma \vdash_\beta\ Q : (T;\ \mathcal{S};\ \mathcal{W})$. It is a set containing two kinds of pairs:

- $(\beta.\alpha, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2)))$, each meaning that in the query there is an α-condition that is a WHERE-error, because of χ_1 and χ_2 assuming values in incompatible types in $\mathcal{T}_1$ and $\mathcal{T}_2$;
- $(\beta.\alpha, (\chi, \mathcal{T}))$, each meaning that in the query there is an α-condition $\texttt{empty}(\chi)$ that is a WHERE-error, because of χ assuming values in types in $\mathcal{T}$, which does not contain any type with sequence values (recall that `empty` is only defined over sequences).

.

4 Property of the Type System

The first property enjoyed by our the type system is soundness, this is expressed in terms of valid substitutions:

Definition 4.1 ($\mathcal{R}(E, \Gamma)$) *For any well-formed type environment E and Γ well-formed in E, we define the set of valid substitutions as*

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

Theorem 4.2 (Soundness of Error-Checking) *For each query Q, and Γ well-formed in E:*

$$\begin{aligned} &E;\ \Gamma \vdash_\beta\ Q : (\_;\ \mathcal{S}; \mathcal{W}) \ \Rightarrow \\ &\quad (\beta.\alpha, \mathcal{T}) \in \mathcal{S} \ \Rightarrow\ Q \text{ has an FE error at } \alpha \text{ w.r.t. } \mathcal{R}(E, \Gamma) \\ &\quad (\beta.\alpha, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2)) \in \mathcal{W} \ \Rightarrow\ Q \text{ has a where error at the } \alpha\text{-condition } \chi_1\ \delta\ \chi_2 \text{ w.r.t. } \mathcal{R}(E, \Gamma) \\ &\quad (\beta.\alpha, (\chi, \mathcal{T})) \in \mathcal{W} \ \Rightarrow\ Q \text{ has a where error at the } \alpha\text{-condition } \texttt{empty}(\chi) \text{ w.r.t. } \mathcal{R}(E, \Gamma) \end{aligned}$$

Completeness does not hold in general. For NE-errors, it holds when we restrict to type environments E which are $*$-guarded [5, 4]. These are defined as environments in which recursion is guarded by a $*$ type constructor. Hence, a $*$-guarded type environment does not allow definitions like $Y = l[Y] \mid m[Y] \mid ()$, while it allows $*$-guarded definitions like $Y = (l[Y] \mid m[Y])* \mid ()$. We have no space here to explain why completeness does not hold in general, and why it holds under this restriction (to this regard, a clear treatment can be found in [5, 4], while a shorter explanation is in the Appendix).

The restriction to $*$-guarded type environments captures most cases, hence is a *mild* restriction. By analyzing repositories over the web, indeed, we realized that in practice most DTDs and XML Schema definitions are $*$-guarded, including *all* the schemas reported in the W3C document "XML Query Use Cases" [3].

Theorem 4.3 (Completeness of NE-Error-Checking) *For each query Q, $*$-guarded E, and Γ $*$-guarded and well-formed in E:*

$$\begin{aligned} &E;\ \Gamma \vdash_\beta\ Q : (\_;\ \mathcal{S}; \mathcal{W}) \ \Rightarrow \\ &\qquad ((\beta.\alpha, \mathcal{T}) \in \mathcal{S} \ \Leftrightarrow\ Q \text{ has a NE-error at } \alpha \text{ w.r.t. } \mathcal{R}(E, \Gamma)) \end{aligned}$$

Although we state completeness in the context of Γ $*$-guarded variable environments, this is not a limitation, since, when E is $*$-guarded, it is possible to split any well-formed Γ into an equivalent set of $*$-guarded Γ_i's, and to type-check queries w.r.t. this set of environments.

We have not yet proved completeness result with respect to where-errors, nor we have proved formal properties regarding the type components in error sets. However, all the counter-examples to completeness of our standard algorithm are correctly analyzed by the complete system; since the kind of case analysis over which is based our complete algorithm is quite powerful, and is a guarantee of strong adherence between static and dynamic semantics, we are quite confident that completeness extends to WHERE-errors as well.

We want to outline that, despite the heavy use of case analysis, our algorithm runs efficiently in most practical cases (see [5, 4] for a discussion about this).

Also, several applications of the type systems to some use cases revealed that type information in error sets are quite precise and useful in discovering and adjusting corrupted peer mappings.

Regarding our completeness result, it is worth observing that, in order to detect corrupted mappings between peers, completeness of the type system is crucial. Indeed we have that if Q is NE-correct with respect to $\mathcal{R}(E,\Gamma)$ and a type change transforms $((E,\Gamma))$ into $((E',\Gamma'))$ such that Q now contains a NE error at β with respect to $\mathcal{R}(E',\Gamma')$, then the type system will detect the presence of such error, by also providing quite precise information about that type variation entailing the error. As testified by our motivating example, this is crucial to fix corrupted mappings.

5 Distributed Typechecking Algorithm

The approach we are proposing is based on the idea of typechecking queries at each involved peer. Consider a query Q originated at peer p_i, and assume that the query traverses peers p_2, ..., p_n, hence being rewritten by the mapping chain $\{\rho_1,\ldots,\rho_{n-1}\}$ (ρ_i is the mapping from p_i to p_{i+1}); the sequence of peers to be traversed can be precomputed on the basis of a crawling process, or it can be determined at run-time (the proposed approach is orthogonal to the algorithms used for finding the mapping chain corresponding to a given query Q). Let ϖ be the function transforming queries according to a given mapping. When $\varpi_{\rho_1\circ\ldots\circ\rho_{j-1}}(Q)$ arrives at peer p_j, p_j type-checks it against its peer view in order to verify that the query requirements are *compatible* with its view; if the type checker raises an error, then the system knows for sure that an error is present in the mapping ρ_{j-1} (the mapping from p_{j-1} to p_j), so it can warn p_{j-1} and p_j about the problem.

Thus, we assume that queries, once transformed, are type-checked by any peer they reach before executing them; since the query execution cost is dominated by the communication costs, the adoption of this strategy should not affect the whole execution cost.

The *distributed* type-checking algorithm is shown in Algorithm 1. At this time, the algorithm is not able to locate the portion of the mapping containing errors, even though we believe that this extension should not be too complex.

Algorithm 1 Distributed type-checking algorithm

Main Algorithm (at peer $p=(id,db,\mathcal{T},\mathcal{V},\{\rho_{ij}\}_j)$**)**
type-check locally Q against p view ($\mathcal{V}$)
if error **then**
  send user a warning $(Q,\mathcal{S},\mathcal{W},E)$
  stop
else
  find a mapping chain for Q
  $chain = <\rho_i, p_i>_i$
  $Q_1 = \varpi_{\rho_1}(Q)$
  send(p_2,Q_1, typecheck, *chain*)
  wait for errors
end if
TypeCheckReceive(*peer*, Q', typecheck, *chain*)
typecheck locally Q' against *peer* view
if error **then**
  send(sender,"error", $(Q,\mathcal{S},\mathcal{W},E)$)
  stop
else
  $\rho_i = nextMapping(chain)$
  $Q'' = \varpi_{\rho_i}(Q')$
  send(p_{i+1},Q'', typecheck, *chain*)
  wait for errors
end if

As shown by the algorithm, the identification of errors in the mapping chain can benefit the query processing algorithm, since it allows the system to dynamically bypass potentially dangerous mappings. This is performed by backtracking to peer p_{j-1} (if an error was found in the mapping from p_{j-1} to p_j), and by recomputing the remaining part of the mapping chain. By doing so, the system could avoid global problems coming from the propagation of errors in the mapping chain.

To illustrate the effectiveness of the proposed approach, we can consider again the scenario described in Example 2.4, and see what happens when the transformed query arrives at New York. When the New York peer receives the transformed query Q', it starts type-checking Q' against its peer view. The local type-checking algorithm stops during the control for the correctness of the *where* clause (Rule

CompWrong of Appendix A), since the query tries to compare a complex element with a base type value. The system, hence, notifies to Pisa and New York that an error in the sub-query `where $aut = $mf` wrt the type *Author* has occurred; by using this information, Pisa can update its mapping to New York and/or change the routing for the query. More in details, the β returned by the type-checker, pointing to the error `where $aut = $mf` and sent to Pisa, tells Pisa how to directly pick up the corrupted mapping (its easy to find which mapping is involved in the production of a β sub-query). While the not matched type *Author* tells Pisa how to modify the corrupted mapping.

More formally, we can say that if a peer p is notified of an error message $(Q, \mathcal{S}, \mathcal{W}, E)$, with, say, $(\beta.\alpha, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2))) \in \mathcal{W}$, then p can rely on a systematic technique that uses information in $(Q, \mathcal{S}, \mathcal{W}, E)$ in order to

- retrieve a quite precise set of candidate corrupted mappings which contains effective corrupted mappings;
- suggest, by using the type information in $\mathcal{S}, \mathcal{W}$, and E, a set of updated mappings, fixing the issues in the previous mappings.

This sheds light about the different roles of error-positions and not-matched types, both computed by our type-checker; the formal definition of the above mentioned systematic approach for retrieval-correction of corrupted mappings is the core of our current efforts in continuing this work, and also represents the formal connection between our type analysis technique and the p2p infrastructure.

6 Related Works

We are not aware of studies related to the use of the static analysis for discovering corrupted mappings in p2p systems. The advantages of our type analysis with respect to current version of W3C XQuery type system [7] has already been discussed in [5]. In a nutshell, our type system ensures complete type inference and error checking in most cases, while ensuring good performance at the same time. Being the definition of W3C XQuery type system still a work in progress, no results are known about the complexity of the whole system, and the system does not ensure completeness of path error-checking (see [5]). Moreover, for the moment, W3C type system does not contain mechanisms able to precisely locate the position of wrong sub-queries. We believe that this problem is not difficult to solve for an implementor of the W3C type system; however, from the experience we gained in our previous investigations, we understood that the problem is definitively not obvious.

We conclude by observing that, in order to have a powerful type-checking algorithm, in our setting we can benefit from the absence of many features that complicate typing in a full XQuery language, but are often unnecessary in p2p systems.

7 Conclusions

This paper presented a technique for automatically discovering corrupted mappings in peer data management systems. This technique is based on the use of a distributed type-checking algorithm that type-checks a query at any involved peer; by matching a (transformed) query against the peer view of a given peer, the algorithm can find inconsistencies in the mapping used for transforming the query, and, then, warn the user about problems in the mapping.

The distributed type-checking algorithm exploits a local type-checking algorithm, based on a type system extending the one of [5]; this new type system is able to capture errors in the query paths as well as errors in the *where* clause, and returns detailed information about the errors found in the query; in particular, the type-checker specifies the sub-query containing errors as well as the fragment of peer view against which the query failed.

As a future work, we plan to further extend the type system so to precisely identify the fragment of mapping containing errors, as well as to develop techniques for suggesting updated mappings: with this supplementary information, the site owner can easily and quickly fix the corrupted mapping. Furthermore, we plan to extend the fragment of XQuery covered by the type system as well as to shift to an XQuery-like type system.

References

[1] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. *First Monday*, 10(5), 2000.

[2] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 241–251, 2004.

[3] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. Technical report, World Wide Web Consortium, 2003. W3C Working Draft.

[4] Dario Colazzo. *Path Correctness for XML Queries: Characterization and Static Type Checking.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 2004.

[5] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. 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.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tisonand, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`.

[7] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, February 2004. W3C Working Draft.

[8] Enrico Franconi, Gabriel M. Kuper, Andrei Lopatenko, and Ilya Zaihrayeu. Queries and Updates in the coDB Peer to Peer Database System. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1277–1280, 2004.

[9] Marc Friedman, Alon Y. Levy and Todd D. Millstein. Navigational Plans for Data Integration. In Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration, Held on July 31, 1999 in conjunction with the Sixteenth International Joint Conference on Artificial Intelligence City Conference Center, Stockholm, Sweden.

[10] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. 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*, pages 556–567. ACM, 2003.

[11] H. Hosoya. Regular Expression Types for XML. PhD thesis, The University of Tokyo, December 2000.

[12] Haruo Hosoya and Benjamin C. Pierce. XDuce: An XML Processing Language, 1999. In *Proceedings of the Third International Workshop on the Web and Databases, WebDB 2000, Adam's Mark Hotel, Dallas, Texas, USA, May 18-19, 2000, in conjunction with ACM PODS/SIGMOD 2000*, pages 111-116, 2000.

[13] D. Lee, M. Mani, and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. 2000. Technical Report - IBM Almaden Research.

[14] Jayant Madhavan and Alon Y. Halevy. Composing mappings among data sources. In *VLDB*, pages 572–583, 2003.

[15] Luciano Serafini, Fausto Giunchiglia, John Mylopoulos, and Philip A. Bernstein. Local relational model: A logical formalization of database coordination. In Patrick Blackburn, Chiara Ghidini, Roy M. Turner, and Fausto Giunchiglia, editors, *Modeling and Using Context, 4th International and Interdisciplinary Conference, CONTEXT 2003, Stanford, CA, USA, June 23-25, 2003, Proceedings*, volume 2680 of *Lecture Notes in Computer Science*, pages 286–299. springer, 2003.

[16] Igor Tatarinov and Alon Y. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD Conference*, pages 539–550, 2004.

[17] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[18] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

A Type Rules and Auxiliary Definitions

In this appendix, we give the main definitions about the type system on which our distributed type analysis for p2p systems is based. Namely, we provide the definitions of all the operators used in the type rules, the main properties characterizing these operators, and the type rules. Moreover, relying on our work [5], we give some intuitions about the technique we use to ensure completeness of NE error checking.

Our type rules prove judgments of the form :

$$\begin{array}{rll} \textbf{Judgments} & J ::= & E;\ \Gamma \vdash_\beta\ Q : (T;\ \mathcal{S};\ \mathcal{W}) \\ & & E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q : (T;\ \mathcal{S};\ \mathcal{W}) \\ & & E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q\ \texttt{where}\ P : (T;\ \mathcal{S};\ \mathcal{W}) \\ & & E;\ \Gamma \vdash_\beta\ P : (\mathcal{W}) \end{array}$$

In $E;\ \Gamma \vdash_\beta\ Q : (T;\ \mathcal{S};\ \mathcal{W})$, the type T is the result type of Q, and defines an upper bound for the actual set of values for Q; the role of β, $\mathcal{S}$ and $\mathcal{W}$ will be discussed shortly.

A judgment J is well-formed (written $WF(J)$) if the involved types and environments are well-formed, and if all free variables in Q are defined in Γ. The judgment $E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q\ \texttt{where}\ P : (T;\ \mathcal{S};\ \mathcal{W})$ is used to type-check for-iterations, as explained below.

To analyze $\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2$, we compute a type T_1 for Q_1 and use the judgment $E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2\ \texttt{where}\ P : (T_2;\ \_;\ \_)$ to check P and to compute the type of Q_2, through a case-analysis on the type T_1 (rules (TypeIn...)). The analysis of $\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2$ is made similarly, by proving $E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : (T_2;\ \_;\ \_)$, with T_1 the type of Q_1.

The use of case analysis is crucial to have quite precise type inference (see [5] for a formal estimation of precision), which in turn is crucial to have a sound and complete error-checking algorithm.

Type checking of conditions in P is made by proving a judgment $E;\ \Gamma' \vdash_\beta\ P : (\mathcal{W})$, typically when case-analysis stops, that is when a tree type $l[T]$ or B is met; $l[]$-guardedness of E implies that recursive type-variables do not make case-analysis loop forever.

Rule (TypeInElSplitting) and Rule (TypeLetSplitting) use the function $Split_E(T)$, to be discussed later. For now, we simply define $Split_E(T) = \{T\}$.

More in details, Rule (TypeFor) starts the case-analysis, as described above, propagates the error sets $\mathcal{S}_1$ and $\mathcal{W}_1$, and adds an error $\beta.0$ if the type of Q_1 only contains the empty forest (β is a current-location parameter propagated and updated by the rules). It uses the auxiliary judgment $T \sim_E ()$, which checks whether $[\![T]\!]_E = [\![()]\!]_E$, and is defined below.[3]

Rules (TypeInUnion) and (TypeInConc) perform the case analysis, and only put in the error sets those locations that are wrong in both branches.

Rule (TypeInElSplitting) stops the case-analysis, inserts the assumption $\overline{x} : m[T]$ in Γ, starts the analysis of the where condition P, and falls back to standard type-checking (recall that we assumed $Split_E(T) = \{T\}$).

Rule (TypeChild) requires the type of $\overline{x}$ to be a tree type $m[T']$, uses $E \vdash\ T' :: NodeTest \Rightarrow U$ (defined below) to restrict the content type T' to the tree types with structure satisfying *NodeTest*, and puts an error location β in $\mathcal{S}$ iff the restricted type U is equivalent to the type () (which is an easy test).

Rule (TypeDos) is similar, but, 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 $\text{Trees}_E(T)$ defined below, and defines a new type $U' = (U_1 \mid \ldots \mid U_n)*$. U' is the type of any forest that only contains 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 $\overline{x}\ \texttt{dos} :: NodeTest$ is obtained by restricting U' to the tree types with structure satisfying *NodeTest*.

Rule (TypeLetSplitting) is standard, since we are assuming that $Split_E(T) = \{T\}$. We will later relax this assumption.

Rules to check WHERE conditions P are quite standard. In Rule (CompOk), $E \vdash B \leq \Gamma(\chi_i)$ means that B is a subtype of $\Gamma(\chi_i)$ (subtyping for regular tree types is the standard XDuce subtyping, so we avoid here to present the sub typing definition and algorithm; type equality is derived from subtyping in the obvious way).

As said before, in order to compute NE errors, our typing judgments return an error set $\mathcal{S}$, which contains a set of pairs $(\beta.\alpha, \mathcal{T})$, where $\mathcal{T}$ is a set of types, such that:

- the sub-query of Q at α is not NE-correct;
- types that have failed to match the sub-query at α are in $\mathcal{T}$.

Over sets $\mathcal{S}$ we define two operations $\sqcup$ and $\sqcap$ in the following way:

[3] Recall that, the type () is not the empty type. It is a singleton type, which only contains the empty forest.

$$
\begin{array}{rcl}
\mathcal{S}_1 \sqcup \mathcal{S}_2 & = & \{(\beta, \mathcal{T}) \mid \exists! i \in \{1,2\}.(\beta, \mathcal{T}) \in \mathcal{S}_i\} \cup \\
& & \{(\beta, \mathcal{T}_1 \cup \mathcal{T}_2) \mid (\beta, \mathcal{T}_1) \in \mathcal{S}_1, (\beta, \mathcal{T}_1) \in \mathcal{S}_2\} \\
\mathcal{S}_1 \sqcap \mathcal{S}_2 & = & \{(\beta, \mathcal{T}_1 \cup \mathcal{T}_2) \mid (\beta, \mathcal{T}_1) \in \mathcal{S}_1, (\beta, \mathcal{T}_1) \in \mathcal{S}_2\}
\end{array}
$$

Typically, $\sqcap$ is used in case analysis over union types. For instance, assume that we want to analyze a query $\texttt{for}\ \overline{x}\ \texttt{in}\ y\ \texttt{return}\ \overline{x}/label$ in an environment assigning y to the type $T \mid U$. In this case, as explained before, the type analysis of the query is split in two different analysis, the first one over the type T and the second one over the type U, resulting, respectively, in the error sets $\mathcal{S}_T$ and $\mathcal{S}_U$. Due to the existential nature of NE correctness, the location α relative to $\overline{x}/label$ is an error only if it is an error for both cases T and U. This is why the final error set for $\texttt{for}\ \overline{x}\ \texttt{in}\ y\ \texttt{return}\ x/label$ is $\mathcal{S}_T \sqcap \mathcal{S}_U$ (see rule (TypeInUnion)).

Union $\sqcup$ is used, in particular, for queries Q_1, Q_2 (rule (TypeForest)). Indeed, if for Q_1 and Q_2 we have inferred the two error sets $\mathcal{S}_{Q_1}$ and $\mathcal{S}_{Q_2}$, then the error set of Q_1, Q_2 is the union of errors $\mathcal{S}_{Q_1} \sqcup \mathcal{S}_{Q_2}$.

Over sets $\mathcal{W}$, $\sqcup$ and $\sqcap$ are defined as follows:

$$
\begin{array}{rcl}
\mathcal{W}_1 \sqcup \mathcal{W}_2 & = & \{(\beta, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2))) \mid \exists! i \in \{1,2\}.(\beta, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2))) \in \mathcal{W}_i\} \cup \\
& & \{(\beta, ((\chi_1, (\mathcal{T}_1 \cup \mathcal{T}_1'))\delta(\chi_2, (\mathcal{T}_2 \cup \mathcal{T}_2'))) \mid (\beta, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2)) \in \mathcal{W}_1,\ (\beta, ((\chi_1, \mathcal{T}_1')\delta(\chi_2, \mathcal{T}_2'))) \in \mathcal{W}_2\} \cup \\
& & \{(\beta, (\chi, (\mathcal{T}))) \mid \exists! i \in \{1,2\}.(\beta, (\chi, (\mathcal{T}))) \in \mathcal{W}_i\} \cup \\
& & \{(\beta, (\chi, (\mathcal{T}_1 \cup \mathcal{T}_2))) \mid (\beta, (\chi, \mathcal{T}_1)) \in \mathcal{W}_1,\ (\beta, (\chi, \mathcal{T}_2)) \in \mathcal{W}_2\} \\
\mathcal{W}_1 \sqcap \mathcal{W}_2 & = & \{(\beta, ((\chi_1, (\mathcal{T}_1 \cup \mathcal{T}_1'))\delta(\chi_2, (\mathcal{T}_2 \cup \mathcal{T}_2'))) \mid (\beta, ((\chi_1, \mathcal{T}_1)\delta(\chi_2, \mathcal{T}_2)) \in \mathcal{W}_1,\ (\beta, ((\chi_1, \mathcal{T}_1')\delta(\chi_2, \mathcal{T}_2'))) \in \mathcal{W}_2\} \cup \\
& & \{(\beta, (\chi, (\mathcal{T}_1 \cup \mathcal{T}_2))) \mid (\beta, (\chi, \mathcal{T}_1)) \in \mathcal{W}_1,\ (\beta, (\chi, \mathcal{T}_2)) \in \mathcal{W}_2\}
\end{array}
$$

In the type rules, $\sqcup$ and $\sqcap$ are used over sets $\mathcal{W}$ exactly as they are used over sets $\mathcal{S}$.

Notation A.1 *In the type rules we use the notation $\beta.A$, where A is a set of paths, to indicate the set $\{(\beta.\alpha, \emptyset) \mid \alpha \in A\}$.*

We now define the auxiliary function $\mathrm{Trees}_E(T)$, the predicate $T \sim_E ()$, and the auxiliary judgments $E \vdash\ T :: \mathit{NodeTest} \Rightarrow U$.

Definition A.2 (Subtrees Type Extraction) *For any E well-formed and T such that $E \vdash T$ Def, we define $\mathit{Trees}_E(T)$ as follows (well-defined by Knaster-Tarski Theorem):*

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

Definition A.3 (Empty-Forest-Type Checking) *For any well-formed environment E and type T well-formed in E, we define $T \sim_E ()$ as the minimal function (assuming $\texttt{false} < \texttt{true}$) that respects the following set of equations (well-defined by Knaster-Tarski Theorem):*

$$
\begin{array}{lcl}
() \sim_E () & \triangleq & \texttt{true} \\
l[T] \sim_E () & \triangleq & \texttt{false} \\
B \sim_E () & \triangleq & \texttt{false} \\
T, U \sim_E () & \triangleq & T \sim_E () \wedge U \sim_E () \\
T* \sim_E () & \triangleq & T \sim_E () \\
T \mid U \sim_E () & \triangleq & T \sim_E () \wedge U \sim_E () \\
X \sim_E () & \triangleq & E(X) \sim_E ()
\end{array}
$$

The correctness of this definition is proved by the following theorem.

Lemma A.4 (Empty-Forest-Type Checking)
For any well-formed environment E and type T well-formed in E:

$$T \sim_E () \ \Leftrightarrow\ [\![T]\!]_E = \{()\}$$

Table A.1. *Query Type Rules*

(TypeEmpty)
$$\frac{WF(E;\ \Gamma \vdash_\beta\ ():(();\ \emptyset;\ \emptyset))}{E;\ \Gamma \vdash_\beta\ ():(();\ \emptyset;\ \emptyset)}$$

(TypeAtomic)
$$\frac{WF(E;\ \Gamma \vdash_\beta\ b:(B;\ \emptyset;\ \emptyset))}{E;\ \Gamma \vdash_\beta\ b:(B;\ \emptyset;\ \emptyset)}$$

(TypeVarLet)
$$\frac{x:T\ \in\ \Gamma\quad WF(E;\ \Gamma \vdash_\beta\ x:(T;\ \emptyset;\ \emptyset))}{E;\ \Gamma \vdash_\beta\ x:(T;\ \emptyset;\ \emptyset)}$$

(TypeVarFor)
$$\frac{\overline{x}:T\ \in\ \Gamma\quad WF(E;\ \Gamma \vdash_\beta\ \overline{x}:(T;\ \emptyset))}{E;\ \Gamma \vdash_\beta\ \overline{x}:(T;\ \emptyset;\ \emptyset)}$$

(TypeElem)
$$\frac{E;\ \Gamma \vdash_{\beta.0}\ Q:(T;\ \mathcal{S};\ \mathcal{W})}{E;\ \Gamma \vdash_\beta\ l[Q]:(l[T];\ \mathcal{S};\ \mathcal{W})}$$

(TypeForest)
$$\frac{E;\ \Gamma \vdash_{\beta.0}\ Q_1:(T_1;\ \mathcal{S}_1;\ \mathcal{W}_1)\quad E;\ \Gamma \vdash_{\beta.1}\ Q_2:(T_2;\ \mathcal{S}_2;\ \mathcal{W}_2)}{E;\ \Gamma \vdash_\beta\ Q_1,Q_2:(T_1,T_2;\ \mathcal{S}_1 \sqcup \mathcal{S}_2;\ \mathcal{W}_1 \sqcup \mathcal{W}_2)}$$

(TypeLetWhereSplitting)
$$\frac{\begin{array}{l}E;\ \Gamma \vdash_{\beta.0}\ Q_1:(T_1;\ \mathcal{S};\ \mathcal{W})\\ \mathit{Split}_E(T_1)=\{A_1,\ldots,A_n\}\\ E;\ \Gamma,\ x:A_i \vdash_\beta\ P:(\mathcal{W}_i')\\ E;\ \Gamma,\ x:A_i \vdash_{\beta.1}\ Q_2:(U_i;\ \mathcal{S}_i;\ \mathcal{W}_i)\end{array}}{E;\ \Gamma \vdash_\beta\ \texttt{let}\ x ::= Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2:(U_1 \mid \ldots \mid U_n \mid ();\ \mathcal{S} \sqcup \sqcap_{i=1\ldots n}\mathcal{S}_i;\ \mathcal{W} \sqcup \sqcap_{i=1\ldots n}\mathcal{W}_i' \sqcup \sqcap_{i=1\ldots n}\mathcal{W}_i)}$$

(TypeForWhere)
$$\frac{\begin{array}{l}E;\ \Gamma \vdash_{\beta.0}\ Q_1:(T_1;\ \mathcal{S}_1;\ \mathcal{W}_1)\\ E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2\ \texttt{where}\ P:(T_2;\ \mathcal{S}_2;\ \mathcal{W}_2)\\ \mathcal{S} =\ \text{if } T_1 \sim_E () \text{ then } \{(\beta.0,\emptyset)\} \text{ else } \emptyset\end{array}}{E;\ \Gamma \vdash_\beta\ \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2:(T_2 \mid ();\ \mathcal{S}_1 \sqcup \mathcal{S}_2 \sqcup \mathcal{S};\ \mathcal{W}_1 \sqcup \mathcal{W}_2)}$$

(TypeLetSplitting)
$$\frac{\begin{array}{l}E;\ \Gamma \vdash_{\beta.0}\ Q_1:(T_1;\ \mathcal{S};\ \mathcal{W})\\ \mathit{Split}_E(T_1)=\{A_1,\ldots,A_n\}\\ E;\ \Gamma,\ x:A_i \vdash_{\beta.1}\ Q_2:(U_i;\ \mathcal{S}_i;\ \mathcal{W}_i)\end{array}}{E;\ \Gamma \vdash_\beta\ \texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2:(U_1 \mid \ldots \mid U_n;\ \mathcal{S} \sqcup \sqcap_{i=1\ldots n}\mathcal{S}_i;\ \mathcal{W} \sqcup \sqcap_{i=1\ldots n}\mathcal{W}_i)}$$

(TypeFor)
$$\frac{\begin{array}{l}E;\ \Gamma \vdash_{\beta.0}\ Q_1:(T_1;\ \mathcal{S}_1;\ \mathcal{W}_1)\\ E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2\ \texttt{where true}:(T_2;\ \mathcal{S}_2;\ \mathcal{W}_2)\\ \mathcal{S} =\ \text{if } T_1 \sim_E () \text{ then } \{(\beta.0,\emptyset)\} \text{ else } \emptyset\end{array}}{E;\ \Gamma \vdash_\beta\ \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2:(T_2;\ \mathcal{S}_1 \sqcup \mathcal{S}_2 \sqcup \mathcal{S};\ \mathcal{W}_1 \sqcup \mathcal{W}_2)}$$

Table A.2. *Query Type Rules: Rules for Iteration.*

(TypeInEmpty)

$$\frac{WF(E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ ()\ \rightarrow\ Q\ \texttt{where}\ P : (();\ \beta.1.CriticalLocs(Q);\ \emptyset))}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ ()\ \rightarrow\ Q\ \texttt{where}\ P : (();\ \beta.1.CriticalLocs(Q);\ \emptyset)}$$

(TypeInElSplitting)

$$\frac{\begin{array}{l} Split_E(m[T]) = \{A_1, \ldots, A_n\} \\ E;\ \Gamma,\ \overline{x} : A_i \vdash_\beta\ P : (\mathcal{W}_i') \\ E;\ \Gamma,\ \overline{x} : A_i \vdash_{\beta.1}\ Q : (U_i;\ \mathcal{S}_i;\ \mathcal{W}_i) \end{array}}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ m[T]\ \rightarrow\ Q\ \texttt{where}\ P : (U_1 \mid \ldots \mid U_n;\ \sqcap_{i=1\ldots n}\mathcal{S}_i;\ \sqcap_{i=1\ldots n}\mathcal{W}_i' \sqcup \sqcap_{i=1\ldots n}\mathcal{W}_i)}$$

(TypeInAtomic)

$$\frac{E;\ \Gamma, \overline{x} : B \vdash_{\beta.1}\ Q : (U;\ \mathcal{S};\ \mathcal{W}') \quad E;\ \Gamma,\ \overline{x} : B \vdash_\beta\ P : (\mathcal{W})}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ B\ \rightarrow\ Q\ \texttt{where}\ P : (U;\ \mathcal{S};\ \mathcal{W}' \sqcup \mathcal{W})}$$

(TypeInConc)

$$\frac{\begin{array}{l} E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q\ \texttt{where}\ P : (T';\ \mathcal{S}_1;\ \mathcal{W}_1) \\ E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ U\ \rightarrow\ Q\ \texttt{where}\ P : (U';\ \mathcal{S}_2;\ \mathcal{W}_2) \end{array}}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T, U\ \rightarrow\ Q\ \texttt{where}\ P : (T', U';\ \mathcal{S}_1 \sqcap \mathcal{S}_2;\ \mathcal{W}_1 \sqcap \mathcal{W}_2)}$$

(TypeInUnion)

$$\frac{\begin{array}{l} E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q\ \texttt{where}\ P : (T_1';\ \mathcal{S}_1;\ \mathcal{W}_1) \\ E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_2\ \rightarrow\ Q\ \texttt{where}\ P : (T_2';\ \mathcal{S}_2;\ \mathcal{W}_2) \end{array}}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1 \mid T_2\ \rightarrow\ Q\ \texttt{where}\ P : (T_1' \mid T_2';\ \mathcal{S}_1 \sqcap \mathcal{S}_2;\ \mathcal{W}_1 \sqcap \mathcal{W}_2)}$$

(TypeInVar)

$$\frac{\begin{array}{l} E(X) = T \\ E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q\ \texttt{where}\ P : (U;\ \mathcal{S};\ \mathcal{W}) \end{array}}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ X\ \rightarrow\ Q\ \texttt{where}\ P : (U;\ \mathcal{S};\ \mathcal{W})}$$

(TypeInStar)

$$\frac{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q\ \texttt{where}\ P : (U;\ \mathcal{S};\ \mathcal{W})}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T*\ \rightarrow\ Q\ \texttt{where}\ P : (U*;\ \mathcal{S};\ \mathcal{W})}$$

Table A.3. *Child and Dos Type Rules*

(TypeChild)

$$\frac{\begin{array}{l} WF(E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{child} :: NodeTest : (U;\ \mathcal{S};\ \emptyset)) \\ \overline{x} : T \in \Gamma\ \wedge\ (T = m[T'] \vee T = B) \\ T' = \text{ if } T = m[T'] \text{ then } T'' \text{else } () \\ E \vdash\ T' :: NodeTest \Rightarrow U \\ \mathcal{S} = \text{ if } U \sim_E () \text{ then } \{(\beta, \{T\})\} \text{ else } \emptyset \end{array}}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{child} :: NodeTest : (U;\ \mathcal{S};\ \emptyset)}$$

(TypeDos)

$$\frac{\begin{array}{l} WF(E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{dos} :: NodeTest : (U;\ \mathcal{S};\ \emptyset)) \\ \overline{x} : T \in \Gamma\ \wedge\ (T = m[T'']\ \vee\ B) \\ \{U_1, \ldots, U_n\} = \text{Trees}_E(T) \\ U' = (U_1 \mid \ldots \mid U_n)* \\ E \vdash\ U' :: NodeTest \Rightarrow U \\ \mathcal{S} = \text{ if } U \sim_E () \text{ then } \{(\beta, \{T\})\} \text{ else } \emptyset \end{array}}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{dos} :: NodeTest : (U;\ \mathcal{S};\ \emptyset)}$$

Table A.4. *Where Type Rules*

(True)
$$\frac{}{E;\ \Gamma \vdash_\beta\ \texttt{true} : (\emptyset)}$$

(CompOk)
$$\frac{E \vdash B \leq \Gamma(\chi_1) \quad E \vdash B \leq \Gamma(\chi_2)}{E;\ \Gamma \vdash_\beta\ \chi_1\ \delta\ \chi_2 : (\emptyset)}$$

(CompWrong)
$$\frac{\textit{if not}(E \vdash B \leq \Gamma(\chi_i) \quad i = 1,2)\ \textit{then}\ \mathcal{W} = \{(\beta, ((\chi_1, \{\Gamma(\chi_1)\})\delta(\chi_2, \{\Gamma(\chi_2)\}))\}\ \textit{else}\ \mathcal{W} = \emptyset}{E;\ \Gamma \vdash_\beta\ \chi_1\ \delta\ \chi_2 : (\mathcal{W})}$$

(Or)
$$\frac{E;\ \Gamma \vdash_\beta\ P_1 : (\mathcal{W}_1) \quad E;\ \Gamma \vdash_\beta\ P_2 : (\mathcal{W}_2)}{E;\ \Gamma \vdash_\beta\ P_1\ \texttt{or}\ P_2 : (\mathcal{W}_1 \sqcup \mathcal{W}_2)}$$

(Not/Brak)
$$\frac{E;\ \Gamma \vdash_\beta\ P : (\mathcal{W})}{E;\ \Gamma \vdash_\beta\ \texttt{not}\ P : (\mathcal{W}) \quad E;\ \Gamma \vdash_\beta\ (P) : (\mathcal{W})}$$

(EmpWrong)
$$\frac{\textit{if}\ (E \vdash B = \Gamma(\chi))\ \textit{then}\ \mathcal{W} = \{(\beta, (\chi, \Gamma(\chi)))\}\ \textit{else}\ \mathcal{W} = \emptyset}{E;\ \Gamma \vdash_\beta\ \texttt{empty}(\chi) : (\mathcal{W})}$$

(EmpOk)
$$\frac{\textit{not}(E \vdash B = \Gamma(\chi))}{E;\ \Gamma \vdash_\beta\ \texttt{empty}(\chi) : (\emptyset)}$$

Rules to prove judgment $E \vdash\ T :: \mathit{NodeTest} \Rightarrow U$ are the following:

$$\frac{}{E \vdash\ T :: \texttt{node}() \Rightarrow T}\ (\textsc{MatchAnyFilt})$$

$$\frac{}{E \vdash\ l[T] :: l \Rightarrow l[T]}\ (\textsc{MatchLabFilt})$$

$$\frac{T = B\ \vee\ T = m[T']}{E \vdash\ T :: l \Rightarrow ()}\ (\textsc{NoMatchLabFilt})$$

$$\frac{}{E \vdash\ B :: \texttt{text}() \Rightarrow B}\ (\textsc{MatchTextFilt})$$

$$\frac{T = ()\ \vee\ T = m[T']}{E \vdash\ T :: \texttt{text}() \Rightarrow ()}\ (\textsc{NoMatchTextFilt})$$

$$\frac{E \vdash\ T :: \mathit{NodeTest} \Rightarrow T' \qquad E \vdash\ U :: \mathit{NodeTest} \Rightarrow U'}{E \vdash\ T, U :: \mathit{NodeTest} \Rightarrow T', U'}\ (\textsc{ForestFilt})$$

$$\frac{E \vdash\ T :: \mathit{NodeTest} \Rightarrow U}{E \vdash\ T* :: \mathit{NodeTest} \Rightarrow U*}\ (\textsc{StarFilt})$$

$$\frac{E \vdash\ T :: \mathit{NodeTest} \Rightarrow T' \qquad E \vdash\ U :: \mathit{NodeTest} \Rightarrow U'}{E \vdash\ T \mid U :: \mathit{NodeTest} \Rightarrow T' \mid U'}\ (\textsc{UnionFilt})$$

$$\frac{E \vdash\ E(X) :: \mathit{NodeTest} \Rightarrow U}{E \vdash\ X :: \mathit{NodeTest} \Rightarrow U}\ (\textsc{VarFilt})$$

Lemma A.5 (Termination of Type Filtering) *For any label l, type environment E well-formed and types T and U, the backward application of the type rules to $E \vdash\ T :: \mathit{NodeTest} \Rightarrow U$ terminates.*

Lemma A.6 (Type Filtering Checking) *For any well-formed type environment E and type T well-formed in E:*

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

Lemma A.7 (Soundness of DOS) *For any E well-formed and T well-defined in E:*

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

A.1 Properties of the Type System

We provisionally assumed that $Split_E(T) = \{T\}$, which results in a completely standard LET-RETURN type rule. This is sufficient to obtain the canonical 'soundness' property (Theorem A.9): types are upper bounds for the set of all possible results.

Definition A.8 ($\mathcal{R}(E,\Gamma)$) *For any well-formed type environment E and Γ well-formed in E, we define the set of valid substitutions as*

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

where χ is either a for-variable or a let-variable.

Theorem A.9 (Upper Bound) *For any well-formed environment E, Γ well-formed in E, and well-formed Q:*

$$E;\ \Gamma \vdash_\beta\ Q : (U;\ \_;\ \_)\ \wedge\ \rho \in \mathcal{R}(E,\Gamma)\ \Rightarrow\ [\![Q]\!]_\rho \in [\![U]\!]_E$$

The next property we have is some form of 'we will never bother you with a false alarm', an important property for our purposes, as we use the proposed approach as a maintenance tool for p2p database systems; hence, the site administrator would be warned about issues in the mappings only if there are real problems (not just related to a figment of the type rules).

Theorem A.10 (Soundness of Error-Checking) *For each query Q, and Γ well-formed in E:*

$E;\ \Gamma \vdash_\beta\ Q : (\_;\ \mathcal{S};\mathcal{W})\ \Rightarrow$

- $(\beta.\alpha, \mathcal{T}) \in \mathcal{S}\ \Rightarrow$ *Q has an FE error at α w.r.t. $\mathcal{R}(E,\Gamma)$*
- $(\beta.\alpha, ((\chi_1,\mathcal{T}_1)\delta(\chi_2,\mathcal{T}_2)) \in \mathcal{W}\ \Rightarrow$ *Q has a where error at the α-condition $\chi_1\ \delta\ \chi_2$ w.r.t. $\mathcal{R}(E,\Gamma)$*
- $(\beta.\alpha, (\chi,\mathcal{T})) \in \mathcal{W}\ \Rightarrow$ *Q has a where error at the α-condition* $\texttt{empty}(\chi)$ *w.r.t. $\mathcal{R}(E,\Gamma)$*

A.2 Type-Splitting

We provisionally assumed that $Split_E(T) = \{T\}$. This simple definition is enough to obtain soundness of type checking and error-checking. These are the canonical properties that are proved for any type system, but they are not very informative: any system that associates the universal type to any expression, and never finds any existential error, enjoys them as well. For the language μXQ, we can actually aim for a much stronger property: a type system that is *complete*, in a sense to be made precise later, and that is able to catch *every* error.

Our provisional type system is not up to this aim. It 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 $X = data[mobile[]* \mid phone[]*]$, and the query

$$\texttt{x/mobile}, \texttt{x/phone}.$$

When `x` has type X, this query yields either a sequence of elements $mobile[]$ or a sequence of elements $phone[]$. Instead, as in XQuery, our type system infers a type ($mobile[]*$, $phone[]*$), which also contains sequences with both $mobile[]$ and $phone[]$ elements.

Our provisional type system does not guarantee completeness of error-checking either. For example, consider the type $Y = c[a[] \mid b[]]$ and the query:

$$Q_8 = \texttt{for } \overline{\texttt{x}} \texttt{ in y/a return y/b}$$

where `y` is of type Y (this code returns a sequence of `y/b` iff `y` has a child `a`, and returns () otherwise). The query is NE-incorrect, as there is no substitution that makes the subquery `y/b` yield a not-empty result: if `y` is of type $c[a[]]$ then `y/b` cannot return any tree, and if `y` is of type $c[b[]]$ then `y/a` is empty, hence `y/b` will not be evaluated at all. Nevertheless, our provisional type system validates the query as correct. This is because the two uses of `y` are deemed acceptable by exploiting two separate, and incompatible, branches of the union type of `y`. More specifically, during type-checking, every free variable is substituted with the *whole* type that the variable is assigned to by the environment. For the same reason, the type inferred for this query, $Z = b[] \mid ()$, is different from the optimal type "()"(similar phenomena happen in all related type systems we are aware of, including the XQuery type system).

Similarly, if we assume $Split_E(T) = \{T\}$, then the type system considers as correct the following query

```
let x := y/a/text()
let z := y/b/text()
where x = z
return x
```

where y is of type $Y = c[a[] \mid b[]]$.

As before, the type system considers this query as WHERE-correct, as both x and y have type $B \mid ()$. However, the query is not WHERE-correct, as, in each possible evaluation of the query, one of the two compared variables is assigned to ().

We solve these problems by using in the rules a finer *Split*() function, which produces more precise types and errors. For example, type $c[a[] \mid b[]]$ is split into $\{c[a[]], c[b[]]\}$, hence the query Q_8 presented above is analyzed once with $y : c[a[]]$ and once with $y : c[b[]]$; then, the sub-query $(Q_8)_{|1}$ is (correctly) flagged as wrong, since the location 1 is in the error set of both runs of the analysis. The definition of $Split_E(T)$ is non-trivial because of recursive type variables. Consider the type $Y = a[Y] \mid b[Y] \mid ()$ and a type assumption $y : Y$. Every time we unfold Y, new instances of $\mid$ appear, which have to be "pulled out" by $Split_E(T)$, and which generate new cases to analyze. We would like to unfold Y just once, and to analyze the query just three times, trying $y : a[Y]$, $y : b[Y]$ and $y : ()$. However, consider the following query, where $(/a)^n$ stands for n consecutive occurrences of $/a$:

$$Q^n = \texttt{for } \overline{\texttt{x}} \texttt{ in y}(/\texttt{a})^{\texttt{n}}/\texttt{a return y}(/\texttt{a})^{\texttt{n}}/\texttt{b}$$

To catch the error, Y must be unfolded $n + 1$ times. However, a solution like this is hard to generalize, because of queries containing the self-or-descendants axis, like:

```
for z in y//node()
for x in z/a
return z/b
```

still with y of type $Y = a[Y] \mid b[Y] \mid ()$.

Our solution, which is acceptable in practice, is based on a mild restriction on the use of recursion. We restrict to environments E, namely $*$-guarded environments, for which recursion is guarded by a $*$ type constructor, hence ruling out the Y type above (see [4] for details). Under this restriction, error-completeness is obtained by unfolding recursion until $*$ is met, and "pulling out" only the union type constructors that are found outside the $*$ (Definition A.11).

Definition A.11 ($Split_E(T)$) *If E is $*$-guarded, then:*

$$\begin{array}{lcl}
Split_E(()) & \triangleq & \{()\} \\
Split_E(B) & \triangleq & \{B\} \\
Split_E(U*) & \triangleq & \{U*\} \\
Split_E(X) & \triangleq & Split_E(E(X)) \\
Split_E(T \mid U) & \triangleq & Split_E(T) \cup Split_E(U) \\
Split_E(l[T]) & \triangleq & \{l[A] \mid A \in Split_E(T)\} \\
Split_E(T, U) & \triangleq & \{(A, B) \mid A \in Split_E(T) \wedge B \in Split_E(U)\}
\end{array}$$

If E is $*$-guarded, $Split_E(T)$ can be computed by a standard top-down recursive implementation of the definition above: $*$-guardedness of E implies that the $*$ case will break any potential infinite loop due to the recursive definition of a type variable. As stated in the body of the paper, for E $*$-guarded, this finer definition of $Split_E(T)$ ensures completeness for ()-correctness, and we are quite confident that completeness extends to WHERE-correctness as well. As an example, consider the previous counter-example:

```
let x := y/a/text()
let z := y/b/text()
where x = z
return x
```

The type-splitting system correctly deems this query as a not WHERE-correct query.