Types for Path Correctness of XML Queries

Dario Colazzo Giorgio Ghelli Paolo Manghi Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
Via Buonarroti 2, Pisa, Italy
{colazzo,ghelli,manghi,sartiani}@di.unipi.it

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. In the process, we describe a dichotomy among *existential* and *universal* type systems, which is useful to understand some unusual features of our type system.

Categories and Subject Descriptors: H.2.3 [Database Management]: Languages-Query Languages

General Terms: Languages, Theory, Algorithms, Verification

Keywords: Type Correctness, XML Queries, XML Types.

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, since wrong paths just generate empty pieces of result. For these languages, the type systems proposed up to now only analyze the result type, disregarding, to a large extent, the navigation-correctness problem [6, 16, 3].

This situation is now beginning to evolve. In our paper [10], we presented a first notion of error, which was a stepping stone toward the one we propose here, based on the intuition that a query is correct if it *may* match some data. Along a similar line, the most recent versions of XQuery (starting from the August 2003 Working Draft) state that it is a static error for any expression other than the empty-sequence expression to have the empty type. Since a static error-checking system is only a tool toward an error-prevention aim, we start by defining which error we are trying to prevent (Section 3). The notion of correctness we define is *existential*, which means 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* (or *conservative*) notions of correctness found in programming languages, where a piece of code is correct if an undesired event is avoided under *every* valid instantiation of its free variables. This quantification 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 error, we define the type rules aimed to prevent it. Our type system is based on a couple of technical tools, the collections of *locations* of wrong subqueries (Section 4.2) and *type-splitting* (Section 5). We prove that, at the price of a mild restriction on the use of recursion, our type system infers types that provide both an upper and a lower bound for the actual values returned by a query, and captures all and only the navigation-errors in the query.

The completeness result is of course lost if the language is generalized to a realistic one, but is still interesting, because soundness results alone do not discriminate an interesting type-system from one that is completely trivial.

Our type system, although designed to deal with an *existential* notion of correctness, can be used to check *universal* notions of correctness at the same time. We base our analysis on a tiny language, μXQ, based on the UnQL, Lorel, StruQL, XML-QL, Quilt, XQuery (and others) tradition [7, 1, 13, 6].

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, similarly to λ-calculus for functional languages, hence we choose not to include features such as the *where* clause, node identity, document order, recursive functions. In Section 7 we discuss how to extend this work to those features.

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* $(\bar{x}, \bar{y}, \bar{z})$ 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 dis-

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.

tinction simplifies the formal treatment, but is not crucial to our approach.

Forests	f	::=	$() \mid t \mid f, f$
Trees	t	::=	$b \mid l[f]$
Queries	Q	::=	$() \mid b \mid l[Q] \mid Q,Q \mid \bar{x} \mid x$
			$\mid \bar{x}\,\texttt{child} :: l \mid \bar{x}\,\texttt{dos} :: l$
			$\mid \texttt{for}\ \bar{x}\ \texttt{in}\ Q\ \texttt{return}\ Q$
			$\mid \texttt{let}\ x ::= Q\ \texttt{return}\ Q$

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

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

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 $\bar{x}$ 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 undefined 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.

DEFINITION 1 ($(Q)_{|\beta}$:). *$(Q)_{|\beta}$ denotes the subterm of the query Q located by the location β, which is a sequence of 0's and 1's:*

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

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

3 Query Correctness

W3C states that a subquery is wrong when *its type* is empty but the query is different from $()$ [12]. We have first to explain why we cannot just adopt this as the definition of navigation-incorrectness.[1]

If a type system is used to identify a class of errors, the error must be defined first (e.g., a core-dump is an error), then the type rules must be introduced, and finally the adherence of the type-system findings with the semantic errors must be evaluated. A notion of error that depends on the type rules under definition prevents the investigation of this fundamental adherence question. For this reason, we start the 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.

[1] Actually, W3C documents do not advertise this statement as a notion of error, but only as a type rule.

In this section we propose our notion, and show that it 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 $\texttt{\$contacts}$ and $\texttt{\$mobilecontacts}$ (we use here \$ to identify variables) with types:

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

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

Q_1 : $\texttt{\$contacts/fone}$
Q_2 : $\texttt{\$contacts/phone}, \texttt{\$contacts/mobile}$
Q_3 : $\texttt{\$contacts/phone}$
Q_4 : $\texttt{\$contacts/fone}, \texttt{\$contacts/mobile}$
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 [10], 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 $\texttt{\$contacts/fone}$ 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 $\texttt{\$mobile}$ 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 2. *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 $\texttt{\$c}$ exists (under a valid substitution for $\texttt{\$contacts}$ and $\texttt{\$mobilecontacts}$) that makes $\texttt{\$c/phone}$ productive. Q_5 is correct a fortiori.

Table 2.1. *μXQ semantics*

$[\![b]\!]_\rho$	$\triangleq b$	$[\![x]\!]_\rho$	$\triangleq \rho(x)$	$[\![\bar{x}]\!]_\rho$	$\triangleq \rho(\bar{x})$	
$[\![()]\!]_\rho$	$\triangleq ()$	$[\![Q_1, Q_2]\!]_\rho$	$\triangleq [\![Q_1]\!]_\rho, [\![Q_2]\!]_\rho$	$[\![l[Q]]\!]_\rho$	$\triangleq l[[\![Q]\!]_\rho]$	
$[\![\bar{x}\,\texttt{child}::l]\!]_\rho$	$\triangleq childr([\![\bar{x}]\!]_\rho)::l$	$[\![\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2]\!]_\rho$	$\triangleq [\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho}$			
$[\![\bar{x}\,\texttt{dos}::l]\!]_\rho$	$\triangleq dos([\![\bar{x}]\!]_\rho)::l$	$[\![\texttt{for}\ \bar{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2]\!]_\rho$	$\triangleq \prod_{t \in trees([\![Q_1]\!]_\rho)} [\![Q_2]\!]_{\rho, \bar{x} \mapsto t}$			
$dos(b)$	$\triangleq ()$	$childr(b)$	$\triangleq \bot$	$b::l$	$\triangleq ()$	
$dos(l[f])$	$\triangleq l[f], dos(f)$	$childr(l[f])$	$\triangleq f$	$l[f]::l$	$\triangleq l[f]$	
$dos(())$	$\triangleq ()$			$()::l$	$\triangleq ()$	
$dos(f, f')$	$\triangleq dos(f), dos(f')$			$(f, f')::l$	$\triangleq f::l, f'::l$	
				$m[f]::l$	$\triangleq ()$	$m \neq l$

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

DEFINITION 3. *Substitution Extension*

$$Ext(\rho, Q, \varepsilon) \triangleq \{\rho\}$$

$$Ext(\rho, \texttt{let}\ x ::= Q_0\ \texttt{return}\ Q_1, 1.\beta) \triangleq Ext((\rho, x \mapsto [\![Q_0]\!]_\rho), Q_1, \beta)$$

$$Ext(\rho, \texttt{for}\ \bar{x}\ \texttt{in}\ Q_0\ \texttt{return}\ Q_1, 1.\beta) \triangleq \bigcup_{t \in trees([\![Q_0]\!]_\rho)} Ext((\rho, \bar{x} \mapsto t), Q_1, \beta)$$

otherwise: $(Q)_{|i} \neq \bot \Rightarrow Ext(\rho, Q, i.\beta) \triangleq Ext(\rho, (Q)_{|i}, \beta)$

FE-correctness can be formally captured in terms of substitution extension. A non-() subquery $(Q)_{|\beta}$ 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 first define the set $CriticalLocs(Q)$ of the locations of Q where we will look for pieces of wrong code.

$$CriticalLocs(Q) \triangleq \{\beta \mid ((Q)_{|\beta} = (\bar{x}\,\texttt{child}::l) \vee (Q)_{|\beta} = (\bar{x}\,\texttt{dos}::l))\} \cup \{\beta.0 \mid (Q)_{|\beta} = \texttt{for}\ \bar{x}\ \texttt{in}\ Q_0\ \texttt{return}\ Q_1\}$$

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

We can now formalize query correctness.

DEFINITION 4. *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:*

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

Dually, Q has an error at path $\beta \in CriticalLocs(Q)$ 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.)

While this notion of (navigation-)correctness is existential, one may still extend μXQ with other operations that more naturally lead to a universal notion of correctness, as happens with operations that modify persistent data. In this context, a piece of code would be correct if it were navigation-correct for at least one substitution and update-correct for every substitution.

4 Type System

4.1 Type Environments and Types

We adopt, essentially, XDuce's type language [14]. Types and type environments are defined as follows:

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

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 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 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 [15, 11], hence expressive enough to capture the essence of DTD and XML Schema [15, 18, 17].

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):

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

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. This condition admits an easy syntactic test (see [9] for details). The non-emptiness condition is not essential, but simplifies the type rules. In a nutshell, if $x : T$, and T may be empty, then $x : T \vdash \texttt{for}\ \bar{y}\ \texttt{in}\ x\ \texttt{return}\ Q$ may be incorrect just because T is empty, and the type rules would have to check this. A type T is well-formed in an environment E if every variable in T is defined in E.

4.2 Analysis of `for` and Locator Sets

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

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

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 $\bar{x}$ is associated to a tree type ($l[T']$ or B).

Our type rules (Table 4.1) are based on judgments of the form:

$$
\begin{array}{lll}
\textbf{judgments} & J ::= & E;\ \Gamma \vdash_\beta\ Q : (T;\ \mathcal{S}) \ \mid \\
& & E;\ \Gamma \vdash_\beta\ \bar{x}\ \texttt{in}\ T\ \rightarrow\ Q : (T;\ \mathcal{S})
\end{array}
$$

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 $\texttt{for}\ \bar{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2$, we compute a type T_1 for Q_1 (Table 4.1, rule TYPEFOR) and use the judgment $E;\ \Gamma \vdash_\beta\ \bar{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : (T_2;\ \_)$ to compute the type of Q_2 through a case-analysis on the type T_1 (rules (TYPEIN...)). By rule (TYPEINELSPLITTING), case-analysis stops when a tree type $l[T]$ is met, therefore $l[]$-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 $Split_E(T) = \{T\}$.

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

$$
\begin{array}{l}
Q_5 : \texttt{for \$c in \$contacts} \\
\quad \texttt{return (\$c/phone,\$c/mobile)} \\
Q_7 : \texttt{for \$c in \$contacts} \\
\quad \texttt{return (\$c/fone,\$c/mobile)}
\end{array}
$$

Because of universal quantification on subqueries (Definition 4), a query (Q, Q') is FE-incorrect iff either Q or Q' is. Because of existential quantification on substitutions, a query $\texttt{for}\ \bar{y}\ \texttt{in}\ x\ \texttt{return}\ Q$ is FE-incorrect iff Q is incorrect for every binding of $\bar{y}$. Hence, a case-analysis-based type checking algorithm would compute the error-checking function $\mathrm{Err}_\Gamma(Q)$ as follows:

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

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

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

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.

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

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.

The type rules are listed in Tables 4.1 and 4.2. We describe them by referring to the example.

Rule (TYPEFOR) starts the case-analysis, as previously discussed, propagates the error set $\mathcal{S}_1$, 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.[2]

Rules (TYPEINUNION) and (TYPEINCONC) perform the case analysis, and only put in $\mathcal{S}$ those locations that are wrong in both branches.

Rule (TYPEINELSPLITTING) stops the case-analysis, inserts the assumption $\bar{x} : m[T]$ in Γ, and falls back to standard type-checking (recall that we assumed $Split_E(T) = \{T\}$). At this point, rule (TYPECHILD) is applied. It requires the type of $\bar{x}$ to be a tree type $m[T']$, uses $E \vdash T' :: l \Rightarrow U$ (defined below) to restrict the content type T' to the tree types with structure $l[\_]$, 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 $\mathrm{Trees}_E(T)$ 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 $\bar{x}$ `dos` $:: l$ is obtained by restricting U' to the tree types with structure $l[\_]$. Rule (TYPELETSPLITTING) is standard, since we are assuming that $Split_E(T) = \{T\}$. We will later relax this assumption.

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

DEFINITION 5. *Subtrees Type Extraction: For any E well-formed and T such that $E \vdash T$ Def, we define $Trees_E(T)$ as follows (well-defined by Knaster-Tarski Th.):*

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

DEFINITION 6. *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 `false` $<$ `true`) that respects the following set of equations, well-defined by Knaster-Tarski theorem:*

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

Correctness of this definition is proved by the following theorem.

LEMMA 1 (EMPTY-FOREST-TYPE CHECKING). *For any well-formed environment E and type T well-formed in E:*

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

The judgment $E \vdash T :: l \Rightarrow U$ is defined by the rules in Table 4.2.

LEMMA 2 (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 $E \vdash T :: l \Rightarrow U$ terminates.*

LEMMA 3 (TYPE FILTERING CHECKING). *For any label l, well-formed type environment E and type T well-formed in E:*

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

4.3 Properties of the Type System

We provisionally assumed that $Split_E(T) = \{T\}$, which results in a completely standard (TYPELET) rule. This is sufficient to obtain the canonical 'soundness' property (Theorem 4): types are upper bounds for the set of all possible results. (This implies that this type system can be used to check *universal* notions of correctness, though we will not exemplify this fact here.)

DEFINITION 7. *$\mathcal{R}(E,\Gamma)$: 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 4 (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 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. But 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 5, which goes the other way around with respect to the standard 'progress+subject reduction' combination.

[2] The type $()$ is not to be confused with the empty type. It is a singleton type, which only contains the empty forest.

Table 4.1. *Query Type Rules*

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

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

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

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

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

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

(TypeLetSplitting)
$$\frac{E;\ \Gamma \vdash_{\beta.0}\ Q_1:(T_1;\ \mathcal{S})\quad Split_E(T_1)=\{A_1,\ldots,A_n\}\quad E;\ \Gamma,\ x:A_i \vdash_{\beta.1}\ Q_2:(U_i;\ \mathcal{S}_i)}{E;\ \Gamma \vdash_\beta\ \texttt{let}\ x:=Q_1\ \texttt{return}\ Q_2:(U_1\mid\ldots\mid U_n;\ \mathcal{S}\cup\bigcap_{i=1\ldots n}\mathcal{S}_i)}$$

(TypeFor)
$$\frac{E;\ \Gamma \vdash_{\beta.0}\ Q_1:(T_1;\ \mathcal{S}_1)\quad E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2:(T_2;\ \mathcal{S}_2)\quad \mathcal{S}=\text{ if } T_1\sim_E ()\text{ then }\{\beta.0\}\text{ else }\emptyset}{E;\ \Gamma \vdash_\beta\ \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2:(T_2;\ \mathcal{S}_1\cup\mathcal{S}_2\cup\mathcal{S})}$$

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

(TypeInElSplitting)
$$\frac{Split_E(m[T])=\{A_1,\ldots,A_n\}\quad E;\ \Gamma,\ \overline{x}:A_i \vdash_\beta\ Q:(U_i;\ \mathcal{S}_i)}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ m[T]\ \rightarrow\ Q:(U_1\mid\ldots\mid U_n;\ \bigcap_{i=1\ldots n}\mathcal{S}_i)}$$

(TypeInAtomic)
$$\frac{E;\ \Gamma,\overline{x}:B \vdash_\beta\ Q:(U;\ \mathcal{S})}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ B\ \rightarrow\ Q:(U;\ \mathcal{S})}$$

(TypeInConc)
$$\frac{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q:(T';\ \mathcal{S}_1)\quad E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ U\ \rightarrow\ Q:(U';\ \mathcal{S}_2)}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T,U\ \rightarrow\ Q:(T',U';\ \mathcal{S}_1\cap\mathcal{S}_2)}$$

(TypeInUnion)
$$\frac{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q:(T_1';\ \mathcal{S}_1)\quad E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_2\ \rightarrow\ Q:(T_2';\ \mathcal{S}_2)}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1\mid T_2\ \rightarrow\ Q:(T_1'\mid T_2';\ \mathcal{S}_1\cap\mathcal{S}_2)}$$

(TypeInVar)
$$\frac{E(X)=T\quad E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q:(U;\ \mathcal{S})}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ X\ \rightarrow\ Q:(U;\ \mathcal{S})}$$

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

(TypeChildNoMatch)
$$\frac{WF(E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{child}::l:(U;\ \mathcal{S}))\quad x:T\in\Gamma\ \wedge\ T=B}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{child}::l:(();\ \beta)}$$

(TypeDosNoMatch)
$$\frac{WF(E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{child}::l:(U;\ \mathcal{S}))\quad x:T\in\Gamma\ \wedge\ T=B}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{dos}::l:(();\ \beta)}$$

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

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

Table 4.2. *Filter Type Rules*

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

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

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

(StarFilt)
$$\frac{E\vdash\ T::l\Rightarrow U}{E\vdash\ T*::l\Rightarrow U*}$$

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

(VarFilt)
$$\frac{E\vdash\ E(X)::l\Rightarrow U}{E\vdash\ X::l\Rightarrow U}$$

THEOREM 5. *(Soundness of Existential Error-Checking) For any well-formed environment E, Γ well-formed in E, and query Q:*

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

4.4 Existential vs. Universal Error-Checking

It is now time to cite some standard theorems that one may expect to hold, and which do not. Recall query `$contacts/phone` from Section 3, and observe that it stops being correct if one substitutes `$contacts` with a query, or a term, of type $(data[mobile[...]])+$, although this is a subtype of the original type. This means that the canonical *subsumption* and *substitution* properties fail for this type system.

PROPERTY 1 (SUBSUMPTION). *In a type system that only checks a universally quantified notion of correctness, if $T' \leq T$ is a subtype relation such that $T' \leq T \Rightarrow [\![T']\!]_E \subseteq [\![T]\!]_E$, then*

$$E;\ \Gamma \vdash_\varepsilon\ Q : (U;\ \emptyset)\ \wedge\ (x : T) \in \Gamma\ \wedge$$
$$\wedge\ E;\ \Gamma \vdash_\varepsilon\ Q_1 : (T';\ \emptyset)\ \wedge\ T' \leq T$$

(where $Q : (\_;\ \emptyset)$ means that Q has no static type error) implies

$$E;\ \Gamma \vdash_\varepsilon\ Q\{x \leftarrow Q_1\} : (U';\ \emptyset)\ \wedge\ U' \leq U$$

PROPERTY 2 (WELL-TYPED SUBSTITUTION). *In a type system that checks a universally quantified notion of correctness,*

$$E;\ \Gamma \vdash_\varepsilon\ Q : (U;\ \emptyset),\ \ \rho \in \mathcal{R}(E,\Gamma),\ \ \textit{and}\ \ [\![f]\!]_\rho = \rho(x)$$

(where f is a term of the subgrammar $() \mid b \mid l[f] \mid f, f'$) implies

$$E;\ \Gamma \vdash_\varepsilon\ Q\{x \leftarrow f\} : (U;\ \emptyset).$$

Subsumption and *substitution* are consequences of the universal nature of the errors one looks for in traditional type system. There, every instantiation of a variable with a type-correct value is guaranteed not to fail, hence, if we substitute the variable with a type-correct expression, no error will arise.

Subsumption derives from the universal nature of the checked errors as well: if no value in a type creates problems, a smaller type creates no problem a fortiori.

Substitution and subsumption are so deeply ingrained in the techniques we use to design type systems and to prove their properties, that their failure implies that nothing should be taken for granted. For example, the fact that a type is just an upper bound of the set of values returned by a query (Theorem 4) creates no problem in a system that enjoys subsumption: whatever can be proved using this upper-approximation would be true a fortiori if one used a better approximation (i.e., a subtype). On the other hand, with an existential notion of error, when types become smaller, more errors appear, hence an upper-bound approximation is going to mask some problems.

In the end, we have been able to design a framework where the pieces fit together, but some unusual features will show up. The fact that our notion of 'soundness of error-checking' (Theorem 5) goes the other way round with respect to canonical type systems is a first example.

4.5 Combining Universal and Existential

The machinery we presented can be actually used to check both existential and universal notions of correctness. To this aim, the inferred type can be used in the standard way, so that a universally-wrong subexpression will not match any type rule. For example, we may have an integer + operator and the following rule.

$$\frac{E;\ \Gamma \vdash_{\beta.0}\ Q : (\text{Int};\ \mathcal{S}) \qquad E;\ \Gamma \vdash_{\beta.1}\ Q' : (\text{Int};\ \mathcal{S}')}{E;\ \Gamma \vdash_\beta\ Q + Q' : (\text{Int};\ \mathcal{S} \cup \mathcal{S}')}\ (\text{TYPE+})$$

If either Q or Q' has a type that is different from *Int*, the expression will not type-check, nor will it type-check if one of them has a type $\text{Int} \mid T$.

In such a system, the Upper Bound Theorem would imply the traditional Error-Soundness theorem, going the other way round with respect to Theorem 5.

THEOREM 6. *(Soundness of Universal Error-Checking) For any well-formed environment E, Γ well-formed in E, and well-formed Q:*

$$E;\ \Gamma \vdash_\beta\ Q : (U;\ \_)\ \Rightarrow$$
$$\Rightarrow\ Q \text{ has no universal-error w.r.t. } \mathcal{R}(E,\Gamma)$$

5 Type-Splitting

5.1 Motivation and Definition

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.[3] For the core-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* FE-error. (No other proposed type system enjoys this property of "completeness over the core".)

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

`x/mobile, 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 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

[3] In the traditional view, soundness means: well-typed programs never go wrong. This is enjoyed by any type system that consider every program as ill-typed, since it finds every universal error in every program.

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. (Similar phenomena happen in all related type systems we are aware of, including the XQuery type system.)

We solve these problems by defining a type system that, when a variable with a union type is introduced, performs a case-analysis on the different cases of the union, even when the union type operator is hidden inside the type (as in $c[a[] \mid b[]]$). In our setting, this amounts to defining a non-trivial version of the $Split()$ function.

We will prove that the resulting type system detects all FE-errors, and infers a type that provides both an upper and a lower bound for the set of all possible query results (Theorem 14).

Of course, the existence of any kind of correct and complete static analysis also shows that the language is, in some sense, poor. Specifically, our result relies on a monotonicity property of μXQ (Lemma 10) that would not hold in most realistic extensions of the language. Still, the existence of a core 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.

The error-complete 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. The amount of splitting is governed by the function $Split_E(T)$ (Definition 8), 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.

Our key result is the fact that splitting can be stopped in front of $*$-types ($Split_E(T*) = \{T*\}$), and still the new type system enjoys the completeness properties formalized by Theorem 16 below. Hence, for example:

$$Split_E((\texttt{data}[\texttt{phone}[...] \mid \texttt{mobile}[...]])*) = \{(\texttt{data}[\texttt{phone}[...] \mid \texttt{mobile}[...]])*\}$$

$$Split_E(\texttt{phone}[...] \mid \texttt{mobile}[...]) = \{\texttt{phone}[...];\ \texttt{mobile}[...]\}$$

$$Split_E(a[(b[], c[]) \mid (d[], e[])]) = \{a[b[], c[]];\ a[d[], e[]]\}.$$

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 $y : ()$. But, consider the following generalization of Q_8, where $(/a)^n$ stands for n consecutive occurrences of $/a$:

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

To catch the error, Y must be unfolded $n+1$ times. This means that we cannot decide how deeply Y has to be unfolded before looking at the query under consideration.

A more complex type system, where unfolding depends on the query, may be worth studying. However, we claim that a simpler solution is acceptable in practice, 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 [9] 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 8).

Hence, we have to prove that an assumption like $y : (c[a[X] \mid b[X]])*$, where union types are guarded by $*$, does not need to be split any further. Splitting was needed, for query Q_8, to detect a situation where the correctness of $(Q_8)_{|1}$ depended on the existence of two *mutually incompatible* paths c/a and c/b inside the type of y. The key observation is the fact that these paths are not incompatible when y has type $c[(a[] \mid b[])*]$, since $c[a[], b[]]$ is a legitimate value for y. The non-existence of incompatible paths in types where union is guarded by $*$ is formalized in Lemma 12, and Theorem 16 shows that this property is enough to guarantee error-completeness.

We claim that our restriction is "mild". Indeed, it is respected by *all* the schemas reported in the W3C document "XML query use cases" [8].

DEFINITION 8. *$Split_E(T)$: If E is $*$-guarded, then:*

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

$Split_E(T)$ is well-defined by Knaster-Tarski theorem. 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.

Splitting preserves type semantics.

LEMMA 7. *For each $*$-guarded environment E and type T defined in E:*

$$[\![T]\!]_E \quad = \quad \bigcup_{A \in Split_E(T)} [\![A]\!]_E$$

5.2 Simulation and Query Monotonicity

To characterize the precision of the type system, we define now a pre-order relation on forests $f \sqsubseteq f'$, *forest simulation*. This pre-order compares forests as if they were sets of trees instead of sequences, so that, for example, $a[], a[] \sqsubseteq a[]$ and $a[], b[] \sqsubseteq b[], a[]$, but implies path inclusion, so that if a/b is a path of f and $f \sqsubseteq f'$, then a/b is a path of f' as well. Simulation has the property that, if a query is correct when run with $y = f$ and $f \sqsubseteq f'$, then the query is correct when run with $y = f'$. Simulation allows us to formally specify that our type inference technique is precise 'up-to-simulation', and is a key tool in our proof of error-completeness.

DEFINITION 9. *Forest Simulation $f \sqsubseteq f'$: Simulation $f \sqsubseteq f'$ is the smallest relation on forests that respects the following conditions:*

$$\begin{array}{lcl} b \sqsubseteq b & & \\ l[f_1] \sqsubseteq l[f_2] & \Leftrightarrow & f_1 \sqsubseteq f_2 \\ f \sqsubseteq f' & \Leftrightarrow & \forall t \in trees(f).\ \exists t' \in trees(f').\ t \sqsubseteq t' \end{array}$$

LEMMA 8. *For each f and f':*

$$(f \sqsubseteq f' \ \wedge\ f \neq ()) \ \Rightarrow\ f' \neq ()$$

Lemma 10 states that μXQ queries are monotone with respect to $\sqsubseteq$ extended to substitutions in the obvious way:

$$\rho \sqsubseteq \rho' \ \Leftrightarrow_{def}\ \forall x \in dom(\rho).\ \rho(x) \ \sqsubseteq \rho'(x).$$

We are not talking here about the usual (trivial) set-of-values monotonicity property, that states that, if the set of values that the variable $\$x$ is allowed to range over increases, then the set of values that can be assumed by the query result increases as well. We are stating here a much stronger property that specifies that queries are monotone with respect to a pre-order that is defined on values (forests, in this case), something similar to strictness properties for functional languages. This strong property is not needed in order to prove soundness results, but is crucial for completeness.

Query monotonicity depends on monotonicity of axis steps.

LEMMA 9 (MONOTONICITY OF AXIS STEPS).

$$\begin{array}{lll} \forall f, f'.\ f \sqsubseteq f' & \Rightarrow & f :: l \sqsubseteq f' :: l, \\ & & dos(f) \sqsubseteq dos(f'), \\ & & childr(f) \sqsubseteq childr(f') \end{array}$$

LEMMA 10 (QUERY MONOTONICITY).

$$\forall Q, \rho, \rho'.\ \rho \sqsubseteq \rho' \ \Rightarrow\ [\![Q]\!]_\rho \sqsubseteq [\![Q]\!]_{\rho'}$$

Query monotonicity implies monotonicity of substitution extension.

LEMMA 11 (EXTENSION MONOTONICITY).
For any well-formed query Q and pair of substitutions ρ_1 and ρ_2 such that $FV(Q) \subseteq dom(\rho_1) = dom(\rho_2)$ and $\rho_1 \sqsubseteq \rho_2$, $\forall \beta \in Locs(Q)$.

$$\forall \rho' \in Ext(\rho_1, Q, \beta).\ \exists \rho'' \in Ext(\rho_2, Q, \beta).\ \rho' \sqsubseteq \rho''$$

Finally, Lemma 12 shows that, after splitting, all the types we get are closed for finite $\sqsubseteq$-upper-bounds. Very informally, this captures the notion of 'no mutual exclusion' among paths, and implies that, if we use substitutions based on different f_i's in different branches of a typing proof, we can then combine all these branches, because one upper-bound of those f_i-based substitutions exists that is acceptable as well. (You do not find this lemma in the canonical type papers, because it is strictly related with the existential interpretation of typing.) This is the key lemma that allows us to prove that this type system is "complete", i.e. that, if a type U is inferred for a query Q, every element of U is actually a possible result for Q, modulo $\sqsubseteq$ (Theorem 14). The combination of soundness and completeness of type inference gives us a soundness-and-completeness property for error-checking as well: our type system discovers all and only the FE-errors of the analyzed query (Theorem 15 and Theorem 16).

LEMMA 12. *For any type A defined in a $*$-guarded environment E, if $Split_E(A) = \{A\}$ then,*

$$\forall f_1, \ldots, f_n \in [\![A]\!]_E.\ \exists f \in [\![A]\!]_E.\ f_i \sqsubseteq f\ i = 1 \ldots n$$

5.3 Soundness and Completeness of Type Inference and Error Checking

The system enjoys soundness and completeness of type inference and error-checking. The full proofs can be found in [9].

THEOREM 13 (UPPER BOUND). *For each query Q, well-formed E, and Γ well-formed in E:*

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

We can finally prove that, when E is $*$-guarded, type inference is complete up-to-simulation, and error-checking is complete.

THEOREM 14 (LOWER BOUND). *For each query Q, $*$-guarded E, and Γ well-formed in E:*

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

THEOREM 15 (SOUNDNESS OF ERROR-CHECKING). *For each query Q, $*$-guarded E, and Γ well-formed in E:*

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

THEOREM 16 (COMPLETENESS OF ERROR-CHECKING). *For each query Q, $*$-guarded E, and Γ well-formed in E:*

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

5.4 The Cost of Case-Analysis

Our type system uses case-analysis to type-check for-iteration, and to examine the alternative types generated by type-splitting when new variables are inserted into the environment. This case-analysis approach is not the only possibility to type-check `for` $\bar{x}$ `in` Q_1 `return` Q_2. XQuery, for example, promotes the type of Q_1 to a quantified disjunction of tree types [12], and then analyzes Q_2 just once, associating $\bar{x}$ with the disjunctive type. While this makes type checking more efficient, it makes it far less precise. For example, if x is a variable of type $r[a[], b[], c[]]$, the subquery $\bar{y}$ of `for` $\bar{y}$ `in` $x/*$ `return` $\bar{y}$ (where $*$ matches any label) would be analyzed once under the assumption $\bar{y} : (a[] \mid b[] \mid c[])+$, because $\bar{y}$ may assume one of the $a[]$, $b[]$, $c[]$ types (hence the $a[] \mid b[] \mid c[]$) and $\bar{y}$ will be evaluated more than once (hence the $+$). The query will then have type $(a[] \mid b[] \mid c[])+$, and thus, its use in a context where the type $a[], b[], c[]$ is expected, would raise a static error. While this can be mitigated by the use of some dynamic typing, we think that a type system where record types are treated with a greater precision is definitely worth studying. (In our type system, the query is assigned its exact type $a[], b[], c[]$.)

Our techniques are, in principle, quite expensive: the natural implementation of nested case-analysis is a nested loop, whose execution time is exponential with the nesting level.

However, we have examined many queries and always found that they can be type-checked without nested case-analysis, thanks to their adherence to some restrictions on the shape of the query and of the involved types. In this section we describe these conditions, and explain why they make type-checking feasible.

We start with for-iteration case-analysis, assuming that no type-splitting takes place.

We say that a type is label-deterministic iff, in its syntax tree, it is never the case that, after expanding recursive types with their definitions, two subterms `l[T]` and `l[U]` are found with the same label `l` but two different content-types `T` and `U`. Label-deterministic types are extremely common; every type defined by a DTD is label-deterministic.

In a nutshell, assuming that no type-splitting takes place, the following query still requires an exponential time to be type-checked, because the subquery that starts with `for $x2` has to be type-checked twice, once under the assumption `$x1:a1[T1]`, and once under the assumption `$x1:b1[U1]`, and every other `$xi` duplicates the needed time as well:

```
for $x1 in ( a1[...], b1[...]  )
...
for $xn in ( an[...], bn[...]  )
return ($x1,...,$xn)
```

However, if the type of `$x1` only had one tree type at the top level, for example if it were `a1[T1]`, or `a1[T1]*`, or `a1[T1] | (a1[T1],a1[T1])`, then the subquery could only be type-checked once, under the assumption `$x1:a1[T1]`, hence this variable would not contribute to the exponential grow of type-checking complexity.[4]

Hence, the following query can be type-checked efficiently, if all the free variables have label-deterministic types, and if the final expression `E` is either for-free or has in turn the same shape:

```
for $x1 in $y1/a//b/c
...
for $xn in $yn/c/*/e
return E($x1,...,$xn)
```

Intuitively, the type of `$y1/a//b/c` is a combination of instances of a unique type `c[T]` using '|', ',', and '*': the `c[]` comes from the final step `/c` of the path, and every `c` is associated with the same `T` since the type of `$y1` is label-deterministic. Hence, the rest of the query can be type-checked just once, under the assumption `$x1:c[T]`. The type of `$y1/a//b/c` can be efficiently computed as well. Every `$yi` with $i > 1$ may be either a free variable or it may be `$xj` with $j < i$, hence we also have to observe that all the `$xi`'s are associated with label-deterministic types.

This class of queries is formally defined in [9]. This class is big enough to include most of the queries presented in [8]. If a query is out of this class because it contains just a couple of 'exotic' iterations, the performance of the type-checking algorithm degrades gracefully. Of course, if the query systematically differs from that query, and is deeply nested, type checking becomes unfeasible.

for-iteration. By analyzing the types used in [8] and other repositories over the Web, we have verified that, in the vast majority of cases, whenever union is used to specify an element-content, then union is $*$-guarded. Therefore, in these cases, type-splitting behaves as the identity mapping, i.e. types are not split. For example, consider the following DTD that we borrow from [8]:

```
<!DOCTYPE report [
   <!ELEMENT report (section*)>
   <!ELEMENT section (title, content)>
   <!ELEMENT title (#PCDATA )>
   <!ELEMENT content (#PCDATA | anesthesia | prep
             | incision | action | observation)*>
   <!ELEMENT anesthesia (#PCDATA)>
   <!ELEMENT prep ( (#PCDATA | action)* )>
   <!ELEMENT incision ( (#PCDATA | geography
             | instrument)* )>
   <!ELEMENT action ( (#PCDATA | instrument )* )>
   <!ELEMENT observation (#PCDATA)>
   <!ELEMENT geography (#PCDATA)>
   <!ELEMENT instrument (#PCDATA)>
]>
```

In this DTD, union is intensively used to specify element-contents. However, it is always $*$-guarded. Hence, each element type reached by case-analysis is never actually split, since $Split_E(m[(T)*]) = m[(T)*]$. Actually, we have found a few schemas where union is not $*$-guarded. We report here the only such example from [8].

```
<!ELEMENT bib (book* )>
<!ELEMENT book (title, (author+ |editor+ ),
          publisher, price )>
<!ATTLIST book year CDATA #REQUIRED >
<!ELEMENT author (last, first )>
<!ELEMENT editor(last, first, affiliation )>
<!ELEMENT title (#PCDATA )>
<!ELEMENT last (#PCDATA )>
<!ELEMENT first (#PCDATA )>
<!ELEMENT affiliation (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price (#PCDATA )>
```

The union `(author+ | editor+ )` is not $*$-guarded, but its two immediate components `author+` and `editor+` are. By looking at repositories of schemas over the Web we have verified that when union is not $*$-guarded then it either involves a few $*$-guarded types (as above) or types that contain a very small number of nested types. Hence, splitting always produces very few types. An example of types with low degree of nesting of not $*$-guarded types is given by the following DTD, which describes time stamps used in Unix systems management.

```
<!ELEMENT TimeStamp (DateTime |
                     (Seconds, Microseconds?))>
<!ELEMENT DateTime (#PCDATA)>
<!ELEMENT Seconds (#PCDATA)>
<!ELEMENT Microseconds (#PCDATA)>
```

The other cases we have found of not $*$-guarded use of union types are not far from this one. All of them feature a very low degree of nesting. We suspect that, in general, deeply nested not $*$-guarded unions are difficult to work with in practice, hence are avoided by schema designers.

We plan further investigations about the patterns, in types and queries, that we have discovered, since several applications may profit from these regularities.

6 Related Work

Our Previous Work The work we presented in [10] was a first step toward the system we have here. In that work we compared the universal and existential notions of correctness, but the notions of weak and strong correctness we proposed were, respectively, too weak and too strong. Weak correctness accepted queries such as Q_4 : `$contacts/fone`, `$contacts/mobile` from Section 3, while strong correctness refused queries like Q_3 : `$contacts/phone`.

XDuce XDuce [14] 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

[4] In the third case (`a1[T1] | (a1[T1],a1[T1])`), the trivial algorithm would actually type-check the subquery three times under the same assumption `$x1:a1[T1]`, but this can be easily amended using standard memoization techniques.

since it is an example of typed language for XML that explicitly provides a notion of type correctness. XDuce supports a *universal* notion of correctness for patterns: functions are correct if and only if their bodies specify a matching pattern (a function case) for all possible alternatives described by the input type. As discussed in the paper, we believe that this notion of correctness, although well suited for a programming language, is too restrictive for an XML query language.

CDuce CDuce [4] is a language that derives from XDuce but adopts a more sophisticated type system, featuring function types, intersection, and negation types. CDuce is not specialized for XML, but the typical XDuce idioms can be easily encoded. CDuce performs sophisticated correctness analysis, but it adopts the same universally-quantified definition of correctness as XDuce: type checking ensures that if a function is well typed then *every* possible input value is matched by at least a branch pattern in the function body. We are working with the CDuce team to extend our approach to that language and the derived query language CQL [5], and the preliminary results are very promising.

XQuery XQuery type inference [12] recursively infers a type for every subexpression, starting from the types known for the input variables. As we discussed in Section 4, it does not perform type case-analysis, which makes type inference faster, but makes the inferred types less precise. Very recently, a new rule has been added to XQuery type system that states that it is a static error for any expression other than the empty-sequence expression to have the empty type. This new rule is not sufficient to achieve error-checking completeness, because of the minor precision of XQuery type inference. If the system were extended with union-types case-analysis in order to have a higher precision, then the error-reporting approach should be extended with some technique related to our locations-set approach.

In the previous versions of the standard, no navigation-error-checking was performed. As we stated in the introduction, even in absence of explicit navigation-error-checking, the inferred type can point out the presence of navigation-problems, but with some limits. When no match is possible for a subquery, the type system will typically (but not always) assign an empty-sequence type to that subquery. This empty-sequence type may become the final type of the query, hence telling the programmer that something went wrong. But if the subquery is inside an element constructor that accepts empty content, or is combined with an expression with a non-empty type as in "`error,Q`", then the final type of the query will not be an empty-sequence, and the error may be completely hidden.

k-pebbles proposal Dan Suciu et al. develop a formal framework for the definition of result analysis tools[16, 2]. These papers define some upper bounds to what can be accomplished by result type analysis. Our results do not contradict these, since we study here a language that is weaker than k-pebbles automata.

7 μXQ and XQuery

μXQ, although inspired by XQuery, omits many important features, which we briefly discuss here.

We did not consider a *where* clause in our version of the FLWOR construct. In the presence of *where*, the identification of navigation-errors with constant emptiness of a subquery becomes questionable. Consider the following query.

`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 FE-error iff *Condition* is always false. Such a notion of type error sounds heretic, but it should not be discarded too lightly. For a programmer, knowing that a condition is uniformly false is as useful as knowing that a subquery is always empty. Its undecidability in realistic languages is not really a problem, since any type-checking algorithm is approximate in a real language.

For an alternative definition, let us define, for each query Q, *where-drop*(Q) as the query obtained by removing all *where* clauses from Q. A less heretic notion of type-error can now be defined, by changing Q' into *where-drop*(Q') in our definition (from Section 3):

REMARK 2. *Where-dropping 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, *where-drop*(Q') *evaluates to a non-empty sequence.*

Since any Q' appearing in the *where* condition is a subquery of Q, this definition does not ignore where clauses altogether. It checks that no navigation-error is hidden inside a where clause, but, when the correctness of the whole FLWR expression is considered, the filtering action of the *where* clause is ignored.

While we are more attracted by the where-dropping notion of correctness, the relationship between the two alternatives is subtle, and we believe it is worth investigating. An *if then else* construct would raise similar issues.

The presence of a *where* (or *if then else*) clause would have another important (and orthogonal) effect on the type system: the language would lose monotonicity. Consider the following query:

`for $x in a[] where not empty($y) return $x`

When the value of $\$y$ grows from an empty to a non-empty sequence, the semantics of the query goes down from non-empty to empty. Hence, Lemma 10 stops holding, and we lose semi-completeness of type-inference (Theorem 14) and completeness of error-checking (Theorem 16). This is not really a problem. We did not define a complete system over the monotone core because we hoped to extend completeness to a realistic language; indeed, no complete semantic analysis can be decidable on a Turing-complete language. Completeness over the core is important because it is a formal way to measure the precision of the type system.

We ignored issues related to "document order", such as the fact that any path expression, in XQuery, returns its result in document order. If the type of the expression has the shape $(T_1 \mid \ldots \mid T_n)*$, where all the T_i's are tree types, the type does not change when the sequence is re-ordered. Otherwise, let us use $\text{UpperTrees}_E(T)$ to denote the set of the tree types of the trees that can be found at the top level of a forest of type T:

$$\text{UpperTrees}_E(T) \triangleq \{U \mid \begin{array}{l} T \rightarrow_e^E U,\ U \text{ is a tree type,} \\ \text{and not exist } e', l, e''.\ e = e'.l.e''\end{array}\}$$

the type T of a path expression can be weakened to its supertype $(U_1 \mid \ldots \mid U_n)*$, where

$$\{U_1, \ldots, U_n\} = \text{UpperTrees}_E(T),$$

without compromising Theorems 13, 14, and 16. This supertype does not carry any information on the order of the trees.

We ignore reverse axis (*parent*, *ancestor*...). The current version of XQuery assigns trivial types to these axes, and we can do nothing better unless we change some of our fundamental assumptions. We ignore node identity and the issue of reference vs. copy semantics, because they have very little effect on the type system. We ignore the issue of predefined functions, (recursive) function definition and

invocation, and validation, because we think that they may be dealt with using standard techniques.

To sum up, we do not expect problems in the extension of our techniques to a full-scale language, although this will have to be carefully studied. The resulting system will be sound but not complete, which is the only property one can aim to when a complete language is treated. However, it would still be the only type system which is complete on the core language, and the only type system which has been consciously designed to deal with both existential and universal errors, and, specifically, to deal with navigation errors.

8 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, and defined a framework that can be used to check both families of errors.

Since our type system is based on extensive use of case-analysis, we want now to study techniques that help reducing the overall computational complexity. Our key result, the fact the no splitting is needed below a $*$, is already significant, since most subqueries have a type $T*$, but we believe that in many other situations type-splitting can be either avoided or performed lazily, hence making our type-system efficiently implementable over realistic programs and schemas.

9 Acknowledgments

The CDuce team (Véronique Benzaken, Giuseppe Castagna, and Alain Frisch) contributed to this work with useful comments about type-splitting. A discussion with Phil Wadler was deeply influential for reaching our definition of query correctness.

10 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. Typechecking XML Views of Relational Databases. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, 16-19 June 2001, Boston, Massachusetts, USA, Proceedings. IEEE Computer Society, 2001*, pages 421–430, 2001.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with Data Values: Typechecking Revisited. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63. ACM Press, 2003.

[5] V. Benzaken, G. Castagna, and C. Miachon. CQL: a Pattern-based Query Language for XML. In *Proceedings of 20th Bases de Données Avancées (BDA) (2004)*, 2004.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] H. Hosoya and B. C. Pierce. XDuce: An XML Processing Language, 1999. Preliminary Report.

[15] 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.

[16] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 11–22. ACM Press, 2000.

[17] 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.

[18] 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.