Types for Correctness of Queries over Semistructured Data

Dario Colazzo Giorgio Ghelli Paolo Manghi Carlo Sartiani

Dipartimento di Informatica - Università di Pisa, Corso Italia 40, Pisa, Italy
e-mail: {colazzo, ghelli, manghi, sartiani}@di.unipi.it

Abstract

A type system for a query language should serve both purposes of verifying whether a query is coherent with what is known about the structure of the database (query correctness) and of giving information about the type of the query result (result analysis). Current proposals for typed query languages for semistructured data are usually focused on result analysis, but perform very few controls, or none at all, of query correctness.

This work presents a type system for a core of XQuery that supports both query correctness and result analysis, and discusses some of the design issues and alternatives.

1 Introduction

In conventional query languages, a type system analyses *query correctness* and the result type. Query correctness is generally defined as a relation of *compatibility* between the type of the query input and the *query type*. The query type represents the type of the data targeted by the query, and is either inferred from the query structure, or directly provided by the programmer. Correctness ensures, at the very minimum, that the query has some hope to find a match in data that respects the input type.

Result analysis is the process of checking whether a query effectively returns data of an expected output type. This check is performed by inferring the output type of a query and by matching it against the expected type.

Both query correctness and result analysis are useful tools for the development of complex database applications, where database queries and high-level programming language applications are usually combined forming a complex web of input and output type dependencies.

In the context of semi-structured data (SSD) and XML only a few query and manipulation languages exploit static type information, given the possibly irregular and unstable nature of the data. Current proposals mainly focus on result analysis [3, 6, 5], but perform very few controls, or none at all, of query correctness. Actually, there is no clear agreement, and neither much discussion, about what query correctness means in this context.

This is due to the fact that SSD, especially XML documents, are usually endowed with rather irregular type descriptions, comprising union types, recursive types, and wildcards; coherently, the corresponding languages include operators such as alternative paths, wildcard matching, collection of descendants.

One possible notion of query correctness, adopted by the *XDuce* language [4], is the full correspondence between the alternative paths in a query and all the possible cases of the union-type that describes the input. This approach seems, however, too restrictive for many SSD specific programming tasks.

At the other extreme, one may flag as non-correct only those queries that are statically deemed to always return an empty collection. This approach has been suggested by the authors of *XQuery* [1]. However, unless the system is able to flag the specific parts of the query where the problem arises, this policy becomes quite loose, and not informative enough for programmers.

An intermediate notion of correctness is to deem as wrong all and only those paths in the query that have no hope to match the input data.

In summary, each of these approaches is reasonable in some specific application, hence none of them can be regarded as *the* general purpose solution.

Our contribution This work describes a type system for μXQuery, an abstract core of XQuery. μXQuery's type system provides a formal framework where different notions of query correctness can be formalized and compared. Specifically, it supports a notion of conformance of data to a type, of result analysis, and is based on a three-levels definition of query correctness, according to which a query can be classified as:

- *incorrect*, if the structural requirements of the query will not find a match against any instance of the database schema;
- *weakly correct*, if the structural requirements of the query will find a match in at least one instance of the database schema.
- *strongly correct*, if the structural requirements will find a match against all possible instances of the database schema.

We believe this characterisation to be particularly suitable in the context of SSDBs, as it is based on a clear semantic characterization but is flexible enough to be compatible with the needs of different applications.

2 Query correctness in XML query languages

In the absence of input type description, query correctness cannot be checked. As a consequence, programmers may interpret an empty result as being due to a structural requirement failure (*from* clause) or a selection failure (no data satisfied the *where* clause).

```
<!DOCTYPE people[
<!ELEMENT people person+>
<!ELEMENT person (name,phone)>
<!ELEMENT name (frsname, sndname | firstname, secondname)>
<!ELEMENT frname  PCDATA>
<!ELEMENT snname  PCDATA>
<!ELEMENT firstname  PCDATA>
<!ELEMENT secondname  PCDATA>
<!ELEMENT phone  PCDATA>
]>
```

Figure 1: A sample DTD.

When schemas are available, instead, some static controls can in principle be performed. However, the irregular nature of SSD types and query languages makes this aim very elusive. In fact, only the language XDuce defines a standard notion of query correctness, but XDuce is quite far from the standard structure of query languages, being a stricter relative to functional languages as ML.

Other approaches, such as XQuery's and Suciu's approach [6], are not concerned with the automatic identification of incorrect queries and concentrate on result analysis. Given a query and a schema for the database at hand, the problem is that of checking whether every output of the query conforms to a given expected output type.

Such solutions target different kinds of application scenarios and therefore differ for a number of design choices. However, if we focus on query correctness, XDuce's approach turns out to be quite restrictive for a query language, while the other approaches are instead poorly informative for the programmer.

XDuce XDuce is a typed, functional, Turing complete programming language. It is based on an ML-like pattern language that implements a *one-match* semantics, i.e. every pattern, instead of collecting every matched piece of data (as in standard query languages), only binds the first match. XDuce is nearer to a programming language than to a query language, but we consider it here since it is the only example of typed language for XML that explicitly provides a notion of type correctness.

For example, consider the following XDuce's function, which returns the list of phone numbers of all people in the document `d`, which conforms to the schema in Figure 1:

```
fun selNums: person* --> (sndname,phone)* =
 person[name[frsname[String],sndname[n:String]],
        phone[p:String]], rest:person*
        --> sndname[n], phone[p], selNums(rest) |
 person[name[firstname[String],secondname[n:String]],
        phone[p:String]], rest:person*
        --> sndname[n], phone[p], selNums(rest) |
 ()     --> ()
```

(XD1)

`selNums` can be applied to the element `people` of `d`. This function is type correct, because XDuce supports a notion of query correctness according to which functions are correct if and only if they specify a matching pattern (a function case) for all possible patterns described by the input type: exhaustive patterns in function definitions are required to ensure the soundness of the type system of XDuce, as stated in [4]. Indeed, the function,

```
fun selNums: person* --> (sndname,phone)* =
 persons[name[frsname[String],sndname[n:String]],
         phone[p:String]], rest:person*
         --> sndname[n], phone[p], selNums(rest) |
 persons[name[firstname[String],secondname[n:String]],
         phone[p:String]], rest:person*
         --> sndname[n], phone[p], selNums(rest) |
 ()      --> ()
```

(XD2)

is statically judged incorrect and never executed, because the field `persons` is not defined in the schema. This notion of correctness, however, is too restrictive for XML querying purposes. For instance, the function

```
fun selNums: person* --> (sndname,phone)* =
 person[name[frsname[String],sndname[n:String]],
        phone[p:String]], rest:person*
        --> sndname[n], phone[p], selNums(rest) |
 ()     --> ()
```

(XD3)

is considered incorrect and never executed, although one would expect the query to be run, as instances of the database exist that are matched by the body of the function.

XQuery XQuery's type system infers the output type of a query by matching the structural requirements of the query (query type) with the type of the query input [3]. In doing this, the type system does not identify and discard incorrect queries, but simply assigns an empty type to those subparts of the query that cannot find a match in any instance of the data. Coherently, a query over an instance of a union type is assigned an empty type only when none of the members of the union type is relevant to the query.[1] Given the expected type of the query's output data and the inferred output type of the query, the system can statically detect if the query's output value has the expected output type.

The function (XD1) can be encoded in the following XQuery's query,

[1] This policy is looser than XDuce's, where a query is accepted only if it searches for all the members of a union type, and more suitable for navigating arbitrarily irregular SSDBs.

```
for $p in d/person,
    $n in op:union($p/name/sndname,
                   $p/name/secondname),
    $ph in $p/phone
return <sndname> data($n) </sndname>,
       $ph
```
(XQ1)

for which the type system statically infers the following output type,

```
(element sndname {xsd:string},
 element phone {xsd:string})*
```

Function (XD3) corresponds to the query,

```
for $p in d/person,
    $n in $p/name/sndname,
    $ph in $p/phone
return <sndname> data($n) </sndname>,
       $ph
```
(XQ3)

The type system infers the same output type of (XQ1). Result analysis states that both (XQ1) and (XQ3) are correct as their output type matches (is a subtype of) the expected type of the output. The function (XD2) becomes instead,

```
for $p in d/persons,
    $n in op:union($p/name/sndname,
                   $p/name/secondname),
    $ph in $p/phone
return <sndname> data($n) </sndname>,
       $ph
```
(XQ2)

The type inferred for this query is the empty type `()`. The system pinpoints the error, as the programmer was expecting a different type.

Essentially, XQuery provides programmers with powerful result analysis tools, which are, in some situations, also useful for detecting errors before execution. In particular, the authors observe that, since queries are assigned an empty type if they cannot find a match in any instance of the input type, when the inferred type is empty the system may automatically report the query as incorrect. This is generally true, unless the programmer were expecting an empty type as output. This notion of incorrectness, however, is rather incomplete, as many incorrect queries do not necessarily return an empty type. Consider for example the following query:

```
for $p in d/person,
    $n in op:union($p/name/sndname,
                   $p/name/secondname),
    $ph in $p/phone
return <sndname> data($n) </sndname>,
       $ph,
       $p/age
```
(XQ4)

Although the schema of `d` contains no `age` field, the type system infers exactly the same output type inferred for the query (XQ1). The same happens with the following query, although the schema of `d` contains no `seconname` field.

```
for $p in d/person,
    $n in op:union($p/name/sndname,
                   $p/name/seconname),
    $ph in $p/phone
return <sndname> data($n) </sndname>,
       $ph
```
(XQ5)

Suciu's proposal Dan Suciu et al. focus on the development of a formal framework for the definition of result analysis tools [6]. They view queries as *transformation programs*, i.e. applications transforming an original data source into an XML database that conforms to a given type.

They explore a backward type inference mechanism, which takes as inputs the query, the query input type, the expected output type, and checks that every database that is the result of the query applied to an instance of the input type, conforms to the output type. Again, the methodology fully addresses the issues of result analysis, but totally disregards a notion of query correctness.

3 μXQuery

μXQuery is an abstract version of the FLWR core of XQuery. The main difference between μXQuery and XQuery is the lack of support for function definitions, and for $if-then-else$ and *typeswitch* expression. Moreover, μXQuery features only *copy-semantics* return clauses, hence discarding *reference-semantics* element construction.

The novelty of μXQuery's type system is that it has been specifically designed for both result analysis and query correctness checking. The type system infers the output type of a query, as in XQuery, but makes a distinction between correct and incorrect queries, as in XDuce. In particular, it supports a three-levels definition of correct queries, distinguishing between weakly correct and strongly correct queries. We shall briefly discuss the advantages of this approach.

3.1 Grammar

Queries are defined by the following grammar:

$$Q \quad ::= \quad () \mid v_B \mid \langle l \rangle\, Q\, \langle /l \rangle \mid Q, Q \mid x \mid Q\, p \\ \mid \texttt{let}\ x := Q\ \texttt{return}\ Q \mid \texttt{for}\ x\ \texttt{in}\ Q\ \texttt{return}\ Q$$

Data are represented as ordered forests (f) of trees (t), as defined by the following grammar:

$$f \quad ::= \quad () \mid t \mid f, f \qquad t \quad ::= \quad v_B \mid \langle l \rangle\, f\, \langle /l \rangle$$

where v_B is a leaf value of type B, ',' is associative, and $(\,), f = f, (\,) = f$.

Paths are defined by the following grammar:

$$\begin{array}{lcl} p & ::= & nil \mid /ls \mid //ls \mid p\,p \mid p + p' \\ ls & ::= & l \mid * \end{array}$$

3.2 Query semantics

A query Q is evaluated according to an environment ρ which associates a forest with each free variable occurring in Q, and the result is denoted by $[\![Q]\!]_\rho$. Informally, $[\![Q]\!]_\rho$ yields the pair $\langle f, \mathcal{S}\rangle$, where f is the forest returned by Q with respect to the substitution ρ, and $\mathcal{S}$ is a status variable that captures a notion of *correct* execution (formal details can be found in [2]). $\mathcal{S}$ ranges over the set {C, F}, respectively representing the *correct* or *faulty* status of execution. Specifically, $\langle f, \mathtt{C}\rangle$ states that f is *correctly returned* by a query, while $\langle f, \mathtt{F}\rangle$ states that f is *faultily returned* by a query.

In detail, a query Q *correctly returns* a forest f in an environment ρ ($[\![Q]\!]_\rho = \langle f, \mathtt{C}\rangle$), if, for all path selections $Q'\,p$ in Q, the path p finds a match with the forest returned by Q'. In particular, for path selections of the form $Q'\,(p_1{+}p_2)p$ it is only required that either p_1p or p_2p finds a match in the forest returned by Q'. A query Q *faultily returns* a forest f in an environment ρ ($[\![Q]\!]_\rho = \langle f, \mathtt{F}\rangle$), if there exists a path selection $Q'\,p$ in Q for which either Q' faultily returns a forest f' or p cannot find a match in f'.

Because of union types, two well-typed input values may exist such that the same query may correctly return a result on one and faultily return a result on the other one. For this reason, we adopt a three-levels classification of the semantic correctness of a query with respect to an input type: *strongly correct* if it correctly returns a forest for any well-typed input, *weakly correct* if it correctly returns a forest for some well-typed input, *incorrect* if it faultily returns a forest for any well-typed input (Definition 3.1).

Consider for example the databases `d1` and `d2` in Figure 2, which conform to the schema in Figure 1. The query (XQ4) on `d1` *faultily* returns the forest:

```
<sndname> Sartiani </sndname> <phone> 123456 </phone>
```

as the path selection (`$p\age`) does not match the data; for the same reason, (XQ4) execution would be faulty over `d2` or any `d` that conforms to the same schema. The same is true for the query (XQ2), which faultily returns the empty forest because the `/persons` path will never match the data. These queries are *incorrect*.

On the other side, a query `d/person/name/secondname` would *faultily* return an empty forest when applied to `d1`, since it finds no match, but would *correctly* return a result when applied to `d2`. Hence this query is *weakly correct*.

For path selections $Q\,(p_1 + p_2)p$, our notion of correct execution is rather permissive, in the sense that, as we already said, matching is required for at least one alternative. The query (XQ1), for example, is correctly executed over `d1`, and returns the same forest f above with $\mathcal{S} = \mathtt{C}$, since one of its alternative paths finds a match in d. (XQ1) is actually *strongly correct*, since, for any well-typed content of the database, at least one of its alternative paths finds a match. The query (XQ5), when applied to `d1`, returns the same forest as (XQ1), thanks to the disjunct `/name/sndname`, hence this execution is correct, despite the presence of the wrong path `/name/seconname`. However, (XQ5) is actually *weakly correct* since, over `d2`, its execution is faulty.

Hence, our semantics defines a three-level notion of the correctness of a query with respect to an input type. Now, our aim is to define a type system that is able to infer, for each query, a reasonable approximation of its semantic correctness with respect to a given input type.

```
d1 = <people><person>
          <name>
            <frsname> Carlo </frsname>
            <sndname> Sartiani </sndname>
          </name>
          <phone> 123456 </phone>
     </person></people>

d2 = <people><person>
          <name>
            <firstname> Dario </firstname>
            <secondname> Colazzo </secondname>
          </name>
          <phone> 654321 </phone>
     </person></people>
```

Figure 2: Two db's conforming to the DTD in Figure 1

3.3 Type system

In this Section we introduce μXQuery's type system. We first introduce the syntax of types, then give an informal characterisation of the semantics of types, and give a characterisation of type correctness in terms of the semantics of queries. Formal definitions, as well as type rules, can be found in [2].

3.3.1 Type language

The type language we consider is essentially XDuce's type language, and is defined by the following grammar.

$$T \quad ::= \quad () \mid B \mid T, U \mid T + U \mid l[T] \mid X$$

where B represents atomic types. The empty type $(\,)$ only contains the empty forest $(\,)$. The type constructor $l[T]$ represents the set of trees rooted as l and containing a forest of type T. Concatenation T, U represents the set of forests f, f', where f and f' are forests in T and U respectively. The untagged union type constructor $T + U$ represents the set of forests f which belong to either T or U.

Type variables are defined by an environment E, which consists of a set of potentially mutual recursive type definitions of the following form:

$$
\begin{array}{lll}
E & ::= & (\,) \qquad\qquad \textit{empty environment} \\
 & & X = T,\ E \quad \textit{type variable definition} \\
 & & x : T,\ E \quad \textit{query variable declaration}
\end{array}
$$

Note that environments also contain query variable type declarations $x : T$. These are used in the typing rules given in [2]. Moreover, observe that regular expressions types, such as repetition and optional types, can be defined by combining recursive and union types as follows:

$$T* \ \equiv\ X \text{ with } X = (\,) + (T, X) \qquad T? \ \equiv\ (\,) + T$$

For instance, the DTD given in Figure 1 corresponds to the following μXQuery's type environment,

```
PEOPLE = people[PERSON +]
PERSON =  person[NAME,PHONE]
NAME = name[(FRSNAME,SNDNAME) + (FIRSTNAME,SECONDNAME)]
FRSNAME = frsname[String]
SNDNAME = sndname[String]
FIRSTNAME = firstname[String]
SECONDNAME = secondname[String]
PHONE = phone[String]
```

While query correctness in XDuce and XQuery is based on subtyping, in μXQuery it is based on a relation of coherence between query paths and query input types.[2]

3.3.2 Semantics of types

We interpret a type as the set of all forests that have that type. In the style of [4], we define the semantics of types by means of a set of deduction rules over judgements of the form $E \vdash f : T$, which state that f conforms to T with respect to E. Informally, we write

$$[\![T]\!]_E = \{f \ \mid\ E \vdash f : T\}.$$

3.3.3 Query correctness

To define query correctness in μXQuery, we denote as $[\![Q]\!]_E$ the set of all possible results, i.e. pairs $\langle f, \mathcal{S}\rangle$, returned by Q, for each assignment to Q's variables that respects the type definitions in E.

Definition 3.1 (Correctness) *Given a query Q and an environment E of type definitions for free variables in Q, we say that Q is*

$$
\begin{array}{lll}
\textbf{strongly correct} & \textit{if} & \forall \langle f, \mathcal{S}\rangle \in [\![Q]\!]_E \,.\ \mathcal{S} = \mathtt{C} \\
\textbf{weakly correct} & \textit{if} & \exists \langle f, \mathcal{S}\rangle \in [\![Q]\!]_E \,.\ \mathcal{S} = \mathtt{C} \\
\textbf{incorrect} & \textit{if} & \forall \langle f, \mathcal{S}\rangle \in [\![Q]\!]_E \,.\ \mathcal{S} = \mathtt{F}
\end{array}
$$

In [2] we have defined a set of algorithmic rules which reflect this characterisation, by returning the *inferred correctness* of a query (the relationship between the correctness as inferred by the type rules and the actual correctness is discussed below). Observe that query correctness is strictly related with type inference, as in the presence of nested queries, correctness of outer queries depends on the type inferred for inner queries. As a consequence, the rules also return the query type thereby enabling result analysis techniques.

Table 1 shows the *inferred correctness* returned by the rules for path selections $Q\ p$, given the type inferred for Q (the symbol _ denotes any label) and the path p.

	Path p	*Type of Q*	*Strong*	*Weak*
1	$/l\ +\ /m$	$\_[l[T] + m[U]]$	*Yes*	*Yes*
2	$/l\ +\ /m + /n$	$\_[l[T] + m[U]]$	*Yes*	*Yes*
3	$/l$	$\_[l[T] + m[U]]$	*No*	*Yes*
4	$/l\ +\ /n$	$\_[l[T] + m[U]]$	*No*	*Yes*
5	$/n$	$\_[l[T] + m[U]]$	*No*	*No*
6	$/l\ +\ /o$	$\_[l[T] + m[U]] + n[V]]$	*No*	*Yes*

Table 1: Inferred correctness.

As illustrated by the rows 1 and 2, $Q\ p$ is strongly type correct if p finds a match in *each* instance of the type of Q. Accordingly, when checking correctness of a path p with an input union type, the rules state that $Q\ p$ is strongly correct only if all members of the union type are matched by p (*data covering*). However, we do not require here the *path covering* property, i.e. that every disjunctive member of p is matched (or may be matched) by a piece of data, hence row 2 is strongly correct as well. Data covering and path covering are, in a sense, independent issues, both of them relevant, though we focus here on the first one only; we will come back to this in Section 4.

When not all members of the type, but at least one, are matched by p, the query is weakly correct, as exemplified in rows 3, 4, and 6. In this case, programmers may exploit this information, deciding either to make their queries strongly correct, by adding the missing alternatives in the path, or run them anyway when they are not interested in querying the non-matching instances. Finally, row 5 exemplifies an incorrect query, whose path will never match any data.

The rules, given a query Q and an environment E, infer a pair $(T; \mathcal{A})$, where T is the output type of Q and $\mathcal{A}$ is a variable that ranges over the set $\{\mathtt{s}, \mathtt{w}, \mathtt{i}\}$. The correct definition of the rules is far from trivial. Consider the following query over a database of type $root[\ a[int] + b[int]\ ]$, bound to the variable `$y`,

```
{ $y/a, $y/b }
```

The query is semantically incorrect, as `$y` either matches `/a` or `/b`, but cannot match both of them. However, a standard inductive type rule such as,

$$E \vdash Q : \mathtt{w},\ E \vdash Q' : \mathtt{w} \ \Rightarrow\ E \vdash Q, Q' : \mathtt{w}$$

does not suffice for the example above, as both `$y/a` and `$y/b` are weakly correct with respect to the type of `$y`. Similarly, but more subtly, the following query

```
{ $y/a/a, $y/a/b }
```

[2] As a consequence of this the system may infer context-free types for some queries, even when they feature regular input types.

where `$y` is of type $root[X]$ with $X = a[X] + b[X] + int$, is semantically incorrect. Indeed, the product query `$y/a/a, $y/a/b` is incorrect as each level of a unary tree either contains a label a or a label b.

In [2] we give a solution to these problems by means of complex algorithmic type rules, which keep track of which members of union types are matched by the paths in the query, and return a correctness status which depends on this information.

The following proposition claims soundness of the type rules with respect to the characterisation of query correctness, although the rules are not complete (a strongly correct query may be flagged as `w`).

Proposition 3.2 (Soundness) *For each query Q and environment E, if the judgement $E \vdash Q : (T;\ \mathcal{A})$ holds, then,*

if $\mathcal{A} = \mathtt{s}$ *then* *Q is strongly correct;*
if $\mathcal{A} = \mathtt{w}$ *then* *Q is weakly or strongly correct;*
if $\mathcal{A} = \mathtt{i}$ *then* *Q is incorrect;*

Observe that the type rules always infer an output type. By doing so, in the style of XQuery, the type system also provides programmers with result analysis tools. For example, for queries (XQ1) to (XQ4), our type rules infer the same output types as XQuery's. However, we are also able to identify (XQ1) as strongly correct, (XQ3) and (XQ5) as weakly correct, and (XQ2) and (XQ4) as incorrect.

4 Path covering

To simplify the discussion, imagine, for a moment, that every query is just a sum of paths, and every input type is just a union type. Then, the system we described up to now is geared towards the prevention of problems that arise (informally) because one branch of the input-data union-type is not covered by any path (lack of strong correctness), or even *no* branch of the union type is covered by any path (lack of weak correctness).

This is already complex enough, but only captures those errors that show up as 'too few paths in the query'. Errors may also show up as 'too many paths', as in rows 2, 4, and 6, in Table 1, or in query (XQ5), where we have paths that are not covered by any branch of the union type.

Typical programming errors, like path misspelling, tend to generate both path and data coverage problems, as in (XQ5) or in row 4, hence suggesting that path coverage *may* be ignored. On the other side, consider row 6 in Table 1. Here the programmer misspelled an $/m$ into an $/o$, was expecting a 'weak correctness' result, and gets it from the type-checker; hence, the misspelling problem does not show up in the type.

Moreover, the errors generated by a data-covering based analysis can only be reported in terms of type-branches that have not been considered by a subpart of the query, while a path-covering based analysis would pinpoint a wrong path. Going back to line 4 if the table, the data-covering error is 'branch $m[U]$ in the type is not covered', while the path-covering one is 'subpath $/n$ is irrelevant'. The second message helps the programmer better.

For these reasons, path coverage should be considered by a correctness-checking type system. However, while we gave a precise semantic characterization of the correctness of a query with respect to data covering, this is not easy when path covering is considered. As an example, the path expression `(/a+/b+/c),(/a+/b+/c),(/a+/b+/c)` should be equivalent to its expansion `/a/a/a+/a/a/b+/a/a/c+/b/a/a...`. However, the first one should probably be considered wrong only when one of the nine atoms `/x` is useless, while the second one is suspect as soon as any one of the twenty-seven addends is useless.

In [2] we discuss some concrete notions of path-covering correctness and type rules; here we can only point out that the problem is relevant and difficult.

5 Conclusions and future issues

This work presents a type system in which both result analysis and query correctness analysis of XML queries can be conveniently expressed. Specifically, queries can be classified as strongly correct, weakly correct, or incorrect. We have seen that such classification describes an intuitive spectrum of query correctness characterisations.

Our type system, beside being the first to try correctness analysis for XQuery-like languages, also provides a framework in which different notions of correctness can be formally identified and studied, as exemplified in the discussion of Section 4. We are currently working on the design and comparison of such alternative notions.

Finally, we plan to augment the type language with other type operators, such as non-ordered sequences, ID and IDREFs types, so as to study our results in the context of a system closer to XML Schema.

References

[1] D. Chamberlin & J. Clark et al. XQuery 1.0: An XML Query Language. Technical report, W3C, 2001.

[2] D. Colazzo & G. Ghelli et al. Types for Correctness of Queries over Semi-structured data. `http://www.di.unipi.it/~colazzo/tcqssd.ps`.

[3] P. Fankhauser & M. Fernandez et al. XQuery 1.0 formal semantics. Technical report, W3C, 2001.

[4] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, Japan, 2000.

[5] D. Suciu. The XML Typechecking Problem. *SIGMOD Record Web Edition, Special Section on Data Management Issues in Electronic Commerce*, 2002.

[6] T. Milo & D. Suciu & V. Vianu. Typechecking for XML Transformers. In *ACM Symposium on Principles of Database Systems*, 2000.