Types for Path Correctness of XML Queries (Extended Abstract)

Dario Colazzo[1], Giorgio Ghelli[2], Paolo Manghi[2], and Carlo Sartiani[2]

[1] LRI - Université Paris Sud
[2] Dipartimento di Informatica - Università di Pisa

Abstract. If a subexpression in a query will never contribute data to the query answer, this should be regarded as an error. This principle has been recently accepted into mainstream XML query languages, but was still waiting for a complete treatment. We provide here a precise definition for this class of errors, and define a type system that is sound and complete, in its search for such errors, for a core language, under mild restrictions on the use of recursion in type definitions.

1 Introduction

A type system for a query language usually fulfills two different aims: computing a type for the query result (*result analysis*), and flagging parts of the query that do not match the structure of the data (*correctness analysis*), such as the use of a field name that is not present in the database schema. Result analysis and correctness analysis are inseparable in traditional languages, where errors prevent result generation. Query languages for semistructured data (SSD) and XML are different. They work by traversing paths on the tree or graph representation of data, and by filtering the result of path evaluations with predicates: when a path does not match the data in the database, its evaluation returns an empty result, but no exception is raised. For these languages, the type systems proposed up to now only analyze the result type, disregarding, to a large extent, the navigation-correctness problem [3, 15, 2].

This situation is now changing. Although result analysis remains the most studied issue, there is a growing interest on tools to statically identify those query fragments that cannot contribute to the query result; this information has also been shown to be useful for query optimization [12] and checking correctness of query reformulation in p2p databases [10]. In our paper [6], we took some first steps in this direction by presenting a notion of error, based on the intuition that a query is correct if it *may* match some data.

Concurrently with our investigations (starting from the August 2003 Working Draft), the W3C XML Query Working Group extended the type system of XQuery[3] by stating that it is a static error for any expression other than the empty-sequence expression to have the empty type [9].

Differently, in [7] we first provide a notion query correctness based on query dynamic semantics, and then provide type rules to statically check it. In this paper we present and discuss the main results of that work.

We start by defining which error we are trying to prevent (Section 3), and then we define a corresponding type system (Section 4). To keep the size of formal proofs tolerable, we base our analysis on a tiny abstract language, μXQ, based on the UnQL, Lorel, StruQL, XML-QL, Quilt, XQuery (and others) tradition [4, 1, 11, 3].

Along the lines of [6], the notion of correctness we present is *existential*, in the sense that a piece of code is correct if there exists at least one valid instance of its free variables such that an undesirable condition (result emptiness, in our case) is avoided. This is in sharp contrast with the *universal* notions of correctness verified by traditional type-systems, where a piece of code is correct if an undesired event is avoided under *every* valid instantiation of its free variables. This quantification

[3] XQuery is the standard query language for XML data developed by the W3C.

switch has deep consequences on the nature of the theory that one develops, as we will discuss in the paper.

Once we have defined the errors, we describe the type rules and the type system aimed to prevent them. These are based on a couple of technical tools, the collections of *locations* of wrong subqueries and *type-splitting* (Section 4). We prove that, at the price of a mild restriction on the use of recursion, this type system captures all and only the navigation-errors in the query (*soundness* and *completeness*).

As showed in [10], the proposed type system can be soundly extended to a wider language fragment comprising *where* clauses. Only a weaker form of completeness holds for this extension.

2 μXQ

μXQ is a minimal query language manipulating forests of ordered trees. It has been designed to be the minimal core of XQuery-like languages, hence it does not include features such as the *where* clause, node identity, document order, and recursive functions. μXQ term and query grammar is shown below. There f and t denote respectively forests and trees, and l ranges over a set of labels L. Furthermore, b denotes a leaf value of a base type B, forest concatenation ',' is associative, and $(), f = f, () = f$.

A typical μXQ query consists of a *binding* section (`let`/`for`), where variables are bound, and a `return` clause that builds the results. Variables can be either *for-variables* or *let-variables. for-variables* $(\overline{x}, \overline{y}, \overline{z})$ are bound to trees t (items) by a `for` binder. *let-variables* (x, y, z) are bound to forests f by a `let` binder. This distinction simplifies the formal treatment, but is not crucial to our approach.

$$\begin{array}{llll} \textbf{Forests} & f ::= () \mid t \mid f, f & \textbf{Trees} & t ::= b \mid l[f] \\ \textbf{Queries} & Q ::= () \mid b \mid l[Q] \mid Q, Q \mid \overline{x} \mid x \mid \overline{x}\,\texttt{child} :: l \mid \overline{x}\,\texttt{dos} :: l & & \\ & \quad \mid \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{return}\ Q \mid \texttt{let}\ x ::= Q\ \texttt{return}\ Q & & \end{array}$$

In the examples we will also use XPath-like clauses Q/l and $Q//l$, defined as:

$$Q\ /\ l \triangleq \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{return}\ \overline{x}\,\texttt{child} :: l \qquad Q\ //\ l \triangleq \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{return}\ \overline{x}\,\texttt{dos} :: l$$

The semantics $[\![Q]\!]_\rho$ of a query Q w.r.t. a substitution ρ is defined in Table 2.1; ρ maps every for-variable $\overline{x}$ free in Q to a tree, and every free let-variable x to a forest. $[\![\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2]\!]_\rho$ evaluates Q_2 in ρ extended with the binding $x \mapsto [\![Q_1]\!]_\rho$. $\prod_{t \in trees(f)} A(t)$, where $trees(f)$ returns the sequence of trees of f, is defined as the forest $A(t_1), \ldots, A(t_n)$ if $f = t_1, \ldots, t_n$, hence is $()$ when $f = ()$. $childr(t)$ returns the list of all children of a tree $l[f]$ (it is $()$ over B), $dos(f)$ returns the list of all *descendants-or-self* of all trees in a forest f. $f :: l$ selects all trees in f whose root is labeled l.

We will need the operation $(Q)_{|\beta}$, which, for any query Q and *location* β, locates the corresponding subquery. 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; some main cases of the definition are below:

$$\begin{array}{llll} (Q)_{|\epsilon} \triangleq Q & (l[Q])_{|0.\beta} \triangleq (Q)_{|\beta} & \ldots & \ldots \\ (\texttt{let}\ x ::= Q_0\ \texttt{return}\ Q_1)_{|i.\beta} \triangleq (Q_i)_{|\beta} & i \in \{0,1\} & & \\ (Q)_{|\beta} \triangleq \perp & & & \text{otherwise} \end{array}$$

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

3 Query Correctness

We start our investigation with the definition of a notion of navigation-correctness that only depends on the language semantics, namely, on the semantics of a subquery to be empty, rather than on its type to be empty.

Table 2.1. *μXQ semantics*

$$\begin{array}{llll}
\llbracket b \rrbracket_\rho & \triangleq b & \llbracket x \rrbracket_\rho & \triangleq \rho(x) \\
\llbracket () \rrbracket_\rho & \triangleq () & \llbracket Q_1, Q_2 \rrbracket_\rho & \triangleq \llbracket Q_1 \rrbracket_\rho, \llbracket Q_2 \rrbracket_\rho \\
\llbracket \overline{x} \rrbracket_\rho & \triangleq \rho(\overline{x}) & \llbracket l[Q] \rrbracket_\rho & \triangleq l[\llbracket Q \rrbracket_\rho] \\
\llbracket \overline{x}\ \texttt{child} :: l \rrbracket_\rho & \triangleq \mathit{childr}(\llbracket \overline{x} \rrbracket_\rho) :: l & \llbracket \texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2 \rrbracket_\rho & \triangleq \llbracket Q_2 \rrbracket_{\rho, x \mapsto \llbracket Q_1 \rrbracket_\rho} \\
\llbracket \overline{x}\ \texttt{dos} :: l \rrbracket_\rho & \triangleq \mathit{dos}(\llbracket \overline{x} \rrbracket_\rho) :: l & \llbracket \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 \rrbracket_\rho & \triangleq \prod_{t \in \mathit{trees}(\llbracket Q_1 \rrbracket_\rho)} \llbracket Q_2 \rrbracket_{\rho, \overline{x} \mapsto t}
\end{array}$$

$$\begin{array}{llllllll}
\mathit{dos}(b) & \triangleq () & \mathit{childr}(b) & \triangleq () & b :: l & \triangleq () & & \\
\mathit{dos}(l[f]) & \triangleq l[f], \mathit{dos}(f) & \mathit{childr}(l[f]) & \triangleq f & l[f] :: l & \triangleq l[f] & & \\
\mathit{dos}(()) & \triangleq () & & & () :: l & \triangleq () & & \\
\mathit{dos}(f, f') & \triangleq \mathit{dos}(f), \mathit{dos}(f') & & & (f, f') :: l & \triangleq f :: l, f' :: l & & \\
 & & & & m[f] :: l & \triangleq () & & m \neq l
\end{array}$$

Our notion is pragmatically acceptable, i.e. it is quite strict (stricter variants would rule out some common jargon) but it is not *too* strict (every non-correct query really has a problem). The next sections will show how this notion is technically acceptable, in the sense that it is possible to design a type system that matches it very precisely.

Assume the existence of two variables `$contacts` and `$mobilecontacts` (we use here $ to identify variables) with types:

$$\texttt{\$contacts} : (\texttt{data}[\texttt{phone}[...] \mid \texttt{mobile}[...]])+ \quad \texttt{\$mobilecontacts} : (\texttt{data}[\texttt{mobile}[...]])+$$

where | is a union type operator (i.e., either-or), and + indicates an arbitrary, non-empty, repetition, and consider the following queries:

Q_1 : `$contacts/fone` Q_2 : `$contacts/phone, $contacts/mobile`
Q_3 : `$contacts/phone` Q_4 : `$contacts/fone, $contacts/mobile`
Q_5 : `for $c in $contacts return ($c/phone, $c/mobile)`
Q_6 : `for $c in ($contacts, $mobilecontacts) return ($c/phone, $c/mobile)`

Q_1 is wrong, since it cannot match the data, while Q_2 is correct, since it perfectly matches the schema, i.e. the query surely matches data conforming to the given schema. Such queries lead to the simplest definition of correctness: a query is correct if it always finds some data, for every substitution of its free variables that is *valid*, i.e. coherent with the known structural information. Q_3, however, shows that this view is over-restrictive: the query is completely reasonable, but it may not match any data, in case we only have mobiles in the current database instance. This query is typical enough to convince us that, in this context, we have to opt for an existential notion of correctness: a query is correct if *there exists* a valid schema instance that is matched by the query. This is the notion we studied in [6], under the name 'weak correctness'.

Q_4 is troublesome. It is clearly wrong, since the first path cannot match the data, however the whole query can return a non-empty result, hence the whole query *does* match some valid schema instance, and is hence 'weak-correct'.

The point is that the non-matching subquery does not generate, according to μXQ semantics, a 'no-match-found error' which propagates up from `$contacts/fone` to the whole result. Moreover, we would *not* want such behavior, otherwise the subqueries of the good query Q_2 would raise and propagate that error as well, for example when no `$mobile` is in the database. In a programming language with error propagation we can say that something goes wrong iff the whole program returns 'error'. Here, instead, we have to talk about the result of every subquery. We hence arrive at the following notion of correctness (where non-() means 'syntactically different from ()'):

Definition 1. *Foreach-Exist (FE) Query Correctness: A query Q is correct w.r.t. a set of valid substitutions $\mathcal{R}$ if,* for each *non-()* subquery *Q' in Q,* there exists *$\rho \in \mathcal{R}$ such that, when Q is evaluated under ρ, Q' evaluates to a non-empty sequence.*

As desired, under this characterization, Q_2 and Q_3 above are correct, while Q_1 and Q_4 are not. Query Q_6, which corresponds to a typical XQuery jargon, is correct as well, if we apply the existential quantification to the bindings of the variables bound by `for`: at least one binding for `$c` exists (under a valid substitution for `$contacts` and `$mobilecontacts`) that makes `$c/phone` productive. Q_5 is correct a fortiori.

Once one accepts that correctness, in this context, has to be existentially quantified on substitutions and universally on subqueries, there is still space to consider a last variation, the *exists-foreach* version, where the quantification order is exchanged:

Remark 1. Exist-Foreach (EF) Query Correctness: A query Q is correct w.r.t. a set of valid substitutions $\mathcal{R}$ if *there exists* $\rho \in \mathcal{R}$ such that, *for each* non-() subquery Q' in Q, when Q is evaluated under ρ, Q' evaluates to a non-empty sequence.

While FE-correctness only requires that each subquery makes sense w.r.t. a different substitution, this stricter version requires the existence of at least one database that exploits every subquery. This variation is equivalent to FE-correctness on queries Q_1-Q_4, but it differs on queries Q_5-Q_6. In these queries, there exists no single substitution for `$c` that makes both `$c/phone` and `$c/mobile` productive at the same time. Since Q_5 and Q_6 are sensible queries, and correspond to XQuery usage patterns, we conclude that the exist-foreach version of correctness would be too strict for our purposes.

So, we have shown that our notion rules out some wrong queries and that its most natural immediate strengthening is too strict. Hence, we have shown that our notion is 'maximally strict'.

We have now to show that our notion is arguably not too strict, since it only flags queries that really have a problem. This is simple: by definition, if a query Q is not FE-correct, a non-() subquery Q' exists, such that for all $\rho \in \mathcal{R}$, Q' evaluates to an empty sequence. Hence, we have a non-() piece of code that is equivalent to (), and warning the programmer makes obviously sense.

To formalize FE-correctness we define $\mathit{Ext}(\rho, Q, \beta)$, the set of all valid substitutions that will be used to evaluate the subquery $(Q)_{|\beta}$ when Q is evaluated under ρ. These substitutions correspond to ρ extended with the bindings introduced by each traversed `let` or `for`:

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

$\mathit{Ext}(\rho, Q, \beta)$ is not just a singleton since each subquery in the scope of a $\texttt{for}\ \overline{x}\ \texttt{in}\ Q_0$ is evaluated once for each tree in $[\![Q_0]\!]_\rho$. Since $[\![Q_0]\!]_\rho$ may be the empty forest, $\mathit{Ext}(\rho, Q, \beta)$ may be empty as well.

We then define the set $\mathit{CriticalLocs}(Q)$ of the locations of Q where we will look for pieces of wrong code.

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

$\mathit{CriticalLocs}(Q)$ does not coincide with $\mathit{Locs}(Q)$ because, at least, all locations that reach a subquery that is () must not be tested for non-emptiness. In general, after a brief and complete analysis, one realizes that only errors located in subqueries from which the programmer explicitly started a `child`/`dos` navigation or a `for` iteration should be considered.

We can now formalize FE-correctness. A non-() subquery $(Q)_{|\beta}$ is correct if there exist $\rho \in \mathcal{R}$ and $\rho' \in \mathit{Ext}(\beta, Q, \rho)$ 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.

Definition 2. *FE 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 has an error at location $\beta \in \mathit{CriticalLocs}(Q)$ iff:*

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

(Observe that $Ext(\rho, Q, \beta) = \emptyset$ implies that Q has an error at β.) Q is correct w.r.t. $\mathcal{R}$ iff Q has no error w.r.t. $\mathcal{R}$.

4 Type System

Types and Judgments We adopt, essentially, XDuce's type language [13]. Types and type environments are defined as follows:

$$\begin{array}{ll}\textbf{Types} & T ::= () \mid B \mid (T, T) \mid (T \mid T) \mid l[T] \mid T* \mid X \\ \textbf{Environments} & E ::= () \mid X = T,\ E\end{array}$$

The type $()$ contains only the empty forest $()$, while B represents atomic types ($\texttt{String}, \texttt{Int}, ...$). The type sequence (T, T') represents the set of forests f, f', where f and f' belong to T and T', respectively. The union type $T \mid T'$ denotes the set of forest f, where f is either of type T or T'. The element type $l[T]$ denotes the set of trees $l[f]$ where f is of type T. Finally, the Kleene Star type $T*$ represents the set of forests $f_1, \ldots, f_n$, where $n \geq 0$ and each f_i is of type T.

A type environment E is a sequence of type definitions of the form $X = T$ where no type variable is bound to two types. We restrict to environments where only $l[]$-guarded vertical recursion is allowed, as in $X = l[X \mid ()]$. For example, 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 [14, 8], hence expressive enough to capture the essence of DTD and XML Schema [14, 17, 16].

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.

Type semantics $[\![T]\!]_E$ is standard and interprets a type T as the set of all forests with that type.

Type checking In the full version [7], we provide a set of complex algorithmic type rules capable of checking FE-correctness of a μXQ's query Q. The rules are based on the following judgments, where Γ is a variable environment that assigns Q's free variables to types:

$$\textbf{Judgments}\quad J ::= E;\ \Gamma \vdash_\beta\ Q : (T;\ \mathcal{S}) \ \mid\ E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q : (T;\ \mathcal{S})$$

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).

In $E;\ \Gamma \vdash_\beta\ Q : (T;\ \mathcal{S})$, 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 $\mathcal{S}$ and β will be discussed shortly. To analyze $\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2$, we compute a type T_1 for Q_1 and use the judgment $E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : (T_2;\ \_)$ to compute the type of Q_2 through a case-analysis on the type T_1.

Our typing judgments also return an error set $\mathcal{S}$, which contains a set of locations with shape $\beta.\alpha$, such that, for each α, the subquery of Q at α is not FE-correct. Observe that this is a sharp departure from the traditional approach, where the result of error-checking is just a boolean. We believe booleans are not enough, in a system that combines case-analysis with subquery quantification. Consider, for example, the following queries over $\texttt{\$contacts} : (\texttt{data}[\texttt{phone}[...]] \mid \texttt{data}[\texttt{mobile}[...]])+$.

```
Q5 : for  $c in $contacts              Q7 : for  $c in $contacts
     return  ($c/phone, $c/mobile)          return  ($c/fone, $c/mobile)
```

Because of universal quantification on subqueries (Definition 2), a query (Q, Q') is FE-incorrect iff either Q or Q' is. Because of existential quantification on substitutions, a query $\texttt{for}\ \overline{y}\ \texttt{in}\ x\ \texttt{return}\ Q$ is FE-incorrect iff Q is incorrect for every binding of

$\overline{y}$. Hence, a case-analysis-based type checking algorithm would compute the error-checking function $\mathrm{Err}_{\Gamma}(Q)$ as follows:

$$\mathrm{Err}_{\$c:(T_1|T_2)}(Q_7) = \bigwedge_{T\in\{T_1,T_2\}}(\mathrm{Err}_{\$c:T}(\$c/\mathit{fone}) \vee \mathrm{Err}_{\$c:T}(\$c/\mathit{mobile}))$$

As expected, Q_7 is deemed wrong because for every T_i at least one of `$c/fone` and `$c/mobile` is wrong. Unfortunately, the correct query Q_5 is deemed wrong as well: since each of the subcases `data`[`phone`[...]] and `data`[`mobile`[...]] makes one of the subqueries incorrect, the external conjunction returns true.

$$\mathrm{Err}_{\$c:(T_1|T_2)}(Q_5) = \bigwedge_{T\in\{T_1,T_2\}}(\mathrm{Err}_{\$c:T}(\$c/\mathit{phone}) \vee \mathrm{Err}_{\$c:T}(\$c/\mathit{mobile}))$$

The problem cannot be solved by playing with the boolean operators, since they exactly correspond to the quantifications in the definition of FE-correctness. However, we can generalize booleans to sets of locations, and use the following equations, where $\mathrm{ErrLeaf}(Q)$ returns the location of Q when Q is wrong.

$$\mathrm{Err}_{\$c:(T_1|T_2)}(Q_5) = \bigcap_{T\in\{T_1,T_2\}} (\{\mathrm{ErrLeaf}_{\$c:T}(\$c/\mathit{phone})\} \cup \{\mathrm{ErrLeaf}_{\$c:T}(\$c/\mathit{mobile})\})$$
$$\mathrm{Err}_{\$c:(T_1|T_2)}(Q_7) = \bigcap_{T\in\{T_1,T_2\}} (\{\mathrm{ErrLeaf}_{\$c:T}(\$c/\mathit{fone})\} \cup \{\mathrm{ErrLeaf}_{\$c:T}(\$c/\mathit{mobile})\})$$

This time $\mathrm{Err}(Q_5)$ is the intersection of two different singletons of locations, hence is empty. This corresponds to the fact that no subquery is always returning an empty result, hence no subquery is incorrect. However, $\mathrm{Err}(Q_7)$ is the intersection of two sets that both contain the location of `$c/fone`. This signifies that, for every well-typed substitution for $\$c$, the subquery `$c/fone` is always empty, hence the subquery is incorrect.

Our type rules follow the idea just described above. Hence, each time a case analysis is performed over a union type, different error sets $\mathcal{S}_i$ are computed, and their intersection is returned as output. The resulting system is quite precise, as we discuss in the following.

Soundness of Error and Type Checking In the full paper, we provide two possible type systems. The first version enjoys the canonical 'soundness' property: inferred types are upper bounds for the set of all possible results.

Definition 3. *$\mathcal{R}(E,\Gamma)$: For any well-formed type environment E and Γ well-formed in E, we define the set of valid substitutions as*

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

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

Theorem 1 (Upper Bound).

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

The next property one expects is some form of 'well typed terms never go wrong' property, that specifies that every run-time error is detected by the type system. Nonetheless, in this context we believe that one should first look for the opposite implication 'we will never bother you with a false alarm'. We expect that a type system based on our proposal would be used as an auxiliary tool in a programming environment based on a commercial language, and that the programmer would be allowed to ignore its error messages. As a consequence, most programmers would just ignore *all* the error messages, if there is the doubt that they do not correspond to real errors, but are just a figment of the type rules.

Hence we believe that, in this context, the essential 'soundness' property of error-checking is that expressed by Theorem 2, which goes the other way around with respect to the standard 'progress+subject reduction' combination.

Theorem 2 (Soundness of Existential Error-Checking).

$$E;\ \Gamma \vdash_\beta\ Q : (U;\ \mathcal{S})\ \wedge\ \beta.\alpha \in \mathcal{S} \qquad \Rightarrow \qquad Q \textit{ has an error at } \alpha \textit{ w.r.t. } \mathcal{R}(E,\Gamma)$$

Completeness of Error Checking Our first type system is not complete as it is not precise enough to detect all possible FE-errors. To illustrate, consider the type $Y = c[a[] \mid b[]]$ and the query:

$Q_8 =$ `for $x in $y/a return $y/b`

where `$y` is of type Y. Q_8 returns a sequence of `$y/b` iff `$y` has a child `a`, and returns () otherwise. The query is FE-incorrect, as there is no substitution that makes the subquery `$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 "()".

We solve these problems by refining our first type system with a *type splitting* strategy: when a variable with a union type is introduced, the rules perform a case-analysis on the different cases of the union, even when the union type operator is hidden inside the type (as in $Y = c[a[] \mid b[]]$).

Thus, the type splitting approach is based on enumerating the branches of the union types of typed variables (*splitting* the type), performing an independent analysis for each branch, and combining the results. Once more, the key technical tool that allows the correct result recombination is the collection of error indications as set of locations, in the way showed in previous examples.

More specifically, the amount of splitting is governed by a function $Split_E(T)$, which rewrites T to a set $\{T_1, \ldots, T_n\}$ such that $T_1 \mid \ldots \mid T_n$ is equivalent to T. Essentially, $Split_E(T)$ rewrites T in order to make $\mid$ be the outermost type operator. For example, type $c[a[] \mid b[]]$ is split into $\{c[a[]], c[b[]]\}$, and the query Q_8 presented above is analyzed once with $y : c[a[]]$ and once with $y : c[b[]]$. The subquery $(Q_8)_{|1}$ is (correctly) flagged as wrong, since the location 1 is in the error set of both runs of the analysis.

By splitting a type more and more finely, a more precise type analysis can be obtained, at the price of a more expensive type-checking process, since the rest of the query is checked once for every addend generated by splitting.

The definition of $Split_E(T)$ is non-trivial because of recursive type variables. Our key result is the fact that splitting can be stopped in front of $*$-types ($Split_E(T*) = \{T*\}$), and still the type system enjoys the completeness properties formalized by Theorem 3. A more complex type system, where unfolding depends on the query, may be worth studying. However, we claim that our solution, based on a mild restriction on the use of recursion, is acceptable in practice. We restrict to environments E for which recursion is guarded by a $*$ type constructor. 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 $*$.

Theorem 3 (Completeness of Error-Checking).

$$E;\, \Gamma \vdash_{\beta} Q : (\_;\, \mathcal{S}) \;\;\Rightarrow\;\; (Q \textit{ has an error at } \alpha \textit{ w.r.t. } \mathcal{R}(E,\Gamma) \;\Rightarrow\; \beta.\alpha \in \mathcal{S})$$

We believe that the existence of a non trivial core language, like μXQ, where the analysis is complete is an important result, because it formally measures the quality of the match between our notion of error and our type system. Moreover, completeness is a guarantee of good precision for possible, more realistic, extensions of μXQ exploiting our type-checking technique.

We conclude by observing that, tough based on case analysis over union types, complexity of our sound and complete type-checking algorithm is acceptable in a wide class of cases (see [7, 5] for details).

5 Conclusions and Future Work

We have presented a type system that performs both result analysis and navigation-correctness analysis for a minimal query language for tree-shaped data.

We have first given a precise definition of navigation-errors, and discussed its merits in relation with some possible alternatives. We introduced a first type system, which is sound and quite precise. We then introduced a more expensive type system that, when applied to schemas that satisfy a mild restriction on the alternation between $*$ and recursion, performs a correct and complete error-checking. This type system validates the claim that our notion of navigation-error is both meaningful for the programmer and amenable to machine-checking.

We defined the notions of universal and existential correctness, as well as a framework that can be used to check both families of errors.

We are currently investigating how to check correctness of backward axes.

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistuctured Data. *Journal of Digital Libraries, 1(1)*, pages 68–88, April 1997.
2. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with Data Values: Typechecking Revisited. In *Proceedings of the Twentieth Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001.
3. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, May 2003. W3C Working Draft.
4. P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of 5th International Workshop on Database Programming Languages*, Gubbio, Italy, September 1995.
5. D. Colazzo. *Path Correctness for XML Queries: Characterization and Static Type Checking*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2004.
6. D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types For Correctness of Queries Over Semistructured Data. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002), Madison, Wisconsin, June 6-7, 2002*, 2002.
7. D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types For Path Correctness of XML Queries. In *Proceedings of 7th ACM International Conference on Functional Programming (ICFP), Snowbird, Utah, USA, 2004.*
8. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.
9. D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, Aug. 2003. W3C Working Draft.
10. D. Colazzo and C. Sartiani. Typechecking queries for maintaining schema mappings in xml p2p databases. In *Proceedings of PLAN-X 2005*, 2005.
11. M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a Web-site management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(3):4–11, September 1997.
12. R. Guerra, J. Jeuring, and D. Swierstra. Generic validation of xpath data bindings. In *Proceedings of PLAN-X 2005*, 2005.
13. H. Hosoya and B. C. Pierce. XDuce: An XML Processing Language, 1999. Preliminary Report.
14. D. Lee, M. Mani, and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. Technical report, IBM Almaden Research, 2000. Technical Report - IBM Almaden Research.
15. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proceedings of the Nineteenth Symposium on Principles of database systems.* ACM Press, 2000.
16. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. Technical report, World Wide Web Consortium, May 2002. W3C Recommendation.
17. F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Third Edition). Technical report, World Wide Web Consortium, Feb 2004. W3C Recommendation.