JFP **16** (4&5): 621–661, 2006. © 2006 Cambridge University Press
doi:10.1017/S0956796806005983 Printed in the United Kingdom

Static analysis for path correctness of XML queries

DARIO COLAZZO

Laboratoire de Recherche en Informatique (LRI), Bat 490 Université Paris Sud, 91405 Orsay Cedex, France
(*e-mail:* dario.colazzo@lri.fr)

GIORGIO GHELLI, PAOLO MANGHI and CARLO SARTIANI

Dipartimento di Informatica – Università di Pisa, Largo B. Pontecorvo 3, Pisa, Italy
(*e-mail:* {ghelli,manghi,sartiani}@di.unipi.it)

Abstract

A part of a query that will never contribute data to the query answer 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 essential to understand some specific features of our type system.

Capsule Review

With the emergence of W3C XQuery, XML query languages have, for a good reason, attracted both industrial and research attention. One of the most unusual and interesting characteristics of their design is how they encompass both database and functional language principles. This paper is a key step toward a complete treatment of static typing for XML query languages, which in turn enables useful error detection for the programmer and compiler optimizations. The authors propose a type system which is practical and provably complete for a simple XML query language, the first of its kind. Along the way, they revisit some key assumptions used as the basis for most type systems of modern functional language.

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; 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 (Boag *et al.*, 2003; Milo *et al.*, 2000; Alon *et al.*, 2001b).

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 (Guerra *et al.*, 2005). In our paper (Colazzo *et al.*, 2002), 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. In this paper we formalize that intuition through a type system, and we prove the soundness and completeness of our construction. A significant subset of the results we prove here has been already announced in the conference version of this paper (Colazzo *et al.*, 2004), but part of the formal construction, and all the proofs, were missing.

Concurrently with our investigations (starting from the August 2003 Working Draft), the W3C XML Query Working Group extended the type system of XQuery[1] by stating that it is a static error for any expression other than the empty-sequence expression to have the empty type (Draper *et al.*, 2003). However, this is quite different from our definition of static error; unlike the W3C, we start by defining which error we are trying to prevent (Section 3), in terms of dynamic query semantics, and then we define a corresponding type system.

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 (Buneman *et al.*, 1995; Abiteboul *et al.*, 1997; Fernandez *et al.*, 1997; Boag *et al.*, 2003). μXQ is *not* intended as a model for full XQuery, since many essential XQuery features are left out. We believe the techniques we present here could be extended to full XQuery, but we will only discuss how to extend μXQ with a *where* clause, because it is the most important feature that μXQ is missing, and because its addition has important consequences on the properties of the system.

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* 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 switch has deep consequences, which we will discuss in the paper.

Once we have defined the errors, we define the type rules and the type system aimed to prevent them. The type system for path correctness is based on a couple of original technical tools, the collections of *locations* of wrong subqueries (Section 4.1) and *type-splitting* (Section 5). We prove that, at the price of a mild restriction on the use of recursion, this type system infers types that provide both an upper and, in some sense, a lower bound for the actual results returned by a query, and captures all and only the navigation-errors in the query (soundness and completeness). Also, although designed to deal with an existential notion of correctness, the defined type system can be used to check universal notions of correctness at the same time.

We prove that our type system is sound. Soundness is all what is usually proved about a type system, but soundness alone does not discriminate an interesting type-system from one that is not very precise, such as one that conservatively rejects

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

every single program. Proving both soundness *and* completeness, hence showing that the type system flags *all* and *only* the pieces of wrong code, would be optimal, but is usually unattainable, because the fact that a piece of code is "wrong" is undecidable on any realistic language. However, we have been able to isolate a rather expressive kernel language, μXQ, where we can prove our type system to be both sound and complete. Of course, completeness is lost when the language is generalized to a realistic one. This is not a limitation, since one cannot aim to completeness over a full-fledged language. Our objective was to extend the usual "soundness only" results to "soundness everywhere and completeness on a significant kernel", and we have been able to fulfill it, and for a quite large kernel.

A *where* clause is the most obvious feature that μXQ is missing. Our type system can be soundly extended to a wider language fragment comprising *where* clauses; only a weaker form of completeness holds for this extension.

Paper outline The paper is structured as follows. In Sections 2 and 3 we describe the μXQ query language and discuss our notion of path correctness for μXQ queries. Section 4 describes a first version of the type system, which we prove to be sound. Section 5 refines the type system with the notion of *type splitting*, used to make error detection and type inference more precise. We prove that this refined type system is complete. In Section 6 we study the cost of our approach. In Section 7 we discuss some possible extensions. In Section 8 we compare our approach with other proposals, and, in particular, with the XQuery type system. Section 9 concludes the paper.

Accompanying material Complete proofs of the main theorems can be found on the web, following the link *supplementary material* from the on-line abstract of this paper, in the journal site.

2 μXQ

μXQ is a minimal query language manipulating ordered forests. The language was specifically designed as a minimal setting where query correctness for XML query languages could be studied. Thus, μXQ mirrors the minimal core of XQuery-like languages, but drops features such as the *where* clause, node identity, recursive functions (and many others). In Section 7 we shall discuss the effect of adding some of these features to our language.

2.1 The data model

μXQ term and query grammar is shown below. There, f and t denote respectively ordered forests and ordered trees, and l ranges over a set of tag names L (a tree $l[f]$ represents an XML item tagged by $\langle l \rangle$,whose content is f). Furthermore, b denotes a leaf value, which we assume ranging over `String`, the set of all possible string values (essentially corresponding to PCDATA values in the DTD jargon). The ',' operator denotes ordered forest concatenation, and () is the empty forest.

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

In the data model forest concatenation is associative, and the empty forest is its neutral element, hence the equations of Table 2.1 hold.

Table 2.1. *Data model equations*

$$(f_1, f_2), f_3 = f_1, (f_2, f_3)$$
$$f, () = (), f = f$$

Table 2.2. *Data model accessors*

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

On trees, we define the accessor $childr(t)$ which returns the ordered list of all children of t. On forests, we also define the accessor $dos(t_1, \ldots, t_n)$, which returns the ordered list of all descendants of t_i's roots, including t_i's roots themselves (*dos* stands for *descendant-or-self*).

We also define a filter operator $f :: NodeTest$, which returns every tree at the top level of f that satisfies the test *NodeTest*. A test *NodeTest* is either l, selecting all trees with an l label, $node()$, selecting all trees, or $text()$, which only selects base values b.

2.2 Type language and type environments

We adopt, essentially, XDuce's type language (Hosoya & Pierce, 2003). This differs from XQuery type system since the latter is based on named typing (Siméon & Wadler, 2003). We choose a pure structural approach since it makes the formal treatment slightly more elegant, and because the structural approach constitutes the foundation of the named approach. We believe that the difference is not essential in this context, since the two approaches mainly differ on subtyping, and we do not deal with subtyping here.

Types and type environments are defined as follows:

$$\begin{array}{llcll}
\textbf{Types} & T & ::= & () & \textit{empty forest type} \\
 & & \mid & B & \textit{base type} \\
 & & \mid & l[T] & \textit{element type} \\
 & & \mid & T, T & \textit{product type} \\
 & & \mid & T \mid T & \textit{union type} \\
 & & \mid & T* & \textit{repetition type} \\
 & & \mid & X & \textit{type variable} \\
\textbf{Environments} & E & ::= & () & \\
 & & \mid & X = T,\ E &
\end{array}$$

Table 2.3. *Label-Star-Variable Chains "e" and the E-reachability relation "$T \rightarrow_e^E U$"*

Label-Star-Variable Chains

$$\begin{array}{lcl} e & ::= & \epsilon \\ & \mid & l.e \\ & \mid & *.e \\ & \mid & X.e \end{array} \qquad \begin{array}{c} (e.e').e'' = e.(e'.e'') \\ \\ e.\epsilon = \epsilon.e = e \end{array}$$

***E*-reachability**

$$\begin{array}{lll} l[T] \rightarrow_l^E T & U, T \rightarrow_\epsilon^E U & U, T \rightarrow_\epsilon^E T \\ T* \rightarrow_*^E T & U \mid T \rightarrow_\epsilon^E U & U \mid T \rightarrow_\epsilon^E T \end{array}$$

$$(X = T \in E) \Rightarrow X \rightarrow_X^E T \qquad (T \rightarrow_e^E A \ \wedge\ A \rightarrow_{e'}^E U) \Rightarrow T \rightarrow_{e.e'}^E U$$

B is the base type of all `String` values. 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.

The following definition introduces *well-formedness* of types.

Definition 2.1 ($E \vdash T$ *Def*)
We say that a type T is well-formed in an environment E, and denote it as $E \vdash T$ Def, if every variable in T is defined in E.

We restrict to l-guarded type environments, which are environments where only l-guarded vertical recursion is allowed (Definition 2.2). For example, we forbid equations such as $X = X \mid ()$ and $X = X, Y$, but allow equations such as $X = l[X \mid ()]$.

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 (Lee *et al.*, 2000; Comon *et al.*, 1997), hence expressive enough to capture the main type mechanisms of DTD and XML Schema (Siméon & Wadler, 2003; Lee *et al.*, 2000; Yergeau *et al.*, 2004; Thompson *et al.*, 2002).

To enforce this restriction, we require every definition of a variable X to be connected to every use of the same variable by a 'chain' of operators, one of which has to be an element type constructor $l[\_]$. This is formalized by means of the relation $T \rightarrow_e^E U$ defined in Table 2.3. For example, if E is $X = (m[U])*, V$, then $l[X] \rightarrow_{l.X.*.m}^E U$ holds, which means that we can reach U from $l[X]$ by crossing l, expanding X, and crossing $*$ and m (observe that $(T, U) \mid V \rightarrow_\epsilon^E U$: we do not track sequencing and union). Tracking of type names X and $*$ will be exploited later, while characterizing schemas for which we provide complete type analysis.

Definition 2.2 (*l-guarded Environments*)
E is l-guarded if $E \vdash T$ Def for each $X = T \in E$, and, for each chain e:

$$X \rightarrow_e^E X \ \Rightarrow\ \exists\, l \in L :\ e = e'.l.e''$$

The rules of our type system unfold recursive types until a tree type is met, hence l-guardedness of environments is essential to guarantee termination of these rules.

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 (Tarski, 1955)).

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

We can now define well-formedness of environments.

Definition 2.3 (*Well-Formed Environments*)
E is well-formed if it is l-guarded and, for each X defined in E, its semantics is not empty: $[\![X]\!]_E \neq \emptyset$.

In well-formed environments, empty types are disallowed; for example, we do not allow empty definitions like $X = l[X]$. The non-emptiness condition is not essential, but simplifies the type rules.

Checking type emptiness is easy. For any E, we have $[\![X]\!]_E = \emptyset$ if and only if $empty(X)_E$, where $empty(T)_E$ is the greatest function (assuming `false` < `true`) that satisfies the following set of equations. $empty(X)_E$ can be evaluated for all X's defined in E in polynomial time, using the standard algorithm: assign `true` to $empty(X)_E$, for every X, scan all definition in E to see whether some X can be actually assigned `false`, repeat the scan until the solution stabilizes (cleverer algorithms are actually known).

$$\begin{array}{lcl}
empty(())_E & = & \texttt{false} \\
empty(B)_E & = & \texttt{false} \\
empty(l[T])_E & = & empty(T)_E \\
empty(T, T')_E & = & empty(T)_E \vee empty(T')_E \\
empty(T \mid U)_E & = & empty(T)_E \wedge empty(U)_E \\
empty(T*)_E & = & \texttt{false} \\
empty(X)_E & = & empty(E(X))_E
\end{array}$$

2.3 Query language

A typical μXQ query, as shown below, 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.

$$\begin{array}{llcl}
\textbf{Queries} & Q & ::= & () \mid b \mid l[Q] \mid Q, Q \mid \bar{x} \mid x \\
& & & \mid\ \bar{x}\ \texttt{child} :: NodeTest \mid \bar{x}\ \texttt{dos} :: NodeTest \\
& & & \mid\ \texttt{for}\ \bar{x}\ \texttt{in}\ Q\ \texttt{return}\ Q \\
& & & \mid\ \texttt{let}\ x ::= Q\ \texttt{return}\ Q \\
& NodeTest & ::= & \texttt{l} \mid \texttt{node()} \mid \texttt{text()}
\end{array}$$

Table 2.4. *μXQ semantics*

$[\![b]\!]_\rho$	$\triangleq$	b	$[\![x]\!]_\rho$	$\triangleq$	$\rho(x)$
$[\![\overline{x}]\!]_\rho$	$\triangleq$	$\rho(\overline{x})$	$[\![()]\!]_\rho$	$\triangleq$	$()$
$[\![Q_1, Q_2]\!]_\rho$	$\triangleq$	$[\![Q_1]\!]_\rho, [\![Q_2]\!]_\rho$	$[\![l[Q]]\!]_\rho$	$\triangleq$	$l[[\![Q]\!]_\rho]$
$[\![\overline{x}\ \texttt{child} :: \mathit{NodeTest}]\!]_\rho$	$\triangleq$	$\mathit{childr}([\![\overline{x}]\!]_\rho) :: \mathit{NodeTest}$			
$[\![\overline{x}\ \texttt{dos} :: \mathit{NodeTest}]\!]_\rho$	$\triangleq$	$\mathit{dos}([\![\overline{x}]\!]_\rho) :: \mathit{NodeTest}$			
$[\![\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2]\!]_\rho$	$\triangleq$	$[\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho}$			
$[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2]\!]_\rho$	$\triangleq$	$\prod_{t \in \mathit{trees}([\![Q_1]\!]_\rho)} [\![Q_2]\!]_{\rho, \overline{x} \mapsto t}$			

This distinction between *let* and *for* variable simplifies the formal treatment, since it allows us to syntactically ensure that _ `child` :: *NodeTest* and _ `dos` :: *NodeTest* are always applied to a tree, but is not crucial to our approach.

We use the same notation for data model instances and language terms. In this way, the notation is kept lighter, and will always be disambiguated by the context, as in the sentence "the term $l[b], ()$ denotes the tree $l[b]$".

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

$$Q\ /\ l \triangleq \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{return}\ (\overline{x}\ \texttt{child} :: l)$$
$$Q\ //\ l \triangleq \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{return}\ \texttt{for}\ \overline{y}\ \texttt{in}\ (\overline{x}\ \texttt{dos} :: \mathit{node}())\ \texttt{return}\ (\overline{y}\ \texttt{child} :: l)$$

The semantics $[\![Q]\!]_\rho$ of a query Q w.r.t. a substitution ρ is defined in Table 2.4.

The valuation ρ maps every free `for`-variable $\overline{x}$ into a tree, and every free `let`-variable x into a forest. A binder $[\![\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2]\!]_\rho$ evaluates Q_2 in ρ extended with the binding $x \mapsto [\![Q_1]\!]_\rho$. The iterator `for` $\overline{x}$ `in` Q_1 `return` Q_2 evaluates Q_2 once for each tree t in $[\![Q_1]\!]_\rho$, and combines the result using forest concatenation '_,_'. In detail, $\mathit{trees}(f)$ returns the sequence of trees at the top level of f, and $\prod_{t \in \mathit{trees}(f)} A(t)$ is defined as the forest $A(t_1), \ldots, A(t_n)$, where $f = t_1, \ldots, t_n$ (hence is $()$ when $f = ()$). $\mathit{childr}(t)$, $\mathit{dos}(t)$, and $f :: \mathit{NodeTest}$, are defined in Table 2.2.

2.4 Locations and subqueries

In the sequel, we will need the operation $(Q)_{|\beta}$, which uses a *location* β to identify a subquery of Q. 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 2.4 ($(Q)_{|\beta}$)
$(Q)_{|\beta}$ denotes the subterm of the query Q located by the location β, which is a sequence of 0's and 1's:

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

We also define $Locs(Q)$ as the set of meaningful locations for a query Q: $Locs(Q) = \{\beta \mid (Q)_{|\beta} \neq \perp\}$.

3 Query correctness

3.1 Motivating example

XQuery definition states that a subquery is wrong when *its type* is empty but the query is different from the empty query () (Draper *et al.*, 2003). We have first to explain why we cannot just adopt this as the definition of navigation-incorrectness.

If a type system is used to identify a class of errors, the error must be defined first, in terms of dynamic semantics (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. On the contrary, XQuery definition depends on the type rules; such a dependency makes it impossible to discuss the relationship between semantic errors and errors as caught by the type rules. 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.

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 `$contacts` and `$mcontacts` (we use here \$ to identify variables) with types:

$$\begin{array}{l}\texttt{\$contacts} : (\texttt{data}[\texttt{phone}[\ldots] \mid \texttt{mobile}[\ldots]])+ \\ \texttt{\$mcontacts} : (\texttt{data}[\texttt{mobile}[\ldots]])+\end{array}$$

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

$$\begin{array}{l}Q_0 \equiv \texttt{\$mcontacts/phone} \\ Q_1 \equiv \texttt{\$contacts/fone} \\ Q_2 \equiv \texttt{\$contacts/phone}, \texttt{\$contacts/mobile} \\ Q_3 \equiv \texttt{\$contacts/phone} \\ Q_4 \equiv \texttt{\$contacts/fone}, \texttt{\$contacts/mobile} \\ Q_5 \equiv \texttt{for \$c in \$contacts} \\ \quad\quad \texttt{return (\$c/phone, \$c/mobile)} \\ Q_6 \equiv \texttt{for \$c in (\$contacts, \$mcontacts)} \\ \quad\quad \texttt{return (\$c/phone, \$c/mobile)}\end{array}$$

Q_0 and Q_1 are wrong, since they cannot match the data, while Q_2 is correct, since 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 (Colazzo *et al.*, 2002), under the name 'weak correctness'.

Q_4 is troublesome. It is clearly wrong, since the first path cannot match the data. However, although a subquery never matches any data, the whole query can return a non-empty result, hence the whole query *has* the ability to return some data, 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 if and only if the whole program returns 'error'. Here, instead, we are forced to explicitly define correctness of a query as the lack of problems in the query *and* in each of its subqueries. We hence arrive at the following notion of correctness (where non-() means 'syntactically different from ()'):

Definition 3.1 (*Foreach-Exist (FE) Query Correctness – informal definition*)
A query Q is correct w.r.t. a set of valid substitutions $\mathscr{R}$ if, *for each* non-() subquery Q' in Q, *there exists* $\rho \in \mathscr{R}$ 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 `$mcontacts`) that makes `$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 3.2 (*Exist-Foreach (EF) Query Correctness*)
A query Q is correct w.r.t. a set of valid substitutions $\mathscr{R}$ if *there exists* $\rho \in \mathscr{R}$ 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, informally, that our notion is 'maximally strict', in the design space that we explored.

We have now to show that our notion is arguably not *too* strict, i.e. that 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 $\rho \in \mathscr{R}$, 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.

3.2 FE-correctness

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 single substitution since each subquery in the scope of a `for` $\bar{x}$ `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 3.3 (*Substitution Extension*)

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

We now define the set $CriticalLocs(Q)$ of the locations of Q where we will look for pieces of wrong code. According to what stated in previous sections, a first definition could be $CriticalLocs(Q) \triangleq \{\beta \mid (Q)_{|\beta} \neq ()\}$, as when $(Q)_{|\beta}$ is (), β must not be tested for non-emptiness. However, we can refine the definition of $CriticalLocs(Q)$ by observing 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. Hence, we also exclude `let` subqueries from $CriticalLocs(Q)$. 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. A complete analysis shows that only ($\bar{x}$ `child` :: *NodeTest*) and ($\bar{x}$ `dos` :: *NodeTest*) subqueries, and the first argument of a `for` iteration need be considered.

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

We can now make Definition 3.1 completely formal, as follows.

Definition 3.4 (*Correctness of Q w.r.t.* $\mathscr{R}$)
Let $\mathscr{R}$ be a set of substitutions for the free variables of a query Q. Q is FE-correct w.r.t. $\mathscr{R}$ if and only if:

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

Dually, Q has an error at the location $\beta \in CriticalLocs(Q)$ if and only if:

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

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

3.3 Type-checking existential and universal correctness

Our notion of navigation-correctness is existential, while the traditional notions of correctness are universally quantified. We needed some time to identify this quantification switch, to appreciate its consequences, and to understand the boundaries of the design space that it opens up. We report in this section what we learned, which can be synthesized as follows.

- A piece of code may be considered correct if it always runs with no run-time problem (*universal correctness*) or if it sometimes runs with no run-time problem (*existential correctness*).
- A type-checker may be *overflagging*—if a piece of code is not correct then it will flag it—or *underflagging*—if it flags a piece of code then the code is not correct.
- A type-checker may, in principle, infer types that are *upper-bounds* or that are *lower-bounds* for the actual set of values that may be returned by an expression.
- Traditional *conservative* type-checkers perform an overflagging type-checking of a universal notion of correctness, hence ensuring that "well-typed pieces of code will never go wrong". To this aim, they need to infer upper-bounds.
- We will propose here a *non-intrusive* type-checker that performs an underflagging type-checking of an existential notion of correctness. This combination is still based on upper-bounds. This means that one can combine in the same upper-bounds based type-system the overflagging type-checking of a universal notion of correctness with the underflagging type-checking of an existential notion of correctness.

Let us start again.

We have first to distinguish between run-time problems and code-problems. For example, we may consider the execution of a division by zero as a run-time problem of interest. Then, we may either focus on the code-problem presented by those pieces of code that perform a division by zero for any possible computational context (in our case, for any possible value of their free variables), or we may focus on the code-problem presented by those pieces of code for which a computational context exists such that the code will perform a division by zero. Usually, one says that the code has a problem if *there exists* a context which will generate the run-time problem, but in the previous section we have seen that the opposite choice may make sense. Of course, when the definition of code-error is existentially quantified (exists a context where the code meets a run-time error), the definition of code-correctness is universally quantified (for every context the code meets no run-time error). In this case we talk about "universal notion of correctness", whose violations we call "universal errors". Every type system we know adopts this notion of code-error.

When a type system is added to the picture, a second choice is presented. If type-checking is static and decidable, it is usually impossible to flag as type-errors all and only the pieces of code with a code-problem. Hence, one has to choose between an overflagging and an underflagging type system. An overflagging type system guarantees that if a piece of code has a code-problem then it will be considered type-wrong. An underflagging type system guarantees that if a piece of code is considered type-wrong then it has a code-problem. Overflagging and underflagging are not one the negation of the other; they are dual properties which are difficult to combine, in the same way as soundness and completeness are.

Traditional type-systems are overflagging. Overflagging type-checking of a universal notion of correctness implies that, if a piece of code is type-correct, then it is code-correct *and* if it is code-correct then it will never have run-time problems. Hence, with such a strict combination, well-typed terms never go wrong. Overflagging is the natural property to look for when a type system is aimed at a universal notion of correctness.

We must add that, when studying an overflagging type-system for a universal correctness, one does not really need going through the three notions of run-time problem, code-error, and type-error. One may skip the middle step, and directly define soundness as the fact that a type-correct piece of code will never meet a run-time problem.

When correctness is existential, the choice between an overflagging and an underflagging type system is slightly less obvious. However, in the case we are studying here, we believe that being underflagging is the essential virtue. Type-checking is optional in XQuery. Whoever used *lint*[2] knows that, if an optional error-checking tool gives some false alarms, programmers tend to ignore its alarms altogether. Moreover, ignoring a piece of dead code seems less of a problem than forcing a programmer to rewrite a piece of good code because of a false alarm.

A type system usually infers a type for any expression, and uses the inferred type in the error-checking process; to simplify the discussion, we will identify a type with a set of values. The type inferred for an expression approximates the set of the values that may be returned by that expression. More precisely, the type may be guaranteed to contain the set of all possible values, or to be contained in such set; we talk of *upper-bound* approximation in the first case, and *lower-bound* in the second.

Every type-system we are aware of takes the upper-bound approach, or aims at computing both upper and lower-bound, as we do here. The upper-bound approach is preferred for many reasons, the most important one being that types as upper-bounds go well together with overflagging checking of universal correctness. If the type is an upper-bound, a type sentence $n : \{1, 2\}$ allows a type checker to deduce that $1/n$ will never raise a divide-by-zero error, which is the information needed to perform overflagging type-checking of the universal correctness condition "never divide by zero". However, underflagging verification of universal correctness needs lower-bounds. Assume that $n : T$. To prove that $1/n$ is *not* universally correct we must prove that 0 is among the values that can be assumed by Q. This is implied by $0 \in T$ only if T is a lower-bound. In the same way, lower-bounds are needed for the overflagging checking of existential correctness. The query Q/b is FE-correct if Q can actually evaluate to a term $l[f, b[f'], f'']$, e.g., if a lower bound of its set of values contains $\{l[b[]]\}$. An upper-bound of the set of values of Q cannot guarantee that Q may actually assume a value with shape $l[f, b[f'], f'']$, hence cannot guarantee the absence of an existential error, hence is useless for overflagging type-checking. However, an upper-bound *without* a value $l[f, b[f'], f'']$ guarantees the presence of an existential error, hence upper-bounds allow underflagging type-checking in the existential case, and this is really what we want here.

To sum up, upper-bound type-inference is good for overflagging type-checking of universal correctness and for underflagging type-checking of existential correctness,

[2] *lint* is a tool that helps a C programmer locating *potential* errors in the code.

but is useless for the other two combinations. Dually, lower-bound type-inference is good for overflagging type-checking of existential correctness and for underflagging type-checking of universal correctness, but is useless for the other two combinations. These are good news. Indeed, we want to study a type-system that checks both existential and universal notions of correctness, because this is needed to understand XQuery type systems, and also because this is really needed in any concrete XML query language. Now we know that this combination is possible. Hereafter, we will use *sound* error-checking as an abbreviation for “overflagging checking of universal correctness and underflagging checking of existential errors”. We will use *complete* error-checking for the dual combination of “underflagging checking of universal correctness and overflagging checking of existential errors”. So our slogan now is: upper-bounds for soundness, lower-bounds for completeness.

4 Type-checking

4.1 Judgements

To type-check a query we need type information about its free variables. The type assignments for the free variables of a query are defined by means of *variable environments* Γ of the form:

$$\begin{array}{llll} \textbf{Variable Environments} & \Gamma & ::= & () \\ & & \mid & x : T, \Gamma \\ & & \mid & \overline{x} : T, \Gamma \end{array}$$

Definition 4.1 (*Well-Formed* Γ)
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).[3]

Our type rules are based on judgments of the form:

$$\begin{array}{llll} \textbf{Judgments} & J & ::= & E;\, \Gamma \vdash_\beta Q : (T;\, \mathscr{S}) \\ & & \mid & E;\, \Gamma \vdash_\beta \overline{x}\ \texttt{in}\ T \rightarrow Q : (T;\, \mathscr{S}) \end{array}$$

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

The definition of well-formed judgments is standard, and requires that every variable is defined once and only once. We use $FV(Q)$ and $DV(Q)$ to denote the variables that are, respectively, free and defined in Q. We use $def(\Gamma)$ to denote the set of all variables defined in Γ.

Definition 4.2 (*Well-Formed Judgments*)
We say that the judgment $J \equiv E;\, \Gamma \vdash_\beta Q : (T;\, \mathscr{S})$ (or $J \equiv (E;\, \Gamma \vdash_\beta \overline{x}\ \texttt{in}\ T \rightarrow Q : (U;\, \mathscr{S}))$) is well-formed, and write $WF(J)$, if and only if:

- E is well-formed; Γ is well-formed in E; T is well-formed in E;

[3] Unions of tree types, such as $l[T] \mid l'[T']$, could be allowed, but this would make no difference, since the rules for case-analysis are such that a for-variables is introduced in the environment only when it is bound to a tree type.

- If $J \equiv (E;\ \Gamma \vdash_{\beta}\ Q : (T;\ \mathscr{S}))$, then $FV(Q) \subseteq def(\Gamma)$, $DV(Q) \cap def(\Gamma) = \emptyset$ and Q is well-formed;
- If $J \equiv (E;\ \Gamma \vdash_{\beta}\ \overline{x}\ \mathtt{in}\ T \rightarrow Q : (U;\ \mathscr{S}))$ then $FV(Q) \subseteq (def(\Gamma) \cup \{\overline{x}\})$, $DV(Q) \cap (def(\Gamma) \cup \{\overline{x}\}) = \emptyset$ and Q is well-formed.

The type rules of Section 4.2 are written so that they can only be used to prove well-formed judgments. To this aim, whenever well-formedness of the conclusion is not an immediate consequence of well-formedness of the premises, an explicit premise $WF(J)$ is added. This is a standard technique which makes proofs slightly easier; $WF(J)$ premises can be safely ignored while reading the rules.

The judgment $E;\ \Gamma \vdash_{\beta}\ \overline{x}\ \mathtt{in}\ T \rightarrow Q : (T;\ \mathscr{S})$ is used to type-check for-iteration. Recall that a query $\mathtt{for}\ \overline{x}\ \mathtt{in}\ Q_1\ \mathtt{return}\ Q_2$ evaluates Q_2 once for every tree in the forest computed by Q_1, by binding such tree to $\overline{x}$. Accordingly, to analyze $\mathtt{for}\ \overline{x}\ \mathtt{in}\ Q_1\ \mathtt{return}\ Q_2$, we compute a type T_1 for Q_1 and use the auxiliary judgment $E;\ \Gamma \vdash_{\beta.1}\ \overline{x}\ \mathtt{in}\ T_1 \rightarrow Q_2 : (T_2;\ \_)$ to compute the type of Q_2 through a case-analysis on the type T_1. So, for example, if $T_1 \equiv T \mid U$ (or $T_1 \equiv T, U$), we compute $E;\ \Gamma \vdash_{\beta.1}\ \overline{x}\ \mathtt{in}\ T \rightarrow Q_2 : (T';\ \_)$ and $E;\ \Gamma \vdash_{\beta.1}\ \overline{x}\ \mathtt{in}\ U \rightarrow Q_2 : (U';\ \_)$ and then $T_2 \equiv T' \mid U'$ (resp. $T_2 \equiv T', U'$). This process terminates when a tree type T_{tree} is met, and typing proceeds with $E;\ \Gamma, \overline{x} : T_{tree} \vdash_{\beta.1}\ Q_2 : (\_;\ \_)$; as a consequence, thanks to l-guardedness of E, this process always terminates. As we will see, although expensive, case-analysis is essential for type-inference precision, hence to obtain a complete type system.

As we have seen, our typing judgments J also return an error set $\mathscr{S}$, and have a parameter β. The set $\mathscr{S}$ contains a set of locations with shape $\beta.\alpha$, such that, if $J \equiv E;\ \Gamma \vdash_{\beta}\ Q : (T;\ \mathscr{S})$ or $J \equiv E;\ \Gamma \vdash_{\beta}\ \overline{x}\ \mathtt{in}\ T \rightarrow Q : (U;\ \mathscr{S})$, the subquery of Q at α is not FE-correct. The location β specifies the position of the currently-checked subquery inside the query where type-checking started with $\beta = \epsilon$.

The collection of an error set 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 in order to check an existential notion of correctness. Consider, for example, the following queries over $\mathtt{\$contacts} : (\mathtt{data}[\mathtt{phone}[...]] \mid \mathtt{data}[\mathtt{mobile}[...]])+$.

$$Q_5 \equiv \mathtt{for\ \ \$c\ in\ \$contacts\ \ return\ \ (\$c/phone, \$c/mobile)}$$
$$Q_7 \equiv \mathtt{for\ \ \$c\ in\ \$contacts\ \ return\ \ (\$c/fone, \$c/mobile)}$$

We perform type-checking by case-analysis, which means that the computation of the errors of Q_7, by an error-checking function $\mathrm{Err}_{\$c:(T_1|T_2)}(Q_7)$, combines the results of analyzing Q_7 for $\$c : T_1$ and $\$c : T_2$ (where $T_1 = \mathtt{data}[\mathtt{phone}[...]]$ and $T_2 = \mathtt{data}[\mathtt{mobile}[...]]$) with some operation Op^1:

$$\mathrm{Err}_{\$c:(T_1|T_2)}(Q_7) = Op^1_{T \in \{T_1, T_2\}}(\mathrm{Err}_{\$c:T}(Q_7))$$

To check Q_7 for $T \in \{T_1, T_2\}$, we check the two subqueries and combine the results with some Op^2, hence:

$$\mathrm{Err}_{\$c:(T_1|T_2)}(Q_7) = Op^1_{T \in \{T_1, T_2\}}(\mathrm{Err}_{\$c:T}(\$c/fone)\ Op^2\ \mathrm{Err}_{\$c:T}(\$c/mobile))$$

By Definition 3.4, a query (Q, Q') is FE-incorrect iff either Q or Q' is, hence Op^2 should be a disjunction. Substitutions are universally quantified in the definition of FE-errors, hence Op^1 should be a conjunction. Hence, a type-checking algorithm based on case-analysis should 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, this function flags Q_7 as 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 conjunction below 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 just reflect the quantifications of Definition 3.4. However, we can generalize booleans to sets of locations, and use the following definition, where $\mathrm{ErrLoc}(Q)$ returns the locations associated to wrong subqueries of Q.

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

This time $\mathrm{ErrLoc}(Q_5)$ 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, $\mathrm{ErrLoc}(Q_7)$ 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. Hence, sets of error-locations seem to be the right generalization of booleans to be adopted to perform case-analysis-based type-checking of an existential notion of correctness.

4.2 Type inference and error checking

The type rules are listed in Tables 4.1, 4.2 and 4.3. Tables 4.1 and 4.2 show the rules required to infer a query result type and errors; rules for case-analysis are collected in Table 4.2.

Table 4.3 illustrates a set of rules required to *filter* a type according to a given condition *NodeTest*. The rules are needed to type the XQuery-like operators `dos` and `child` in (TypeChild) and (TypeDos). Finally, note that rules (TypeInElSplitting) and (TypeLetSplitting) use the function $\mathit{Split}_E(T)$. These rules are sound for any function $\mathit{Split}_E(\_)$ such that $\mathit{Split}_E(T) = \{T_1,\ldots,T_n\} \Rightarrow [\![T]\!]_E = [\![T_1 \mid \ldots \mid T_n]\!]_E$. In this Section we simply define $\mathit{Split}_E(T) = \{T\}$; in Section 5 we will adopt a function that performs a finer splitting, in order to compute more precise types.

Rule (TypeFor) starts the case-analysis, as previously discussed, propagates the error sets $\mathscr{S}_1$ and $\mathscr{S}_2$, and adds an error $\beta.0$ if the type of Q_1 only contains the empty forest. It uses the auxiliary judgment $T \sim_E ()$, which checks whether $[\![T]\!]_E = [\![()]\!]_E$ (see Definition 4.4).[4]

[4] 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: type inference*

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

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

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

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

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

(TypeFor)
$$\frac{\begin{array}{l} E;\,\Gamma \vdash_{\beta.0} Q_1 : (T_1;\, \mathcal{S}_1) \\ E;\,\Gamma \vdash_{\beta.1} \overline{x}\ \texttt{in}\ T_1 \rightarrow Q_2 : (T_2;\, \mathcal{S}_2) \qquad \mathcal{S} = \text{ if } T_1 \sim_E () \text{ then } \{\beta.0\} \text{ else } \emptyset \end{array}}{E;\,\Gamma \vdash_{\beta} \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : (T_2;\, \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S})}$$

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

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

Rule (TypeInUnion) performs case-analsys. When $\overline{x}$ is associated with a union type ($\overline{x}$ `in` $T_1 \mid T_2 \rightarrow Q$), the rule first infers two pairs $(T'_1;\, \mathcal{S}'_1)$ and $(T'_2;\, \mathcal{S}'_2)$ for Q, by analyzing $\overline{x}$ `in` $T_1 \rightarrow Q$ and $\overline{x}$ `in` $T_2 \rightarrow Q$, respectively. The final pair inferred for Q is then $(T'_1 \mid T'_2;\, \mathcal{S}'_1 \cap \mathcal{S}'_2)$, with $\mathcal{S}'_1 \cap \mathcal{S}'_2$ only containing the locations that are wrong in both branches. Rule (TypeInConc) follows the same approach.

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

Type unfolding performed by the rules for case-analysis stops when the empty-forest type $()$ or a tree type ($l[T]$ or B) is met (rules (TypeInAtomic) and

Table 4.2. *Query Type Rules: type inference by case analysis*

(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)

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

(TypeInVar)

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

(TypeInStar)

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

(TypeInAtomic)

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

(TypeInConc)

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

(TypeInUnion)

$$\frac{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1 \rightarrow Q : (T'_1;\ \mathscr{S}_1) \qquad E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_2 \rightarrow Q : (T'_2;\ \mathscr{S}_2)}{E;\ \Gamma \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1 \mid T_2 \rightarrow Q : (T'_1 \mid T'_2;\ \mathscr{S}_1 \cap \mathscr{S}_2)}$$

(TypeInElSplitting)). As already stated, termination of this unfolding is guaranteed by l-guardedness of E, which guarantees that only a finite number of name expansions can be applied by the type rule (TypeInVar).

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

In the premises of rules (TypeChild) and (TypeDos), the axis argument $\overline{x}$ is required to have a tree type. Note that this condition holds when the judgment is well-formed (see Definition 4.2).

In rule (TypeChild) the type for $\overline{x}$ `child` :: *NodeTest* is computed by filtering, according to *NodeTest*, the content of the tree type T, associated to $\overline{x}$ in Γ. Filtering is performed by the auxiliary judgment $E \vdash\ T' :: NodeTest \Rightarrow U$ (see Table 4.3 for definition).

In rule (TypeDos) the type U of $\overline{x}$ `dos` :: *NodeTest* is inferred by first computing the type U' of the descendant-or-self nodes derivable from the values in T, where T is the tree type associated to $\overline{x}$ in Γ. U' is generated as the $*$-guarded union $(U_1 \mid \ldots \mid U_n)*$ of all types that are reachable from T, which are returned by the function $SubTrees_E(T)$ (see Definition 4.3). Finally, U is the result of filtering U' according to *NodeTest*.

Both rules (TypeChild) and (TypeDos) insert an error location β in $\mathscr{S}$ if and only if the restricted type U is equivalent to the type (). Observe that applying `child` to a `for` variable assigned to a base type always returns an error.

Table 4.3. *Filter Type Rules*

(BaseNodeFilt) $\dfrac{}{E \vdash\ B :: \texttt{node()} \Rightarrow B}$	(BaseTextFilt) $\dfrac{}{E \vdash\ B :: \texttt{text()} \Rightarrow B}$	(BaseLabFilt) $\dfrac{}{E \vdash\ B :: l \Rightarrow ()}$
(TreeNodeFilt) $\dfrac{}{E \vdash\ l[T] :: \texttt{node()} \Rightarrow l[T]}$	(TreeTextFilt) $\dfrac{}{E \vdash\ l[T] :: \texttt{text()} \Rightarrow ()}$	
(TreeMatchLabFilt) $\dfrac{}{E \vdash\ l[T] :: l \Rightarrow l[T]}$	(TreeNoMatchLabFilt) $\dfrac{l \neq m}{E \vdash\ m[T] :: l \Rightarrow ()}$	
(EmptyFilt) $\dfrac{}{E \vdash\ () :: NodeTest \Rightarrow ()}$	(ForestFilt) $\dfrac{E \vdash\ T :: NodeTest \Rightarrow T' \qquad E \vdash\ U :: NodeTest \Rightarrow U'}{E \vdash\ T, U :: NodeTest \Rightarrow T', U'}$	
(StarFilt) $\dfrac{E \vdash\ T :: NodeTest \Rightarrow U}{E \vdash\ T* :: NodeTest \Rightarrow U*}$	(UnionFilt) $\dfrac{E \vdash\ T :: NodeTest \Rightarrow T' \qquad E \vdash\ U :: NodeTest \Rightarrow U'}{E \vdash\ T \mid U :: NodeTest \Rightarrow T' \mid U'}$	
(VarFilt) $\dfrac{E \vdash\ E(X) :: NodeTest \Rightarrow U}{E \vdash\ X :: NodeTest \Rightarrow U}$		

Rules (TypeChild) and (TypeDos) use the auxiliary function $SubTrees_E(T)$ and the predicate $T \sim_E ()$, defined below, and the auxiliary judgment $E \vdash\ T :: NodeTest \Rightarrow U$, defined in Table 4.3.

Definition 4.3 (*Subtrees Type Extraction*)
For any E well-formed and T such that $E \vdash T$ Def, we define $SubTrees_E(T)$ as follows (well-defined by Knaster-Tarski Theorem):

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

The union of all types in $SubTrees_E(T)$ contains exactly the subtrees of every tree in U (Lemma 4.11). This entails that, if $[\![T]\!]_E = [\![U]\!]_E$, then

$$\bigcup_{T' \in SubTrees_E(T)} [\![T']\!]_E = \bigcup_{U' \in SubTrees_E(U)} [\![U']\!]_E$$

Definition 4.4 (*Empty-Forest-Type Checking*)
For any well-formed environment E and type T well-formed in E, we define $T \sim_E ()$ as the minimal function (assuming $\texttt{false} < \texttt{true}$) that respects the following set of equations, well-defined by Knaster-Tarski Theorem:

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

Correctness and completeness of this definition is proved by the following lemma, whose proof we omit.

Lemma 4.5 (*Empty-Forest-Type Checking*)
For any well-formed environment E and type T well-formed in E:

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

To type `child` $::$ *NodeTest* and `dos` $::$ *NodeTest* we use auxiliary judgments of the following form:

$$E \vdash \; T :: \mathit{NodeTest} \Rightarrow U$$

meaning that, for each f in T, the filtering $f :: \mathit{NodeTest}$ is in U. These judgments are defined in Table 4.3.

Lemma 4.6 (*Termination of Type Filtering*)
For any well-formed type environment E, and types T and U, the backward application of the type rules to $E \vdash \; T :: \mathit{NodeTest} \Rightarrow U$ terminates.

Lemma 4.7 (*Type Filtering Checking*)
For any well-formed type environment E and type T well-formed in E:

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

We conclude this section with a list of properties that will be used later on.

Definition 4.8 (*Subtree Relation*)
We say that the tree t is a subtree of f, written $t \in_{st} f$, if and only if

$$\exists f_1, f_2.\; dos(f) = f_1, t, f_2$$

Lemma 4.9
For any E well-formed and T such that $E \vdash T$ Def and for each tree t:

$$(\exists f \in [\![T]\!]_E.\; t \in_{st} f) \;\Leftrightarrow\; (\exists U.\; T \rightarrow^E_e U \;\wedge\; t \in [\![U]\!]_E \;\wedge\; (U \equiv l[T'] \vee U \equiv B))$$

Lemma 4.10
For any E well-formed and T such that $E \vdash T$ Def, and for any U:

$$T \rightarrow_e^E U \;\wedge\; (U \equiv l[T'] \vee U \equiv B) \;\Leftrightarrow\; U \in SubTrees_E(T)$$

Lemma 4.11
For any well-formed E and T such that $E \vdash T$ Def, for each tree t:

$$(\exists f \in [\![T]\!]_E.\ t \in_{st} f) \;\Leftrightarrow\; (\exists U.\ U \in SubTrees_E(T) \wedge t \in [\![U]\!]_E)$$

Lemma 4.12 (Soundness of DOS Type)
For any well-formed E and T such that $E \vdash T$ Def and

$$\begin{array}{rcl} SubTrees_E(T) & = & \{U_1, \ldots, U_n\} \\ U & \equiv & (U_1 \mid \ldots \mid U_n)* \end{array}$$

then:

$$\forall f \in [\![T]\!]_E.\ dos(f) \in [\![U]\!]_E$$

4.3 Soundness of result analysis and error-checking

We provisionally assumed that $Split_E(T) = \{T\}$, which results in a completely standard (TypeLet) rule. This is sufficient to obtain the canonical property that types are upper bounds for the set of all possible results (Theorem 4.14), which is the basis of the soundness results, namely the underflagging property for a type-checker for existential correctness and the overflagging property for a type-checker for universal correctness.

Soundness is the canonical properties that is proved for any type system, but is not very informative: any system that associates the universal type to any expression, and never finds any existential error, enjoys them as well. While soundness is usually the only property that a type-system enjoys, we will show in the next section that our system is also complete, in a sense that will be defined.

To formalize and prove soundness we need the following definition.

Definition 4.13 ($\mathscr{R}(E,\Gamma)$)
For any well-formed type environment E and Γ well-formed in E, we define the set of valid substitutions as:

$$\begin{array}{rl} \mathscr{R}(E,\Gamma) = \{\rho \mid & (\exists f.\ (\chi \mapsto f) \in \rho \;\Leftrightarrow\; \exists T.\ (\chi : T) \in \Gamma) \\ & \text{and } ((\chi \mapsto f) \in \rho \text{ and } (\chi : T) \in \Gamma \;\Rightarrow\; f \in [\![T]\!]_E)\} \end{array}$$

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

As Γ is well-formed in E well-formed, the set $\mathscr{R}(E,\Gamma)$ is never empty. Hence, for any well-formed judgment $E;\, \Gamma \vdash_\beta Q : (T;\, \mathscr{S})$, $\mathscr{R}(E,\Gamma)$ is not empty.

Theorem 4.14 (Upper Bound)
For any well-formed environment E, Γ well-formed in E, and query Q:

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

The upper-bound property allows us to prove soundness of error-checking, i.e. the fact that, whenever a piece of code is flagged as type-wrong, it really suffers of an FE-error (the *underflagging* property).

Theorem 4.15 (*Soundness of Existential Error-Checking*)
For any well-formed environment E, Γ well-formed in E, and query Q:

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

4.4 Subtyping and substitution

It is now time to cite some standard theorems that one may expect to hold, and which do not, because of our existential error-checking approach. 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 *type-specialization* and *term-substitution* properties fail for this type system.

Property 4.16 (*Type-Specialization for Overflagging Systems*)
In an overflagging type-system for a universally quantified notion of correctness, if $T' \leqslant T$ is a subtype relation such that $T' \leqslant T \Rightarrow [\![T']\!]_E \subseteq [\![T]\!]_E$, then

$$E;\ \Gamma, x : T \vdash_\epsilon\ Q : (U;\ \emptyset),\quad E;\ \Gamma \vdash_\epsilon\ Q_1 : (T';\ \emptyset),\quad \text{and}\quad T' \leqslant T$$

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

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

Property 4.17 (*Term-Substitution for Overflagging Systems*)
In an overflagging type-system for a universally quantified notion of correctness:[5]

$$E;\ \Gamma, x : T \vdash_\epsilon\ Q : (U;\ \emptyset)\ \text{ and }\ f \in [\![T]\!]_E\ \ \Rightarrow\ \ E;\ \Gamma \vdash_\epsilon\ Q\{x \leftarrow f\} : (U;\ \emptyset).$$

Type-specialization and *term-substitution* are consequences of the conservative nature of 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.

Type-specialization derives from the conservative nature of the type system as well: if no value in a type creates problems, a smaller type creates no problem a fortiori.

However, a form of term-substitution and type-specialization holds for non-intrusive type-systems as well, with the essential difference that the set of errors is increased by the act of reducing the set of the values that a variable may assume. Hence, these properties assume a structure like that of properties 4.18 and 4.19.

Property 4.18 (*Type-Specialization for Underflagging Systems*)
In an underflagging type-system for an existentially quantified notion of correctness, if $T' \leqslant T$ is a subtype relation such that $T' \leqslant T \Rightarrow [\![T']\!]_E \subseteq [\![T]\!]_E$, then

$$E;\ \Gamma, x : T \vdash_\epsilon\ Q : (U;\ \mathscr{S}),\quad E;\ \Gamma \vdash_\epsilon\ Q_1 : (T';\ \mathscr{S}'),\quad \text{and}\quad T' \leqslant T$$

implies

$$E;\ \Gamma \vdash_\epsilon\ Q\{x \leftarrow Q_1\} : (U';\ \mathscr{S}''),\quad U' \leqslant U,\quad (\mathscr{S} \cup \mathscr{S}') \subseteq \mathscr{S}''$$

[5] We use here the fact that every semantic object $f \in [\![T]\!]_E$ also belongs to the syntax. We could, instead, use a canonical injection from $[\![T]\!]_E$ into the syntax, but this seems an unnecessary complication.

Property 4.19 (*Term-Substitution for Underflagging Systems*)
In an underflagging type-system for an existentially quantified notion of correctness:

$$E;\ \Gamma, x : T \vdash_{\epsilon} Q : (U;\ \mathscr{S}) \ \text{ and } \ f \in [\![T]\!]_E$$
$$\Rightarrow \quad \exists \mathscr{S}' \supseteq \mathscr{S}.\ E;\ \Gamma \vdash_{\epsilon} Q\{x \leftarrow f\} : (U;\ \mathscr{S}'),$$

We are not going to prove these properties. This subsection was only meant to illustrate that the switch from the universal to the existential approach has some delicate consequences.

5 Type-splitting

5.1 *Motivation and example*

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, as we noticed already. For the core-language μXQ, we can actually aim for a much stronger property: a type system that infers types which are both lower-bounds and upper-bounds, hence is able to catch *all and only* the FE-errors. In a word, a type-system that is both sound and complete.

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:

$Q_8 \equiv$ `for` $\bar{\texttt{x}}$ `in y/a return y/b`

where y is of type Y (this code returns a sequence y/b if and only if 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. (Similar phenomena happen in all related type systems we are aware of, including the XQuery type system.)

5.2 *Definitions*

We will define a type system which 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 5.22).

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 5.7) 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 5.2), 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 5.27. Hence, for example:

$$\begin{aligned} Split_E((\mathtt{data[phone[...]} \mid \mathtt{mobile[...]])*}) &= \{(\mathtt{data[phone[...]} \mid \mathtt{mobile[...]])*}\} \\ Split_E(\mathtt{phone[...]} \mid \mathtt{mobile[...]}) &= \{\mathtt{phone[...]};\ \mathtt{mobile[...]}\} \\ Split_E(a[(b[],c[]) \mid (d[],e[])]) &= \{a[b[],c[]];\ a[d[],e[]]\}. \end{aligned}$$

For an intuitive understanding of this result, consider that splitting was needed for query Q_8 because of the presence of the two *mutually incompatible* paths c/a and c/b inside the type of y. For the query Q_8, an assumption like $y : (c[a[] \mid b[]])*$, where union is guarded by $*$, does not need to be split any further. The key observation is the fact that paths c/a and c/b are not incompatible when y has type $c[(a[] \mid b[])*]$, since $c[a[], b[]]$ is a legitimate value for y, and for this value, for instance, both y/a and y/b find a match and make the query yield the not empty result $c[b[]]$. Moreover, with $y : (c[a[] \mid b[]])*$ the query has type $(c[b[]])*$, which is an exact type for Q. Lemma 5.11 provides a formalization of the intuition that incompatible paths cannot exist in types where union is guarded by $*$.

The actual 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 $/a$:

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

This query yields the empty sequence under each valid substitution for y. In order to infer the empty sequence type for this query, Y must be unfolded $n+1$ times before splitting and performing separate analysis. This means that we cannot decide how deeply Y should be unfolded without looking at the query under consideration.

This example may suggest that a more complex type system, where unfolding depends on the query, may be worth studying. But this seems hard to obtain. In the following example, indeed, splitting after any finite unfolding does not solve the problem, because of the presence of `descendant-or-self`.

$$Q' \equiv \texttt{for } \bar{\texttt{x}} \texttt{ in y//a} \quad \texttt{return (for } \bar{\texttt{z}} \texttt{ in } \bar{\texttt{x}}\texttt{/a return } \bar{\texttt{x}}\texttt{/b)}$$

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

The solution we propose, instead, is based on a mild restriction on the use of recursion in type environments, which, as we will show, seems to be acceptable in practice.

We rule out types as the Y type above by requiring that any environment be $*$-guarded, which means that recursion is guarded by a $*$ type constructor,

Definition 5.1 (*$*$-guarded Environments*)
E is $*$-guarded if it is well-formed and:

$$\forall X.\ \forall e.\ \ X \rightarrow_e^E X \quad \Rightarrow \quad \exists e', e''.\ e = e'.*.e''$$

We claim that our restriction is "mild". Indeed, it is respected by *all* the schemas reported in the W3C document "XML Query Use Cases" (Chamberlin *et al.*, 2003).

Under this restriction, $Split_E(T)$ unfolds recursion until $*$ is met, and "pulls out" all and only the union type constructors that are found outside any $*$. Hence, the *Split* function (Definition 5.2) always produces a finite set of types, and induces a type-system that computes lower-bounds and is complete (Theorems 5.22 and 5.27).

Definition 5.2 ($Split_E(T)$)
If E is $*$-guarded, then:

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

$Split_E(T)$ is well-defined by Knaster-Tarski theorem. If E is $*$-guarded, $Split_E(T)$ is finite and 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 generated by the recursive definition of a type variable.

Splitting preserves type semantics.

Lemma 5.3
For each $*$-guarded environment E and type T defined in E:

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

From now on, we stop assuming $Split_E(T) = \{T\}$, and start assuming that $Split_E(T)$ is defined as shown in Definition 5.2. No type rule changes. The resulting type system is called *type-splitting system*.

5.3 Simulation and query monotonicity

To characterize the precision of the type-splitting system, we define now a pre-order relation on forests, noted as $\sqsubseteq$, and called *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 $a[], b[] \sqsubseteq b[], a[]$; and implies path inclusion, so that if a/d is a path of f and $f \sqsubseteq f'$, then a/d is a path of f' as well.

We will prove that query semantics is monotone with respect to $\sqsubseteq$, and that, if a query is correct when run with $y = f$ and $f \sqsubseteq f'$, then the query is correct when run with $y = f'$ as well; these are the only properties that $\sqsubseteq$ must enjoy. The relation $\sqsubseteq$ ignores the fact that trees are ordered, and does not distinguish among base values; this reflects the fact that our notion of error is only related to the existence of paths, which do not depend on sibling order nor on base values.

Definition 5.4 (*Forest Simulation $f \sqsubseteq f'$*)
Simulation $f \sqsubseteq f'$ is the smallest relation on forests that respects the following conditions:

$$\begin{array}{llll}
\forall b_1, b_2 \textit{ of type } B. & b_1 \sqsubseteq b_2 & & \\
\forall\, l.\ \forall f_1, f_2. & l[f_1] \sqsubseteq l[f_2] & \Leftrightarrow & f_1 \sqsubseteq f_2 \\
\forall f_1, f_2. & f_1 \sqsubseteq f_2 & \Leftrightarrow & \forall t \in \mathit{trees}(f_1).\ \exists t' \in \mathit{trees}(f_1).\ t \sqsubseteq t'
\end{array}$$

Reflexivity and transitivity of $\sqsubseteq$ easily follow by its definition.

Lemma 5.5
For each f and f':

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

Lemma 5.7 states that μXQ queries are monotone with respect to $\sqsubseteq$ extended to substitutions in the obvious way:

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

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 $\overline{x}$ 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). This strong property is not needed in order to prove soundness results, but is crucial for completeness.

Query monotonicity depends on monotonicity of filtering and axis steps.

Lemma 5.6 (*Monotonicity of Filtering, Childr and DOS*)

$$\begin{array}{llll}
1. & \forall f, f'.\ f \sqsubseteq f' & \Rightarrow & f :: \mathit{NodeTest} \sqsubseteq f' :: \mathit{NodeTest} \\
 & & & \mathit{dos}(f) \sqsubseteq \mathit{dos}(f') \\
2. & \forall t, t'.\ t \sqsubseteq t' & \Rightarrow & \mathit{childr}(t) \sqsubseteq \mathit{childr}(t')
\end{array}$$

Lemma 5.7 (*Query Monotonicity*)

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

Query monotonicity has the following corollary, which implies monotonicity of substitution extension.

Corollary 5.8
Given a well formed query Q and a substitution ρ such that $FV(Q) \subseteq dom(\rho) \cup \{\chi\}$:

$$f_1 \sqsubseteq f_2 \;\Rightarrow\; \prod_{t \in trees(f_1)} [\![Q]\!]_{\rho,\chi \mapsto t} \sqsubseteq \prod_{t \in trees(f_2)} [\![Q]\!]_{\rho,\chi \mapsto t}$$

Lemma 5.9 (Extension Monotonicity)
For any 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 5.11 shows that, after splitting a type T, all the types in $Split_E(T)$ 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 (cases) 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 notion of correctness). This is the key lemma that allows us to prove that this type system infers a type which is a lower-bound for the set of all possible results for Q, modulo $\sqsubseteq$ (Theorem 5.22). The combination of upper and lower-bound properties 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 5.25 and Theorem 5.27).

Lemma 5.11 is a corollary of the following lemma.

Lemma 5.10 (Closure of Split Types)
For any $*$-guarded environment E and type T well-formed in E, for any $A \in Split_E(T)$:

$$\forall f_1, f_2 \in [\![A]\!]_E.\ \exists f \in [\![A]\!]_E.\ f_i \sqsubseteq f \text{ for } i = 1, 2$$

Lemma 5.11
For any type A defined in a $*$-guarded environment E, if $Split_E(A) = \{A\}$ then,

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

5.4 Soundness and Completeness of Error-Checking

Type-splitting type rules assume environments Γ to be strongly-$*$-guarded, according to the following definition.

Definition 5.12 (strongly-$*$-guarded Γ)
Γ is strongly-$*$-guarded in E $*$-guarded if and only if, for every $\chi : T$ in Γ, $Split_E(T) = \{T\}$.

Notice that, in a $*$-guarded type environment E, recursion is $*$-guarded, but a type T that appears in an equation $X = T$ may present non-$*$-guarded unions, as in $T \equiv l[U* \mid V*]$. In a strongly-$*$-guarded variable environment Γ, instead, for every variable definition $\chi : T$, unions in T must be $*$-guarded, as in $T = l[(U \mid V)*]$, because of the condition $Split_E(T) = \{T\}$.

Rules (TypeInElSplitting) and (TypeLetSplitting) are the only rules that extend the environment Γ, and they preserve strongly-$*$-guardedness, as, whenever a not $*$-guarded union is met in the typing process, the union is split by these rules.

Observe that, in order to use type-splitting rules over a judgment containing a $*$-guarded environment E and a generic environment Γ, the latter has first to be split in a finite number of strongly-$*$-guarded environments, as defined below.

Definition 5.13 (*Query Variables Environment Splitting*)
For each Γ well-formed in a $*$-guarded type environment E, we extend splitting to Γ in order to decompose it into a finite set of strongly-$*$-guarded environments, which we call $\mathit{SplitVEnv}(\Gamma, E)$:

$$\begin{array}{lcl} \mathit{SplitVEnv}((), E) & \triangleq & \emptyset \\ \mathit{SplitVEnv}((\Gamma, \chi : T), E) & \triangleq & \{\Gamma', \chi : A \mid \Gamma' \in \mathit{SplitVEnv}(\Gamma, E) \wedge A \in \mathit{Split}_E(T)\} \end{array}$$

Lemma 5.14
For each $*$-guarded type environment E and Γ well-formed in E:

$$\bigcup_{\Gamma' \in \mathit{SplitVEnv}(\Gamma, E)} \mathscr{R}(E, \Gamma') = \mathscr{R}(E, \Gamma)$$

The strongly-$*$-guarded variable environments enjoy the property stated in Lemma 5.15, which generalizes Lemma 5.11 from typed values to typed substitutions, and is one of the crucial lemmas needed to prove the lower bound property in Theorem 5.22.

Lemma 5.15
For any strongly-$*$-guarded and well-formed Γ in a $*$-guarded type environment E, and $\rho_1, \ldots, \rho_n \in \mathscr{R}(E, \Gamma)$, there exists $\rho \in \mathscr{R}(E, \Gamma)$ such that $\rho_i \sqsubseteq \rho$ for $i = 1 \ldots n$.

Definition 5.16 (*Typing by Splitting*)
For any well-formed query Q, any $*$-guarded environment E and Γ well-formed in E, we indicate with $E; \Gamma \Vdash_\beta Q : (U; \mathscr{S})$ the following facts:

$$\begin{array}{ll} (1) & \mathit{SplitVEnv}(\Gamma, E) = \{\Gamma_1, \ldots, \Gamma_n\} \\ (2) & E; \Gamma_i \vdash_\beta Q : (U_i; \mathscr{S}_i) \quad i = 1 \ldots n \\ (3) & U \equiv U_1 \mid \ldots \mid U_n \\ (4) & \mathscr{S} = \bigcap_{i=1 \ldots n} \mathscr{S}_i \end{array}$$

The type-splitting system enjoys soundness and completeness of type inference and error-checking. To prove these properties, we will use the following lemmas.

Lemma 5.17 (*Invariance of Well-Formation*)
For any well-formed judgement $E; \Gamma \vdash_\beta Q : (U; \mathscr{S})$ with Γ strongly-$*$-guarded, the backward application of the rules produces judgements that are well-formed as well, and only contain strongly-$*$-guarded environments.

The following lemma is crucial to prove completeness of the type-splitting systems, as it proves a lower bound property for the particular case of $dos(f)$. This lemma is actually an extension of Lemma 4.12.

Lemma 5.18 (*Soundness and Completeness of DOS Type*)
For any E well-formed, T such that $E \vdash T$ Def, and U such that

$$\begin{array}{rcl} \mathit{SubTrees}_E(T) & = & \{U_1, \ldots, U_n\} \\ U & \equiv & (U_1 \mid \ldots \mid U_n)* \end{array}$$

then:

$$
\begin{array}{ll}
(1) & \forall f \in [\![T]\!]_E.\ dos(f) \in [\![U]\!]_E \\
(2) & \forall f \in [\![U]\!]_E.\ \exists\{f'_1,\dots,f'_m\} \subseteq [\![T]\!]_E.\ f \sqsubseteq dos(f'_1,\dots,f'_m) \\
(3) & Split_E(T) = \{T\} \ \Rightarrow\ \forall f \in [\![U]\!]_E.\ \exists f' \in [\![T]\!]_E.\ \ f \sqsubseteq dos(f')
\end{array}
$$

If we do not assume $Split_E(T) = \{T\}$, then property

$$\forall f \in [\![U]\!]_E.\ \exists f' \in [\![T]\!]_E.\ \ f \sqsubseteq dos(f')$$

does not hold. Consider the type $T \equiv c[a[] \mid b[]]$. We have $U \equiv (c[a[] \mid b[]] \mid a[] \mid b[])*$ and $f = c[a[]], b[] \in [\![U]\!]_E$. And for f there exists no $f' \in [\![T]\!]_E$ such that $f \sqsubseteq dos(f')$.

The lemma below entails the upper bound property for the type splitting system; as for the previous type-system, this is the basis for soundness of error checking.

Lemma 5.19
In the type splitting system, for each Q, $*$-guarded E, and Γ strongly-$*$-guarded and well-formed in E:

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

Theorem 5.20 (*Upper Bound for the Type-Splitting System*)
For each Q, $*$-guarded and well-formed E, and Γ well-formed in E:

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

As anticipated, thanks to the strongly-$*$-guardedness restriction, the type-splitting system is complete up to forest simulation $\sqsubseteq$. Completeness is proved in Theorem 5.22 and follows from the next lemmas, where we first consider the special case where Γ is strongly-$*$-guarded.

Lemma 5.21
In the type-splitting system, for each Q, $*$-guarded E, and Γ strongly-$*$-guarded and well-formed in E:

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

Theorem 5.22 (*Lower Bound for the Type-Splitting System*)
For each Q, $*$-guarded E, and Γ well-formed in E:

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

The lower and upper bound properties imply ()-precision, which will be crucial for completeness of error-checking.

Corollary 5.23 (*( )-precision*)
In the type splitting system, for each Q, $*$-guarded E, and Γ strongly-$*$-guarded and well-formed in E, if $E;\Gamma \vdash_\beta\ Q : (U;\ \_)$ then:

$$[\![U]\!]_E = \{()\}\ \Leftrightarrow\ \forall \rho \in \mathscr{R}(E,\Gamma).\ [\![Q]\!]_\rho = ()$$

We are now ready to state the soundness and completeness properties of error-checking for the type splitting system. We start with soundness, whose proof is quite similar to that of soundness in the absence of splitting.

Lemma 5.24
In the type-splitting system, for each query Q, $*$-guarded E, Γ strongly-$*$-guarded and well-formed in E:

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

Theorem 5.25 (*Soundness of Error-Checking for the Type-Splitting System*)
For each Q, $*$-guarded E, and Γ well-formed in E:

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

We can finally state the completeness property of error checking; as usual, the proof is in the web supplementary material.

Lemma 5.26
In the type-splitting system, for each Q, $*$-guarded E, and Γ strongly-$*$-guarded and well-formed in E:

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

Theorem 5.27 (*Completeness of Error-Checking for the Type-Splitting System*)
For each Q, $*$-guarded E, and Γ well-formed in E:

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

6 The cost of case-analysis

Case-analysis is central in our type systems, but it can make type checking rather expensive. Essentially, type-checking time can be exponential in the size of queries and types due to the possible presence of nested `for` queries.

Case analysis is caused by the (TypeIn*) rules used to type-check `for` iteration and by the (Type*Splitting) rules used when the variable environment Γ is extended with a new variable.

We start with discussing for-case-analysis. We recall that `for` queries are checked by the following rule.

$$\frac{\begin{array}{l} E;\,\Gamma \vdash_{\beta.0}\ Q_1 : (T_1;\ \mathscr{S}_1) \\ E;\,\Gamma \vdash_{\beta.1}\ \overline{x}\ \mathtt{in}\ T_1 \rightarrow Q_2 : (T';\ \mathscr{S}_2) \\ \mathscr{S} =\ \text{if } T_1 \sim_E () \text{ then } \{\beta\} \text{ else } \emptyset \end{array}}{E;\,\Gamma \vdash_{\beta}\ \mathtt{for}\ \overline{x}\ \mathtt{in}\ Q_1\ \mathtt{return}\ Q_2 : (T';\ \mathscr{S}_1 \cup \mathscr{S}_2 \cup \mathscr{S})} \quad (\textsc{TypeFor})$$

The rules for the second premise decompose T_1 in tree types and then type-check Q_2 once for each tree type at the top level of T_1. So, if $T_1 \equiv l[m[X]] \mid (n[], B)$, then Q_2 is checked three times, with respect to the assumptions $\overline{x} : l[m[X]]$, $\overline{x} : n[]$, and $\overline{x} : B$. Hence, when Q_2 contains nested `for` queries, the number of cases to analyze can be exponential in the depth of `for` nesting.

However, in practice, queries and types usually satisfy some properties that allow a fast type checking. Namely, for-case-analysis is often performed on a type having only one tree type at the top-level, hence there is only one case to iterate on. Moreover, in most cases, type-splitting operations stop quite soon as union is often $*$-guarded (recall that $Split_E(T*) = \{T*\}$). As we will see, these facts allow one easy optimization that drastically reduces the cost of case analysis.

We are now going to describe these properties of types and queries that allow fast checking. As we will explain, these properties are extremely common, in the sense that they are satisfied by most use cases we found described (almost all XQuery use cases in (Chamberlin *et al.*, 2003)). We will first discuss the for-case-analysis, and then type-splitting case analysis.

It is important to notice that, when these properties are violated only by a small subset of the types or the subqueries of a query, the query can still be type-checked in a reasonable time. Type checking becomes infeasible only when these properties are systematically violated.

The property of interest for types is given by the following definition.

Definition 6.1 (*Label-Deterministic Types*)
A type T well-formed in E is label-deterministic, with respect to E, if and only if $T \rightarrow_e^E m[U]$ and $T \rightarrow_e^E m[U']$ imply that $U \equiv U'$

This property always holds for types defined by a DTD. XML Schema declarations allow one to define types that are not label-deterministic, but this possibility is, in practice, used for a minority of types.

This property is extended to environments type-by-type: each type has to be label-deterministic, but different types may associate different content types to the same label.

Definition 6.2 (*Label-Deterministic Environments*)
Given Γ well-formed in E, we say that Γ is label-deterministic, with respect to E, or that (E, Γ) is label-deterministic, if and only if for each $\chi : T$ in Γ, T is label-deterministic with respect to E.

The property is usually satisfied when type-checking starts, but it could be broken during type checking, when the environment Γ is extended by the application of rules (TypeLetSplitting) and (TypeInElSplitting). This is not going to happen, however, if the checked query satisfies another common property, that we define now.

Definition 6.3 (*Left-Path Queries*)
A query Q is left-path if $\forall \beta \in Locs(Q)$:

$$((Q)_{|\beta} \equiv \texttt{for } \overline{x} \texttt{ in } Q_1 \texttt{ return } Q_2 \ \vee \ (Q)_{|\beta} \equiv \texttt{let } x ::= Q_1 \texttt{ return } Q_2) \Rightarrow$$

$$Q_1 \equiv \chi\ Step_1\ Step_2 \ldots Step_n$$

where $Step_i$ is either $/l$ or $//l$.

The left-path property is satisfied by almost all queries we found in the published use-cases. However, the iteration over paths ending with `//node()` or `/node()`, as the ones below:

```
$x/tag1/.../tagn/node()
$x/tag1/.../tagn//node()
```

is sometimes met. We will discuss it shortly.

We can now prove that for label-deterministic types and left-path queries, type checking can be efficiently performed by the type system based on for-case-analysis. Afterward, we will discuss the further case-analysis introduced by type splitting.

In the sequel, we use the following definition:

Definition 6.4
We define $\text{UpperTrees}_E(T)$ as the set of the tree types of the trees that can be found at the top level of a forest of type T, formally:

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

The following lemmas allow us to optimize for-case-analysis.

Lemma 6.5
Assume $E;\, \Gamma \vdash_\beta\ Q : (T;\ \_)$, (E,Γ) is label-deterministic and

$$Q \equiv \chi\ Step_1\ Step_2 \ldots Step_n$$

where $Step_i$ is either $/l_i$ or $//l_i$. Then T is label-deterministic. Moreover,

$$\text{UpperTrees}_E(T) \subseteq \{l_n[T']\}$$

for some T', where l_n is the label of $Step_n$.

Lemma 6.6
If E is $*$-guarded and T is well-defined and label-deterministic with respect to E, then each $A \in Split_E(T)$ is label-deterministic with respect to E.

Lemma 6.7
If $E;\, \Gamma \vdash_\beta\ Q : (T;\ \_)$, (E,Γ) is label-deterministic and Q is left-path, then for each judgement of shape

$$E';\, \Gamma' \vdash_\beta\ Q' : (T'';\ \mathscr{S})$$

or

$$E';\, \Gamma' \vdash_\beta\ \overline{x}\ \texttt{in}\ T_1 \rightarrow Q' : (T'';\ \mathscr{S})$$

in the proof tree of $E;\, \Gamma \vdash_\beta\ Q : (T;\ \_)$, the pair (E', Γ') is label-deterministic and Q' is left-path. Moreover, in the second case, T_1 is label-deterministic.

The following lemma formalizes the desired optimization.

Lemma 6.8 (*Label-Deterministic Analysis*)
If $E;\, \Gamma \vdash_\beta\ Q : (T;\ \_\ )$, (Γ, E) is label-deterministic and Q is left-path, then for each judgement

$$E';\, \Gamma' \vdash_\beta\ \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : (T';\ \mathscr{S}')$$

in the proof tree of $E;\, \Gamma \vdash_\beta\ Q : (T;\ \mathscr{S})$, we have

$$\exists\ T_1, m, U.\ \ E';\, \Gamma' \vdash_\beta\ Q_1 : (T_1;\ \_)\ \wedge\ \text{UpperTrees}_{E'}(T_1) \subseteq \{m[U]\}$$

The above lemma implies that each time a judgement

$$E';\, \Gamma' \vdash_\beta\ \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : (T';\ \mathscr{S}')$$

is met while proving $E;\, \Gamma \vdash_\beta\ Q : (T;\ \mathscr{S})$, then case-analysis can be drastically optimized. Indeed, by (TypeFor), $E';\ \Gamma' \vdash_\beta\ \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : (T';\ \mathscr{S}')$ is proved by first proving $E';\ \Gamma' \vdash_{\beta.0}\ Q_1 : (T_1;\ \mathscr{S}_1)$ and then by performing case-analysis on T_1 to prove $E';\ \Gamma' \vdash_{\beta.1}\ \overline{x}\ \texttt{in}\ T_1 \rightarrow Q_2 : (T';\ \mathscr{S}'')$. But, since $\text{UpperTrees}_{E'}(T_1) \subseteq \{m[U]\}$, the case-analysis always ends up with a tree type $m[U]$ which is used by rule (TypeInElSplitting) to prove $E';\ \Gamma', \overline{x} : m[U] \vdash_{\beta.1}\ Q_2$:

$(U'; \mathscr{S}_2)$. This means that Q_2 can be checked once, and then its type-proof can be reused each time $m[U]$ is reached during case analysis over T_1.

As we anticipated, the left-path condition is actually violated by some queries that, although infrequent, are not completely unusual. For instance, in the XQuery use cases `node()` is sometimes used in XPath expressions that are inside a left-subquery of a `for` query. This forces the `return` clause to be checked a number of times that may be linear in the size of the involved types. However, if this violation is not nested, the slow down is linear rather than exponential, which is acceptable; this covers all violations found in the use cases in Chamberlin *et al.* (2003), but it also holds for rather complex queries as those in the XMark benchmark set (Schmidt *et al.*, 2002). More generally, we believe that the class of queries with less than, say, three nested `node()` steps is quite vast, and for this class our typing is not exponential. We are, however, planning to implement the system and perform some empirical testing, in order to confirm these beliefs.

We analyze now type-splitting, which is the second source of case-analysis.

Type-splitting, in practice, has a very limited cost. By analyzing types defined in Chamberlin *et al.* (2003) and in many other repositories over the Web, we have realized that, whenever union is used to specify element content, then union is almost invariably $*$-guarded. This means that, during type-analysis, for a large class of input types in these cases, type-splitting produces very few cases, since splitting stops in front of $T*$ types.

Consider, for example, the following DTD from Chamberlin *et al.* (2003):

```
<!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 Chamberlin *et al.* (2003).

```
<!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, as the DTD fragment below, which describes time stamps used in Unix systems management. Hence, splitting generally produces very few cases to analyze.

```
<!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 amount of nesting. We suspect that 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.

7 Extending μXQ

μXQ, although inspired by XQuery, omits many important features. We discuss here some of them.

7.1 where *clauses*

In this section we extend μXQ's FLWR constructs with a *where* clause. We shall discuss how this extension affects the properties of soundness and completeness that we proved previously.

We first extend the language syntax with a non-terminal P of predicates and with a *where* P clause. We will now interpret a *for* or *let* construct with no *where* clause as an abbreviation for a missing where true.

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

If we extend the semantics, the type rules and the notions of *CriticalLocs*(Q) and *Ext*(ρ, Q, ϵ) in the natural way, we obtain a type system that is sound, but is not complete. Table 7.1 shows the new definitions of $[\![Q]\!]_\rho$, *Ext*(ρ, Q, ϵ) and *CriticalLocs*(Q), while Corollary 7.7 is the corresponding soundness result. Observe that our type rules do not require that Q and Q' have related types in a comparison

Table 7.1. *μXQ with* where

New semantic clauses (the definition of $[\![P]\!]_\rho$ is completely standard)

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

Modified type rules (the other rules to type-check P conditions are standard)

(TypeLetWSplitting)
$$\frac{\begin{array}{l} E;\,\Gamma \vdash_{\beta.0} Q_1 : (T_1;\ \mathscr{S}) \\ Split_E(T_1) = \{A_1,\ldots,A_n\} \\ E;\,\Gamma,\ x : A_i \vdash_{\beta.1} P : (Bool;\ \mathscr{S}'_i) \\ E;\,\Gamma,\ x : A_i \vdash_{\beta.2} Q_2 : (U_i;\ \mathscr{S}_i;\ \mathscr{W}_i) \end{array}}{\begin{array}{l} E;\,\Gamma \vdash_\beta \texttt{let } x := Q_1 \texttt{ where } P \texttt{ return } Q_2 \\ \quad : (U_1 \mid \ldots \mid U_n \mid ();\ \mathscr{S} \cup \cap_{i=1\ldots n}\mathscr{S}'_i \cup \cap_{i=1\ldots n}\mathscr{S}_i) \end{array}}$$

(TypeEq)
$$\frac{\begin{array}{l} E;\,\Gamma \vdash_{\beta.0} Q_0 : (T_0;\ \mathscr{S}_0) \\ E;\,\Gamma \vdash_{\beta.1} Q_1 : (T_1;\ \mathscr{S}_1) \end{array}}{E;\,\Gamma \vdash_\beta Q_0 = Q_1 : (Bool;\ \mathscr{S}_0 \cup \mathscr{S}_1)}$$

(TypeFor)
$$\frac{\begin{array}{l} E;\,\Gamma \vdash_{\beta.0} Q_1 : (T_1;\ \mathscr{S}_0) \\ E;\,\Gamma \vdash_{\beta.1} \bar{x} \texttt{ in } T_1 \rightarrow Q_2 \texttt{ where } P : (T_2;\ \mathscr{S}_1) \\ \mathscr{S} = \text{ if } T_1 \sim_E () \text{ then } \{\beta.0\} \text{ else } \emptyset \end{array}}{E;\,\Gamma \vdash_\beta \texttt{for } \bar{x} \texttt{ in } Q_1 \texttt{ where } P \texttt{ return } Q_2 : (T_2 \mid ();\ \mathscr{S}_0 \cup \mathscr{S}_1 \cup \mathscr{S})}$$

New definition of *CriticalLocs*(Q)

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

Substitution extension

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

$Ext(\rho,\, \texttt{let } x := Q_0 \texttt{ where } P \texttt{ return } Q_1,\, 2.\beta)$
$$\triangleq \text{if } [\![P]\!]_{\rho,x\mapsto[\![Q_1]\!]_\rho} \text{ then } Ext((\rho, x \mapsto [\![Q_0]\!]_\rho),\, Q_1,\, \beta) \text{ else } \emptyset$$

$Ext(\rho,\, \texttt{for } \bar{x} \texttt{ in } Q_0 \texttt{ where } P \texttt{ return } Q_1,\, 2.\beta)$
$$\triangleq \bigcup_{t\in trees([\![Q_0]\!]_\rho)\ s.t.\ [\![P]\!]_{\rho,\bar{x}\mapsto t}=true} Ext((\rho, \bar{x} \mapsto t),\, Q_1,\, \beta)$$

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

$Q\ \delta\ Q'$. This is an important issue, but is orthogonal to the navigation-correctness problem that we study in this paper.

The type system is not complete for a couple of reasons. First of all, any query whose condition is not satisfiable, like `for` $\bar{x}$ `in` Q `where false return` Q', is FE-incorrect, even if it contains no wrong path. But such queries are correct in the type system.

Completeness is also lost for a subtler, and more interesting, reason: because the *where* clause makes the language lose its monotonicity. Consider the following query:

$$Q_0 \equiv \texttt{for \$x in a[] where not (\$y = ()) return \$x}$$

Monotonicity would imply that $f \sqsubseteq f' \Rightarrow [\![Q_0]\!]_{\$y\mapsto f} \sqsubseteq [\![Q_0]\!]_{\$y\mapsto f'}$ (Lemma 5.7). In this case, instead, we have $() \sqsubseteq b[]$ but $[\![Q_0]\!]_{\$y\mapsto ()} \not\sqsubseteq [\![Q_0]\!]_{\$y\mapsto b[]}$, since $[\![Q_0]\!]_{\$y\mapsto ()} = a[]$ and $[\![Q_0]\!]_{\$y\mapsto b[]} = ()$. Hence, Lemma 5.7 (monotonicity) does not hold any more, and we lose the lower-bound property of type-inference (Theorem 5.22) and completeness of error-checking (Theorem 5.27), which depend on that lemma.

The fact that type-checking over the extended language is not complete 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. Nevertheless, completeness over an important kernel that includes paths, element construction, iteration, and query composition, is an essential property since these operators play a major role in any significant piece of code.

Still, our type-system *is* complete, in a weaker sense, over the language extended with *where*, provided that one adopts the appropriate notion of "path-errors". In other terms, we can define a notion of "path-errors" which exactly corresponds to the pieces of code that are flagged by our type-system.

To justify this notion, consider the following queries, posed on the same schema used in Section 3.

Q_1 : `for $c in $contacts where ($c/fone/text() = 1000) return $c/phone`
Q_2 : `for $c in $contacts where (false) return $c/phone`
Q_3 : `for $c in $contacts where ($c = data[fax[2000]]) return $c/phone`
Q_4 : `for $c in $contacts where not($c/mobile = ()) return $c/phone`

All these queries always return empty forests, hence are FE-incorrect. The first has a path-error (`$c/fone`). The second and the third present unsatisfiable conditions, but no wrong path. The fourth only presents sensible paths as well, but it always returns an empty forest since the *where* condition stops any contacts with $\$c/mobile = ()$, and the type of $\$c$ tells us that these stopped contacts are the only ones that may contribute data to the result. One *may* argue that the "filtering anomalies" in queries Q_2, Q_3, and Q_4 would deserve some form of static detection, and we are actually tempted to go in this direction, in some future. However, we believe that these "filtering anomalies" are FE-errors which should *not* be classified as path-errors, and that their prevention should be studied separately.

In order to formally define a notion of path-error that corresponds to this intuition, hence that only considers Q_1 as path-wrong, we will define a transformation *where-drop*(Q) that moves every subquery from the where-predicates P to new `let` binders, and then removes what is left of the *where* clause. Any FE-error that remains after this transformation is then defined to be a *path-error*. It is not difficult to prove that our type-system is complete with respect to this subclass of FE-errors. We are now going to formalize this.

The function *where-drop*(Q) is defined as the result of a two-steps process. In the first step we repeatedly apply the following rewritings, until every P condition is only applied to variables; the rewriting step is applied in any order, at any nesting depth inside the query. The result is affected by the order we choose, but the semantics of the result is not.[6]

Subquery extraction
for any non-variable Q' such that $Q' \; \delta \; Q''$ or $Q'' \; \delta \; Q'$ appears in P
and for some fresh variable y :

$$\begin{array}{l}
\texttt{for } \overline{x} \texttt{ in } Q_0 \texttt{ where } P \texttt{ return } Q_1 \\
\;\to\; \texttt{for } \overline{x} \texttt{ in } Q_0 \texttt{ return let } y := Q' \texttt{ where } P\{Q' \leftarrow y\} \texttt{ return } Q_1 \\
\texttt{let } x := Q_0 \texttt{ where } P \texttt{ return } Q_1 \\
\;\to\; \texttt{let } x ::= Q_0 \texttt{ return let } y := Q' \texttt{ where } P\{Q' \leftarrow y\} \texttt{ return } Q_1
\end{array}$$

We then repeatedly apply the following transformations, everywhere inside the query, until every *where* clause has been removed. In this case, the result does not depend on the clause we start from.

Where cancellation

$$\begin{array}{lcl}
\texttt{for } \overline{x} \texttt{ in } Q_0 \texttt{ where } P \texttt{ return } Q_1 & \to & \texttt{for } \overline{x} \texttt{ in } Q_0 \texttt{ return } Q_1 \\
\texttt{let } x := Q_0 \texttt{ where } P \texttt{ return } Q_1 & \to & \texttt{let } x ::= Q_0 \texttt{ return } Q_1
\end{array}$$

We present here an example of the *where-drop*(_) process. We omit `return` before `for` and `let`, hence obtaining the usual syntax of FLWR expression.

$$\begin{array}{l} \texttt{for } \overline{x} \texttt{ in } \textit{contacts} \\ \texttt{where } \overline{x}/\textit{fone} = \textit{phone}[13] \\ \texttt{return } \overline{x} \end{array} \;\Rightarrow\; \begin{array}{l} \texttt{for } \overline{x} \texttt{ in } \textit{contacts} \\ \texttt{let } z := \overline{x}/\textit{fone}, \\ \texttt{let } y := \textit{phone}[13] \\ \texttt{where } z = y \\ \texttt{return } \overline{x} \end{array} \;\Rightarrow\; \begin{array}{l} \texttt{for } \overline{x} \texttt{ in } \textit{contacts} \\ \texttt{let } z := \overline{x}/\textit{fone}, \\ \texttt{let } y := \textit{phone}[13] \\ \texttt{return } \overline{x} \end{array}$$

We can now define path-correctness as follows.

Definition 7.1 (*Where-dropping path-correctness*)
A query Q in the language extended with *where* is path-correct w.r.t. a set of valid substitutions $\mathscr{R}$ if *where-drop*(Q) is FE-correct.

We can now observe some facts.

1. a *subquery extraction* step does not change query semantics;
2. if *where cancellation* is performed starting from outermost clauses, every step increases the query semantics, according to simulation order;[7]
3. if a *subquery extraction* step transforms Q into Q', then any proof tree for $E; \Gamma \vdash_\beta Q : (T; \mathscr{S})$ can be transformed into a proof tree for $E; \Gamma \vdash_\beta Q' : (T; \mathscr{S}')$, where $\mathscr{S}$ and $\mathscr{S}'$ have the same cardinality;
4. if a *where cancellation* step, applied during the second phase, transforms Q into Q', and $E; \Gamma \vdash_\beta Q : (T; \mathscr{S})$, then $E; \Gamma \vdash_\beta Q' : (T; \mathscr{S}')$ holds, where $\mathscr{S}$ and $\mathscr{S}'$ have the same cardinality;
5. $[\![Q]\!]_\rho \sqsubseteq [\![\textit{where-drop}(Q)]\!]_\rho$;
6. $E; \Gamma \vdash_\beta Q : (T; \mathscr{S})$ implies that $E; \Gamma \vdash_\beta \textit{where-drop}(Q) : (T; \mathscr{S}')$, where $\mathscr{S}$ and $\mathscr{S}'$ have the same cardinality.

[6] This step could be avoided by restricting the syntax so that only variable comparisons are allowed.
[7] Actually, the semantics is also increased wrt the relation that specifies that $f \leqslant f'$ if f is obtained by removing some subtrees from f', but simulation is enough for our purposes here.

From these observations, we get our results. The results are a bit weaker than expected, since they do not actually relate the location where an error is found by the type rules with the actual error location. This is a minor problem related to the manipulations performed by *where-drop*(_).

Theorem 7.2 (*Upper Bound*)
For any well-formed environment E, Γ well-formed in E, and well-formed Q with *where* clauses:

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

Theorem 7.3 (*Soundness of Path-Error-Checking*)
For any well-formed environment E, Γ well-formed in E, and query Q with *where* clauses:

$$E;\, \Gamma \Vdash_\beta\ Q : (U;\ \mathscr{S}) \ \wedge\ \beta.\alpha \in \mathscr{S} \ \Rightarrow Q \text{ has a path-error w.r.t. } \mathscr{R}(E, \Gamma)$$

Theorem 7.4 (*Lower Bound*)
For each query Q with *where* clauses, $*$-guarded E, and Γ well-formed in E, in the type-splitting system:

$$E;\, \Gamma \Vdash_\beta\ Q : (U;\ \_) \ \Rightarrow\ \forall f \in [\![U]\!]_E.\ \exists \rho \in \mathscr{R}(E, \Gamma).\ f \sqsubseteq [\![\textit{where-drop}(Q)]\!]_\rho$$

Theorem 7.5
For each query Q with *where* clauses, $*$-guarded E, and Γ and well-formed in E:

$$E;\, \Gamma \Vdash_\beta\ Q : (\_;\ \mathscr{S}) \ \Rightarrow\ (Q \text{ has a path-error w.r.t. } \mathscr{R}(E, \Gamma) \ \Rightarrow\ \exists \alpha.\ \beta.\alpha \in \mathscr{S})$$

Soundness and correctness with respect to path-errors immediately imply the result that we announced first, soundness with respect to FE-errors. We now state the following lemma, which relates path-correctness with FE-correctness. The location of the path-error and of the FE-error can be different, because of the *where-drop*(_) process.

Lemma 7.6
For each query Q with *where* clauses, if Q has a path-error at $\beta \in \mathit{CriticalLocs}(Q)$ w.r.t. $\mathscr{R}$, then Q has an FE-error.

As a corollary, we get soundness of type-checking with respect to FE-errors.

Corollary 7.7 (*Soundness of FE-Error-Checking*)
For any well-formed environment E, Γ well-formed in E, and query Q with *where* clauses:

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

The above rules completely ignore the errors related to the comparison of non-comparable values. Currently, we are investigating a new kind of approach that considers this aspects as well, and preliminary results are positive (Colazzo & Sartiani, 2005).

7.2 Other issues

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, if T is the type of a path expression and

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

then type T can be weakened to its supertype

$$(U_1 \mid \ldots \mid U_n)*$$

without compromising Theorems 5.20, 5.22, and 5.27. 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 issues of predefined functions, (recursive) function definition and invocation, and validation, because we think that they may be dealt with by 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, but soundness is the only property one can aim to when a full language is treated.

8 Related work

Our Previous Work The work we presented in Colazzo *et al.* (2002) 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 (Hosoya & Pierce, 2003) is a typed, functional, Turing complete programming language. It is based on an ML-like pattern language that implements a *one-match* semantics, i.e. every pattern, instead of collecting every matched piece of data (as in standard query languages), only binds the first match. XDuce is nearer to a programming language than to a query language, but we consider it here since it is 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 (Benzaken *et al.*, 2003) 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 worked with the CDuce team to extend our approach to that language and the derived query language CQL (Benzaken *et al.*, 2004; Benzaken *et al.*, 2005), and we have shown that the approach formalized here can also be used in programming paradigms rather different from that of XQuery (Castagna *et al.*, 2005).

XQuery XQuery type inference (Draper *et al.*, 2003) 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. In the August 2003 Working Draft, the W3C XML Query Working Group added a new rule to the type system of XQuery, stating that it is a static error for any expression other than the empty-sequence expression to have the empty type. This 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.

XQuery type system is based on nominal types, while our type system is based on structural types. The main difference between nominal and structural type systems for XML is that the former, although slightly less elegant, admits much simpler, and more efficient, subtype-checking algorithms. This is not very relevant to this work, since we decided to ignore subtyping. We believe that our type rules would require minimal modifications in order to express a nominal type-system, and our results should not be affected. However, we leave the non-trivial proof of this fact to future work.

Finally, we observe that case-analysis in the typing of XQuery `for` clauses was first introduced in Fernández *et al.* (2001), where completeness was claimed for the basic case of path projection (e.g., `$x/tag`). As we have shown, case-analysis is not sufficient to ensure completeness of result analysis and error-checking in the more general case including sequence composition Q, Q' and descendant-or-self axis; type-splitting, instead, as formalized in Section 5, is able to ensure completeness. Moreover, adopting case-analysis implies adopting locations-based error checking techniques, as shown in Section 3.

k-pebbles proposal Dan Suciu et al. develop a formal framework for the definition of result analysis tools (Milo *et al.*, 2000; Alon *et al.*, 2001a). These papers define some upper bounds to what can be accomplished by result type analysis. Our results do not contradict these, since we focus here on a language that is weaker than k-pebbles automata.

9 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.

We discussed the fact that, although our type-system bases its precision on extensive use of case-analysis, it seems not to be too expensive in practice. We described just one optimization that should suffice in most situations; we are investigating other optimizations to widen this result.

We discussed the extension of our results to a language with the features that we did not include in μXQ. This discussion is very preliminary, and many details should be verified. The first thing we would like to check is how our results would be modified, if they were modified in any way, by the adoption of a nominal type system.

We plan to implement an XQuery version of our type system. We would then use that prototype to verify our assumptions about the feasibility of case-analysis in practical cases.

Acknowledgements

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, and his comments helped us a lot. We would also like to thank the anonymous reviewers, for their constructive and helpful remarks.

References

Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J. L. (1997) The Lorel Query Language for Semistuctured Data. *Journal of Digital Libraries,* **1***(1)*, April, 68–88.

Alon, Noga, Milo, Tova, Neven, Frank, Suciu, Dan & Vianu, Victor (2001a) Typechecking XML Views of Relational Databases. *Pages 421–430 of: Proceedings of the 16th annual IEEE Symposium on Logic in Computer Science, 16–19 June 2001, Boston, MA. IEEE Computer Society, 2001.*

Alon, Noga, Milo, Tova, Neven, Frank, Suciu, Dan & Vianu, Victor (2001b) XML with Data Values: Typechecking Revisited. *Proceedings of the Twentieth ACM Sigact-Sigmod-Sigart Symposium on Principles of Database Systems, May 21–23 2001, Santa Barbara, CA.*

Benzaken, Véronique, Castagna, Giuseppe & Miachon, Cédric (2005) A full pattern-based paradigm for xml query processing. *Proceeding of 7th International Symposium on Practical Aspects of Declarative Languages (PADL 2005)*, Long Beach, CA, January 10–11 2005. Lecture Notes in Computer Science, vol. 3350. Springer.

Benzaken, Vronique, Castagna, Giuseppe & Frisch, Alain (2003) CDuce: an XML-centric general-purpose language. *Pages 51–63 of: Proceedings of the Eighth ACM Sigplan International Conference on Functional Programming*. ACM Press.

Benzaken, Vronique, Castagna, Giuseppe & Miachon, Cdric (2004) CQL: a Pattern-based Query Language for XML. *Proceedings of 20th Bases de Données Avancées (BDA) (2004).*

Boag, Scott, Chamberlin, Don, Fernandez, Mary F., Florescu, Daniela, Robie, Jonathan & Siméon, Jérôme (2003) *XQuery 1.0: An XML Query Language*. Tech. rept. World Wide Web Consortium. W3C Working Draft.

Buneman, P., Davidson, S. & Suciu, D. (1995) Programming constructs for unstructured data. *Proceedings of 5th International Workshop on Database Programming Languages.*

Castagna, Giuseppe, Colazzo, Dario & Frisch, Alain (2005) Error Mining for Regular Expression Patterns. *Italian Conference on Theoretical Computer Science (ICTCS).* LNCS.

Chamberlin, Don, Fankhauser, Peter, Florescu, Daniela, Marchiori, Massimo & Robie, Jonathan (2003) *XML Query Use Cases.* Tech. rept. World Wide Web Consortium. W3C Working Draft.

Colazzo, Dario & Sartiani, Carlo (2005) Typechecking Queries for Maintaining Schema Mappings in XML P2P Databases. *Proceedings of the Workshop on Programming Language Technologies for XML (Plan-x 2005), in Conjunction with POPL 2005.*

Colazzo, Dario, Ghelli, Giorgio, Manghi, Paolo & Sartiani, Carlo (2002) Types For Correctness of Queries Over Semistructured Data. *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002),* Madison, WI, June 6–7 2002.

Colazzo, Dario, Ghelli, Giorgio, Manghi, Paolo & Sartiani, Carlo (2004) Types for Path Correctness for XML Queries. *Proceedings of the ACM International Conference on Functional Programming (ICFP),* Snowbird, UT.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S. & Tommasi, M. (1997) *Tree Automata Techniques and Applications.* Available on: `http://www.grappa.univ-lille3.fr/tata`. Release October 1 2002.

Draper, Denise, Fankhauser, Peter, Fernandez, Mary, Malhotra, Ashok, Rose, Kristoffer, Rys, Michael, Siméon, Jérôme & Wadler, Philip (2003) *XQuery 1.0 and XPath 2.0 Formal Semantics.* Tech. rept. World Wide Web Consortium. W3C Working Draft.

Fernandez, M., Florescu, D., Levy, A. & Suciu, D. (1997) A query language for a Web-site management system. *Sigmod Record (ACM SIG on Management of Data)*, **26**(3), 4–11.

Fernández, Mary F., Siméon, Jérôme & Wadler, Philip (2001) A semi-monad for semi-structured data. *Pages 263–300 of:* den Bussche, Jan Van and Vianu, Victor (eds.), *LCDT.* Lecture Notes in Computer Science, vol. 1973. Springer.

Guerra, Rui, Jeuring, Johan & Swierstra, Doaitse (2005) Generic Validation of XPath Data Bindings. *Proceedings of the Workshop on Programming Language Technologies for XML (Plan-x 2005), colocated with POPL 2005.*

Hosoya, Haruo & Pierce, Benjamin C. (2003) Xduce: A statically typed XML processing language. *Acm Trans. Internet Techn.*, **3**(2), 117–148.

Lee, D., Mani, M. & Murata, M. (2000) *Reasoning about XML Schema Languages using Formal Language Theory.* Tech. rept. IBM Almaden Research. Technical Report – IBM Almaden Research.

Milo, Tova, Suciu, Dan & Vianu, Victor (2000) Typechecking for XML Transformers. *Pages 11–22 of: Proceedings of the Nineteenth ACM Sigmod-Sigact-Sigart Symposium on Principles of Database Systems.* ACM Press.

Schmidt, Albrecht, Waas, Florian, Kersten, Martin L., Carey, Michael J., Manolescu, Ioana & Busse, Ralph (2002) Xmark: A benchmark for XML data management. *Pages 974–985 of: Vldb 2002, Proceedings of 28th International Conference on Very Large Data Bases,* August 20–23 2002, Hong Kong, China.

Siméon, Jérôme & Wadler, Philip (2003) The essence of XML. *Pages 1–13 of: Popl 2003, Proceedings of the 30th Sigplan-Sigact Symposium on Principles of Programming Languages,* New Orleans, LA, January 15–17 2003.

Tarski, Alfred (1955) A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5**, 285–309.

Thompson, Henry S., Beech, David, Maloney, Murray & Mendelsohn, Noah (2002) *XML Schema Part 1: Structures.* Tech. rept. World Wide Web Consortium. W3C Recommendation.

Yergeau, Franҫis, Bray, Tim, Paoli, Jean, Sperberg-McQueen, C. M. & Maler, Eve (2004) *Extensible Markup Language (XML) 1.0 (Third Edition).* Tech. rept. World Wide Web Consortium. W3C Recommendation.