Precision and Complexity of XQuery Type Inference

Dario Colazzo [*]
LRI - Université Paris Sud & Leo Team - INRIA
dario.colazzo@lri.fr

Carlo Sartiani
DMI - Università della Basilicata
sartiani@gmail.com

Abstract

A key feature of XQuery is its type system. Any language expression is statically typed and its type is used during program type-checking.

In XQuery, types of input data and functions are defined in terms of regular expression types, but it is quite easy to write queries that generate non-regular languages. As a consequence, any type system for XQuery has to rely on a *type inference* process that approximates the (possibly non-regular) output type of a query with a regular type. This approximation process, while mandatory and unavoidable, may significantly decrease the precision of the inferred types.

In this paper we will analyze the precision and the complexity of the W3C type inference algorithm. By defining an abstract model for the core of XQuery and for its type language (miniXQuery), we will identify the critical issues in the inference process and the sources of precision loss. We will also propose an alternative type inference system, used in the μXQ+ language, and show that in most cases it is more precise without any performance penalties. Finally, we will identify relevant classes of input types for which inference precision can be dramatically improved.

Categories and Subject Descriptors H.2.3 [*Information Systems*]: DATABASE MANAGEMENT—languages

General Terms Languages, Theory

Keywords XML, XQuery, Type Inference

1. Introduction

XML is a formalism for representing any kind of data, ranging from rigid and structured sources to loose semistructured data. XML diffusion is now ubiquitous and most applications leverage on XML to represent their data and to exchange them with other applications.

Many technologies for processing and manipulating XML data have been developed in the recent years. In particular, the World Wide Web Consortium (W3C) has designed a standard query language for XML data called XQuery [2]. XQuery is a functional, Turing-complete, strongly typed language that allows the programmer to navigate through an XML document (or a set of documents), to select relevant fragments of the document, and to combine them to return new documents.

A key feature of XQuery is its type system. In XQuery any language expression is statically typed and its type is used during program type-checking, even though the programmer can disable this feature.

In XQuery, types of input data and functions are defined in terms of regular expression types, but it is quite easy to write queries that generate non-regular languages. As a consequence, any type system for XQuery has to rely on a *type inference* process that approximates the (possibly non-regular) output type of a query with a regular type. This approximation process, while mandatory and unavoidable, may significantly decrease the precision of the inferred types. This is the case of the W3C proposed type system, which relies on some over-approximating rules for expressions widely used in practice (e.g., `for`-iterations). Another source of undesired over-approximation is given by rules to type horizontal and upward XPath axes, for which the type *any* is always inferred.

It is a common folklore that W3C has sacrificed precision in favor of better complexity, and that the W3C typing algorithm runs in polynomial time. An alternative and more precise approach for typing XQuery has been proposed in [6] and used as a basis for other proposals [1, 5]. This type system, used in the μXQ+ language, has a more precise type inference, at the price of a potential exponential explosion of the query output type.

Though the two above mentioned approaches are relatively well known today by the database and programming language communities, a formal, rigorous, and complete analysis showing in which cases the two proposals differ in terms of precision and complexity for type inference, is still missing. Such formal analysis could have a practical relevance as well, since it would provide important information to implementation designers.

In this paper we fill this gap by providing a first comparative analysis. Besides providing a clean and simple formalization of the main typing mechanisms of both approaches, we will formally study their complexity, show in which cases the W3C excessively over-approximates inferred types, identify cases for which inference precision can be dramatically improved, and propose new type rules to better handle these cases. We will also show that, contrary to the common belief, the W3C type system may itself infer types of exponential size wrt the query and the input size.

Paper Outline The rest of the paper is organized as follows. In Section 2 we motivate our work by showing some example of precision loss. In Section 3 we define the type language being used, while in Sections 4, 5, and 6 we present the core language miniXQuery, the W3C type system, and the μXQ+ type system. In Sections 7 and 8, then, we analyze the precision properties of the two type inference systems being studied, as well as their computational complexity. In Section 9 we discuss some related work and Section 10 concludes.

[*] Dario Colazzo has been partially funded by the Codex project, Agence Nationale de la Recherche, decision ANR-08- DEFIS-004.

2. Motivation

Our work on the precision and the complexity of XQuery inference systems originates from the key observation that, while the type language of XQuery is based on regular expression types, it is quite easy to write queries that generate non-regular languages. This introduces approximation phenomena that may harm the result of static analysis.

Consider, for instance, the following query, where $\$db$ has type $a*$ (we use in this section a simple notation inspired by XDuce [11]):

```
<b> { for $x in $db return $x}
    { <c> </c> }
    { for $x in $db return $x}
</b>
```

This query contains two nested queries of the form `for` $\$x$ `in` $\$db$ `return` $\$x$, which iterate over a sequence of n `a`-element and returns each element read. The outer query returns a `b`-element with the following structure: $b[a^n, c, a^n]$. The language generated by the query is obviously non-regular, and is typed by the XQuery type system with $b[a*, c, a*]$. As it can be noted, this type loses the constraint between the first and the second sequence of `a`-elements.

While this kind of approximation phenomena cannot be avoided, as they depend on the nature of the type language, there are still other sources of approximation.

A significant example of approximation in the inference process is represented by the typing of `descendant-or-self` operations. These operations are used to recursively traverse a whole XML tree and pick all the nodes satisfying a given condition. Consider the following query, where $\$db$ has type $b, c, b*$:

```
for $x in $db::descendant-or-self::node()
return $x
```

This simple query inspects the input sequence and returns all the nodes inside the sequence. By applying the inference rules described in the W3C recommendation, we get the following type: $(b|c)*$. This type is a gross over-approximation of the real output type, as its semantics contains, for instance, c. In this case, a more precise output type would be $b, c, b*$.

Another significant source of approximation is given by the typing of `for` iterations. A `for` clause iterates over a sequence of nodes, and binds its variable to each node in the sequence. Consider the following query:

```
for $x in (<b> </b> , <c> </c> , <b> </b>)
return $x
```

This query just iterates over a sequence of three elements and returns each node in the sequence. By applying the inference rules described in the W3C specification we derive the following type: $(b \mid c)+$. In this case, the W3C inference rule for iterations over-approximates the exact output type, i.e., b, c, b, which can be derived by applying the more expensive inference rules of μXQ+. This approximation is justified by the need to keep time complexity low, hence sacrificing precision in favor of a better complexity.

The previous examples show that type inference may introduce significant approximations that may impact the development process, forcing the developer to alter correct queries, as shown by the following example.

Example 2.1 Consider the previous queries and assume they are used to populate a view with the following schema: $b+, c, b*$. When the queries are typechecked against the view schema, the compiler raises a type error in both cases, as $(b \mid c)*$ and $(b \mid c)+$ are not subtypes of $b+, c, b*$. However, the queries are perfectly legal and the error is caused by the over-approximation introduced by the inference process. ∎

Types	T	$::=$	$()$	*empty sequence*
		$\mid$	B	*base type*
		$\mid$	$l[T]$	*element type*
		$\mid$	T, T	*sequence type*
		$\mid$	$T \mid T$	*union type*
		$\mid$	$T*$	*repetition type*
		$\mid$	$T+$	*mandatory repetition type*
		$\mid$	$T?$	*optional type*
		$\mid$	X	*type variable*
Base Type	B	$::=$	`String`	

Figure 1. Type language.

The previous examples, while very simple, are representative of common practical scenarios, as we will see next (Section 7), and highlight the following facts:

- low precision in type inference may be problematic;
- precision has a price to pay in terms of complexity.

In the remaining part of the paper we formally analyze features of the W3C and μXQ+ type systems that are behind the issues highlighted by previous examples. The analysis is then used to show that there is still space for improving the precision of current approaches, while keeping type inference time polynomial.

3. Type Language

The language we want to type describes forests of *unranked*, *node-labeled* trees and is generated by the following grammar:

$$f \quad ::= \quad () \quad \mid \quad b \quad \mid \quad l[f] \quad \mid \quad f, f$$

where $()$ is the empty forest; b identifies atomic values (e.g., strings and integers); $l[f]$ represents an element labeled by a label l belonging to a finite set of labels L, and having nodes in f as its children; the term f, f denotes the ordered concatenation of forests. In the following t will denote an XML tree, which can be either a base value b or an element $l[f]$

Our type language, shown in Figure 1, is based on XDuce [11] regular expression types. The language lacks full horizontal recursion and features the Kleene star $*$ operator. This restriction is canonical, and makes the type language as expressive as regular tree languages [9, 15], hence expressive enough to capture the main type mechanisms of DTD and XML Schema [3, 15, 17, 18].

To support vertical recursion, we use *type environments* and type variables. A type environment associates to each type variable a type definition; the set of variables defined by a type environment is returned by the function $\mathrm{def}(E) = \{X \mid X = T \in E\}$. The type definition associated to a given type variable can be inspected through the function $E(\cdot)$, where $E(X) = T \iff X = T \in E$.

Definition 3.1 ($E \vdash_{Def} T$) *A type T is* well-formed *in E if each variable referenced in T is defined in E.*

We restrict ourselves to element-guarded type environments, which are environments where only element-guarded vertical recursion is allowed (Definition 3.2). For example, we forbid equations such as $X = X \mid ()$ and $X = X, Y$, but allow equations such as $X = l[X \mid ()]$.

Label-Star-Variable Chains

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

***E*-reachability**

$$\begin{array}{ll} l[T] \rightarrow^E_l T & T* \rightarrow^E_* T \\ U, T \rightarrow^E_\epsilon T & U, T \rightarrow^E_\epsilon U \\ U \mid T \rightarrow^E_\epsilon U & U \mid T \rightarrow^E_\epsilon T \end{array}$$

$$(X = T \in E) \Rightarrow X \rightarrow^E_X T$$
$$(T \rightarrow^E_e A \ \wedge \ A \rightarrow^E_{e'} U) \Rightarrow T \rightarrow^E_{e.e'} U$$

Figure 2. Label-Star-Variable Chains "e" and the E-reachability relation "$T \rightarrow^E_e U$".

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 $l[\_]$, where l is a label in the label set L. This is formalized by means of the relation $T \rightarrow^E_e U$ defined in Figure 2. For example, if E is $X = (m[U])*, V$, then $l[X] \rightarrow^E_{l.X.*.m} U$ holds, which means that we can reach U from $l[X]$ by crossing l, expanding X, and crossing $*$ and m (observe that $(T, U) \mid V \rightarrow^E_\epsilon U$: we do not track sequencing and union).

Definition 3.2 (Element-guarded Environments) *E is element-guarded if for each $X = T \in E$ we have $E \vdash_{Def} T$, and for each chain e we have:*

$$X \rightarrow^E_e X \ \Rightarrow \ \exists\, l \in L, e', e'' : \ e = e'.l.e''$$

The rules of the type systems being presented in this paper unfold recursive types until a tree type is met, hence element-guardedness of environments is essential to guarantee termination of these rules.

As usual, the semantics of types is defined as the minimal function that satisfies the following set of monotone equations (the function is well-defined by the Knaster-Tarski Theorem):

$$\begin{array}{lcl} [\![()]\!]_E & \triangleq & \{()\} \\ [\![\mathsf{B}]\!]_E & \triangleq & \{b \ \mid \ b \textit{ is a base value}\} \\ [\![l[T]]\!]_E & \triangleq & \{l[f] \ \mid \ f \in [\![T]\!]_E\} \\ [\![T_1 \mid T_2]\!]_E & \triangleq & [\![T_1]\!]_E \cup [\![T_2]\!]_E \\ [\![T_1, T_2]\!]_E & \triangleq & \{f_1, f_2 \ \mid \ f_i \in [\![T_i]\!]_E\} \\ [\![X]\!]_E & \triangleq & [\![X(E)]\!]_E \\ [\![T?]\!]_E & \triangleq & [\![T \mid ()]\!]_E \\ [\![T+]\!]_E & \triangleq & [\![T]\!]^+_E \\ [\![T*]\!]_E & \triangleq & [\![T]\!]^*_E \end{array}$$

Subtyping is defined via type semantics, as shown below.

Definition 3.3 (Semantic subtyping) *Given two types T and U, T is a subtype of U if and only if the semantics of T is contained into the semantics of U:*

$$T \leqslant U \quad \iff \quad [\![T]\!]_E \subseteq [\![U]\!]_E$$

4. miniXQuery

The language miniXQuery is a minimal language modeling the FLWR core of XQuery. It contains `for`, `let`, `where`, and `return` clauses, and enables the user to specify both the child and the descendants-or-self axes. The predicate language is quite simple

$$\begin{array}{lcl} Q & ::= & () \ \mid \ b \ \mid \ l[Q] \ \mid \ Q, Q \\ & & \mid \ \overline{x}\ \texttt{child} :: \textit{NodeTest} \ \mid \ \overline{x}\ \texttt{dos} :: \textit{NodeTest} \\ & & \mid \ \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{return}\ Q \\ & & \mid \ \texttt{for}\ \overline{x}\ \texttt{in}\ Q\ \texttt{where}\ P\ \texttt{return}\ Q \\ & & \mid \ \texttt{let}\ x ::= Q\ \texttt{return}\ Q \\ & & \mid \ \texttt{let}\ x ::= Q\ \texttt{where}\ P\ \texttt{return}\ Q \\ \textit{NodeTest} & ::= & \texttt{l} \ \mid \texttt{node()} \ \mid \ \texttt{text()} \end{array}$$

$$\begin{array}{lcl} P & ::= & \texttt{true} \ \mid \chi\, \delta\, \chi \mid \texttt{empty}(\chi) \mid P\ \texttt{or}\ P \ \mid \texttt{not}\ P \ \mid \ (P) \\ \chi & ::= & \overline{x} \ \mid \ x \\ \delta & ::= & = \mid < \end{array}$$

Figure 3. The grammar of miniXQuery.

$$\begin{array}{lcl} \texttt{true}(\rho) & \triangleq & \texttt{true} \\ (\chi\, \delta\, \chi')(\rho) & \triangleq & \exists t \in \textit{trees}(\rho(\chi)), t' \in \textit{trees}(\rho(\chi')).\ t\, \delta\, t' \\ (P_1\ \texttt{or}\ P_2)(\rho) & \triangleq & P_1(\rho)\ \textit{OR}\ P_2(\rho) \\ \texttt{empty}((\chi))(\rho) & \triangleq & \text{if } \rho(\chi) = () \text{ then } \texttt{true} \text{ else } \texttt{false} \\ (\texttt{not}\ P)(\rho) & \triangleq & \textit{NOT}\ P(\rho) \end{array}$$

Figure 6. Predicate evaluation.

and comprises variable comparisons only. We distinguish between *for-variables*, e.g, $\overline{x}$, which are bound by `for` iterations, and *let-variables*, e.g., x, that are instead bound by `let` clauses. The syntax of miniXQuery is shown in Figure 3.

The semantics of the language and the required auxiliary functions are shown in Figures 4 and 5, where ρ is a substitution assigning a forest to each free variable in the query. We make the assumption that each ρ is well-formed, meaning that it always associates a tree t to a for-variable $\overline{x}$ it defines; also, `dos` is a shortcut for descendant-or-self. The semantics of `for` queries is defined via the operator $\prod_{t_1,\ldots,t_n} A(t_i)$, where each t_i is an XML tree, yielding the forest $A(t_1), \ldots, A(t_n)$. In Figure 4 the notation $P(\rho)$ indicates the truth value obtained by evaluating the predicate P under a variable assignment environment ρ, as indicated in Figure 6. In this figure $\textit{trees}(f)$ is the set of all top-level trees of f:

$$\textit{trees}(f) = \{t \mid f = f', t, f''\}$$

All the rest is self explicative.

5. W3C Type System

In this Section we will describe a core version of the W3C XQuery type system. This type system is implemented in many W3C-compliant XQuery implementations and plays a key role in the static analysis of XQuery programs.

5.1 Auxiliary definitions

The W3C inference technique relies on the notion of *prime types*. A prime type obeys the following grammar:

$$\begin{array}{llll} P & ::= & \mathsf{B} & \textit{base type} \\ & & \mid l[T] & \textit{element type} \\ & & \mid P \mid P & \textit{union type} \end{array}$$

A prime type, hence, is a non-empty disjunction of base and element types.

Given a type T and a type environment E, we can extract a prime type from T by using the $\textit{Prime}_E(T)$ function.

$$
\begin{array}{lll}
[\![b]\!]_\rho & \triangleq b & \\
[\![()]\!]_\rho & \triangleq () & \\
[\![\overline{x}]\!]_\rho & \triangleq \rho(\overline{x}) & \\
[\![x]\!]_\rho & \triangleq \rho(x) & \\
[\![Q_1, Q_2]\!]_\rho & \triangleq [\![Q_1]\!]_\rho, [\![Q_2]\!]_\rho & \\
[\![l[Q]]\!]_\rho & \triangleq l[[\![Q]\!]_\rho] & \\
[\![\overline{x}\ \texttt{child} :: \mathit{NodeTest}]\!]_\rho & \triangleq \mathit{childr}([\![\overline{x}]\!]_\rho) :: NodeTest & \\
[\![\overline{x}\ \texttt{dos} :: \mathit{NodeTest}]\!]_\rho & \triangleq \mathit{dos}([\![\overline{x}]\!]_\rho) :: NodeTest & \\
[\![\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2]\!]_\rho & \triangleq [\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho} & \\
[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2]\!]_\rho & \triangleq \prod_{t_1,\ldots,t_n} [\![Q_2]\!]_{\rho, \overline{x} \mapsto t_i} & \text{if } [\![Q_1]\!]_\rho = t_1, \ldots, t_n \\
[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2]\!]_\rho & \triangleq () & \text{if } [\![Q_1]\!]_\rho = () \\
[\![\texttt{let}\ x ::= Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2]\!]_\rho & \triangleq \text{if } P(\rho, x \mapsto [\![Q_1]\!]_\rho) \text{ then } [\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho} \text{ else } () & \\
[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2]\!]_\rho & \triangleq \prod_{t_1,\ldots,t_n} (\text{if } P(\rho, \overline{x} \mapsto t_i) \text{ then } [\![Q_2]\!]_{\rho, \overline{x} \mapsto t_i} \text{ else } ()) & \text{if } [\![Q_1]\!]_\rho = t_1, \ldots, t_n \\
[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2]\!]_\rho & \triangleq () & \text{if } [\![Q_1]\!]_\rho = ()
\end{array}
$$

Figure 4. miniXQuery semantics.

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

Figure 5. Auxiliary functions.

Definition 5.1 *$Prime_E(T)$ is inductively defined as follows:*

$$
\begin{array}{ll}
Prime_E(()) & = () \\
Prime_E(B) & = B \\
Prime_E(T_1, T_2) & = Prime_E(T_1) \mid Prime_E(T_2) \\
Prime_E(l[T]) & = l[T] \\
Prime_E(T_1 \mid T_2) & = Prime_E(T_1) \mid Prime_E(T_2) \\
Prime_E(T*) & = Prime_E(T) \\
Prime_E(T+) & = Prime_E(T) \\
Prime_E(T?) & = Prime_E(T) \\
Prime_E(X) & = Prime_E(E(X))
\end{array}
$$

Observe that, when no prime type can be extracted, this function just returns $()$.

Another important auxiliary function $Quant_E(T)$ returns a symbol in $\{1, +, ?, *\}$ denoting the cardinality of the type sequence of the outermost level of T.

Definition 5.2 **($Quant_E(T)$)** *$Quant_E(T)$ is inductively defined as follows:*

$$
\begin{array}{ll}
Quant_E(()) & = ? \\
Quant_E(B) & = 1 \\
Quant_E(T_1, T_2) & = Quant_E(T_1), Quant_E(T_2) \\
Quant_E(l[T]) & = 1 \\
Quant_E(T_1 \mid T_2) & = Quant_E(T_1) \mid Quant_E(T_2) \\
Quant_E(T*) & = Quant_E(T).* \\
Quant_E(T+) & = Quant_E(T).+ \\
Quant_E(T?) & = Quant_E(T).? \\
Quant_E(X) & = Quant_E(E(X))
\end{array}
$$

,	1	?	+	*
1	+	+	+	+
?	+	*	+	*
+	+	+	+	+
*	+	*	+	*

$\mid$	1	?	+	*
1	1	?	+	+
?	?	?	*	*
+	+	*	+	*
*	*	*	*	*

.	1	?	+	*
1	1	?	+	*
?	?	?	*	*
+	+	*	+	*
*	*	*	*	*

Figure 7. Quantifier composition.

Quantifiers can be composed by using three operators: ‘,’, ‘|’, and ‘.’. Quantifier composition obeys the rules described in Figure 7.

A quantifier and a type can be combined by the . operator as shown below.

$$
\begin{array}{lllllll}
T.1 & = & T & \quad & T.? & = & T? \\
T.+ & = & T+ & \quad & T.* & = & T*
\end{array}
$$

It will be clear from the context when . is used to combine two quantifiers together or a type and a quantifier.

It should be observed that $T \leqslant Prime_E(T).Quant_E(T)$ for any type T and any well-formed environment E.

The function $content()$ is the last auxiliary function we present here; it just extracts the content model of a given element type and it can be trivially lifted to base and union types, as shown by the following definition.

Definition 5.3 ($content()$**)** *The function* $content()$ *is defined as follows:*

$$\begin{array}{rcl} content(()) & \triangleq & () \\ content(B) & \triangleq & () \\ content(l[T]) & \triangleq & T \\ content(T_1 \mid \ldots \mid T_n) & \triangleq & content(T_1) \mid \ldots \mid content(T_n) \end{array}$$

5.2 Judgments and variable environments

The type inference system is able to prove judgments of the form $E; \Gamma \vdash_m Q : T$, where Q is a query, Γ is an environment providing type information about the free variables of Q, and T is an upper bound for all possible values returned by Q[1]. In particular, E, Γ, and Q are intended to be the input arguments of the type inference process, while T is the output type to be inferred.

Definition 5.4 (Variable environment) *A variable environment* Γ *is a list of pairs* $\chi : T$*, where* χ *is a for-variable or a let-variable, and* T *is a type. Variable environments meet the following grammar:*

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

Definition 5.5 (Variable environment variable set) *Given a variable environment* Γ*, we indicate with* $\Gamma Var(\Gamma)$ *the set of all variables defined in* Γ*:* $\Gamma Var(\Gamma) = \{\overline{x} \mid \overline{x} : T \in \Gamma\} \cup \{x \mid x : T \in \Gamma\}$.

A variable environment Γ is well-formed if no variable is defined twice, and if every for-variable $\overline{x}$ (i.e., a variable bound by a `for` clause) is associated to a union of tree types ($l[T']$ or B). Moreover, in the following each time we consider a variable environment Γ for a query Q, we will assume that Γ provides definitions for all free-variables of Q. $WF(E; \Gamma \vdash_m Q : U)$ means that the judgement $E; \Gamma \vdash_m Q : U$ is well-formed, that is: in Γ no variable occurs twice, and each free-variable in Q occurs (is defined) in Γ.

Beyond $E; \Gamma \vdash_m Q : T$, two auxiliary judgments are employed in our type rules. These judgments serve the purpose of keeping rules relatively simple, while allowing for a good level of precision of type inference.

The judgment $E \vdash_m S \leqslant T$ is used to test whether S is a subtype of T. The judgment $E \vdash_m T' \rightarrow_{\textit{NodeTest}} U$ is used to restrict the content type T' to tree types with structure satisfying *NodeTest*. Rules to prove judgments $E \vdash_m T' \rightarrow_{\textit{NodeTest}} U$ are shown in Figure 8, and their meaning is stated in the following lemma.

Lemma 5.6 (Type Filtering Checking) *For any* T*:*

$$E \vdash_m T \rightarrow_{\textit{NodeTest}} U \Leftrightarrow [\![U]\!]_E = \{f :: \textit{NodeTest} \mid f \in [\![T]\!]_E\}$$

5.3 Type rules

The type rules for the W3C type system are shown in Figures 9 and 10. Rules in Figure 9 are shared with the μXQ+ type system, while the rules in Figure 10 are specific to the W3C type system. For reasons of space, we describe here only the most important ones.

[1] The m subscript in the judgments distinguishes the miniXQuery W3C type rules from those specific of μXQ+, which are labeled by μ.

$$\textsc{(MatchAnyFilt)} \quad \frac{}{E \vdash_m T \rightarrow_{\texttt{node()}} T}$$

$$\textsc{(MatchTextFilt)} \quad \frac{}{E \vdash_m B \rightarrow_{\texttt{text()}} B}$$

$$\textsc{(MatchLabFilt)} \quad \frac{}{E \vdash_m l[T] \rightarrow_l l[T]}$$

$$\textsc{(NoMatchLabFilt)} \quad \frac{T = \mathsf{B} \ \vee \ T = n[T']}{E \vdash_m T \rightarrow_l ()}$$

$$\textsc{(ForestFilt)} \quad \frac{E \vdash_m T \rightarrow_{\textit{NodeTest}} T' \qquad E \vdash_m U \rightarrow_l U'}{E \vdash_m T, U \rightarrow_{\textit{NodeTest}} T', U'}$$

$$\textsc{(StarFilt)} \quad \frac{E \vdash_m T \rightarrow_{\textit{NodeTest}} U}{E \vdash_m T* \rightarrow_{\textit{NodeTest}} U*}$$

$$\textsc{(PlusFilt)} \quad \frac{E \vdash_m T \rightarrow_{\textit{NodeTest}} U}{E \vdash_m T+ \rightarrow_{\textit{NodeTest}} U+}$$

$$\textsc{(OptFilt)} \quad \frac{E \vdash_m T \rightarrow_{\textit{NodeTest}} U}{E \vdash_m T? \rightarrow_{\textit{NodeTest}} U?}$$

$$\textsc{(UnionFilt)} \quad \frac{E \vdash_m T \rightarrow_{\textit{NodeTest}} T' \qquad E \vdash_m U \rightarrow_{\textit{NodeTest}} U'}{E \vdash_m T \mid U \rightarrow_{\textit{NodeTest}} T' \mid U'}$$

$$\textsc{(VarFilt)} \quad \frac{E \vdash_m E(X) \rightarrow_{\textit{NodeTest}} U}{E \vdash_m X \rightarrow_{\textit{NodeTest}} U}$$

$$\textsc{(EmptyFilt)} \quad \frac{}{E \vdash_m () \rightarrow_{\textit{NodeTest}} ()}$$

Figure 8. Filtering rules.

Rule (TypeFor) describes the behaviour of the type inference system when a `for`-iteration is visited. The output type is computed as follows. The rule first computes the inferred type for Q_1; from this type a *prime* type is extracted by the function $Prime_E(T_1)$, which returns the union of the uppermost base or tree types in T_1 and it can be computed in linear space and time. $Prime_E(T_1)$ is, then, bound to $\overline{x}$ in the variable environment and used to infer the output type T_2, which is further refined by the application of a quantifier in $\{1, ?, +, *\}$ ($T_2 \,.\, Quant_E(T_1)$).

Rule (TypeDos) applies to `dos` selectors and infers a type for a $\overline{x}$ `dos` :: *NodeTest* filter. The rule essentially traverses the parse tree of T and, at each step, collects the prime types it encounters. The premise $E \vdash_m Prime_E(T_{n+1}) \leqslant Prime_E(T_1) \mid \ldots \mid Prime_E(T_n)$ is just a formal way to express the termination of the search in the parse tree and it does not involve any real subtyping operation. The rule returns a type consisting of the star-guarded union of the types collected during the exploration.

It should be observed that, while not algorithmic, these rules are deterministic. Hence, for any well-formed query Q, type environment E, and variable environment Γ, the system can infer only one type T such that $E; \Gamma \vdash_m Q : T$.

6. μXQ+ Type System

The μXQ+ specific type rules are shown in Figures 11 and 12.

$$\frac{\begin{array}{c}\text{(TYPEFOR)}\\ E;\Gamma \vdash_m Q_1 : T_1 \\ E;\Gamma,\ \overline{x} : Prime_E(T_1) \vdash_m Q_2 : T_2\end{array}}{E;\Gamma \vdash_m \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : T_2.Quant_E(T_1)}$$

$$\frac{\begin{array}{c}\text{(TYPEFORWHERE)}\\ E;\Gamma \vdash_m Q_1 : T_1 \\ E;\Gamma,\ \overline{x} : Prime_E(T_1) \vdash_m Q_2 : T_2\end{array}}{E;\Gamma \vdash_m \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2 : (T_2.Quant_E(T_1))?}$$

Figure 10. miniXQuery specific type rules.

The μXQ+ type system differs from the W3C one for the handling of `for`-iterations and `descendant-or-self` selectors. Given a query $\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2$, Rule (XTYPEFOR) first infers a type T_1 for Q_1. Unlike the W3C type system, where T_1 is transformed in prime form and then bound to $\overline{x}$, in the μXQ+ type system T_1 is decomposed in its basic components, and Q_2 is analyzed separately for each component of T_1. This different approach is illustrated in the following example.

Example 6.1 Consider the following basic query:

$\texttt{for}\ \overline{x}\ \texttt{in}\ l[b], m[b]$
$\texttt{return}\ \overline{x}$

Rule (TYPEFOR) first infers a type $T_1 = l[\mathsf{B}], m[\mathsf{B}]$ for $l[b], m[b]$. The variable environment is, then, enriched with the binding between $\overline{x}$ and $Prime_E(T_1) = l[\mathsf{B}] \mid m[\mathsf{B}]$. As $Q_2 = \overline{x}$, the output type is just $(l[\mathsf{B}] \mid m[\mathsf{B}])+$, where $+$ is introduced by $Quant_E(T_1)$.

In the μXQ+ type system the inference process is different. Once T_1 has been inferred, Rule (TYPEINCONC) is used to infer a type for Q_2; this rule iterates over $T_1 = l[\mathsf{B}], m[\mathsf{B}]$ and returns $T_2 = l[\mathsf{B}], m[\mathsf{B}]$, without any additional $+$ or $*$ quantifier. ■

7. Precision Analysis

In this section we will compare the W3C type system and that of μXQ+ in terms of precision of the inferred types. First, we will show that the type system of μXQ+ is more precise, as it always generates a subtype of the type inferred by the W3C one. Then, we will identify some restrictions that make the two type systems equivalent in terms of precision. Finally, we will see that, if some properties are satisfied by the input type, precision can be dramatically improved.

7.1 Precision

In Example 6.1 we found that, for a very basic query iterating on a sequence of element types, the type systems being studied here behave quite differently. In particular, the W3C type system returned a type of the form $(A_1 \mid \ldots \mid A_n)+$, where A_i is an element type in the input sequence, while the μXQ+ type system inferred $A_1, \ldots, A_n$. It is straightforward to see that the latter type is a *subtype* of the former type.

This situation is not an exception. Indeed, the type inferred by μXQ+ for a query Q is always a subtype of the type inferred by the W3C system for the same query, as stated by the following theorem.

Theorem 7.1 *For each well formed* (Γ, E, Q)*:*

$$E;\Gamma \vdash_m Q : T_{w3c} \ \wedge\ E;\Gamma \vdash_\mu Q : T_\mu \ \Rightarrow\ T_\mu \leqslant T_{w3c}$$

This result tells us that the μXQ+ type system is a good and safe choice for improving the precision of inferred types, since its types are at least as precise as those returned by the W3C system.

Of course, there are several situations where the two type systems infer exactly the same type. These cases depend on both the structure of the query and the properties of the input type environment. In particular, we can identify three properties that must be satisfied to ensure the equality of the inferred types.

Definition 7.2 (Tagged-path queries) *A query Q is* tagged-path *if Q only uses XPath steps of the form $\overline{x}$* `child` :: *NodeTest or $\overline{x}$* `dos` :: *NodeTest with NodeTest $\neq$* `node`().

A tagged-path query, hence, cannot use the `self` axis nor it can exploit the `node()` node test. For instance, the query of Example 6.1 is not tagged-path, as it uses the `self` axis.

Definition 7.3 (One-type environments) *Let* (Γ, E) *be well-formed. Then,* (Γ, E) *is* one-type *if, for each $\chi : T \in \Gamma$ and for each label m, there exists a type T' such that:*

$$\{m[U] \mid T \rightarrow_e^E m[U]\} \subseteq \{m[T']\}$$

In other words, in a one-type environment no type can contain multiple element types with the same label and different content. This property is enjoyed by DTDs, and reflects a very common scenario.

The final restriction is inspired by the notion of conflict-free types described in [8].

Definition 7.4 (Conflict-free regular expression) *A regular expression r is* conflict-free *if for each subexpression $(s \mid t)$ or (s, t): $S(s) \cap S(t) = \emptyset$, where $S(p)$ is the set of all symbols appearing in a regular expression p.*

Definition 7.5 ($RegExp_E(T)$)

$$\begin{array}{lcl}
RegExp_E(()) & = & () \\
RegExp_E(\mathsf{B}) & = & \mathsf{B} \\
RegExp_E(l[T]) & = & l \\
RegExp_E(T*) & = & RegExp_E(T)* \\
RegExp_E(T+) & = & RegExp_E(T)+ \\
RegExp_E(T?) & = & RegExp_E(T)? \\
RegExp_E(X) & = & RegExp_E(E(X)) \\
RegExp_E(T_1, T_2) & = & RegExp_E(T_1), RegExp_E(T_2) \\
RegExp_E(T_1 \mid T_2) & = & RegExp_E(T_1) \mid RegExp_E(T_2)
\end{array}$$

Definition 7.6 (Conflict-free environment) *Let* (Γ, E) *be well-formed. Then,* (Γ, E) *is* conflict-free *if for each $X = U \in E$ we have that $U = l[T]$ and $RegExp_E(T)$ is* conflict-free.

We can now state a theorem that sheds light on a case of equality of type inference.

(TypeEmpty)
$$\frac{\mathit{WF}(E;\Gamma \vdash_m () : ())}{E;\Gamma \vdash_m () : ()}$$

(TypeAtomic)
$$\frac{\mathit{WF}(E;\Gamma \vdash_m b : \mathsf{B})}{E;\Gamma \vdash_m b : \mathsf{B}}$$

(TypeVarLet)
$$\frac{x : T \ \in \ \Gamma \quad \mathit{WF}(E;\Gamma \vdash_m x : T)}{E;\Gamma \vdash_m x : T}$$

(TypeVarFor)
$$\frac{\overline{x} : T \ \in \ \Gamma \quad \mathit{WF}(E;\Gamma \vdash_m \overline{x} : T)}{E;\Gamma \vdash_m \overline{x} : T}$$

(TypeElem)
$$\frac{E;\Gamma \vdash_m Q : T}{E;\Gamma \vdash_m l[Q] : l[T]}$$

(TypeForest)
$$\frac{E;\Gamma \vdash_m Q_1 : T_1 \quad E;\Gamma \vdash_m Q_2 : T_2}{E;\Gamma \vdash_m Q_1, Q_2 : T_1, T_2}$$

(TypeLet)
$$\frac{\begin{array}{l} E;\Gamma \vdash_m Q_1 : T_1 \\ E;\Gamma,\ x : T_1 \vdash_m Q_2 : U \end{array}}{E;\Gamma \vdash_m \texttt{let}\ x := Q_1\ \texttt{return}\ Q_2 : U}$$

(TypeLetWhere)
$$\frac{\begin{array}{l} E;\Gamma \vdash_m Q_1 : T_1 \\ E;\Gamma,\ x : T_1 \vdash_m Q_2 : U \end{array}}{E;\Gamma \vdash_m \texttt{let}\ x ::= Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2 : U?}$$

(TypeChildNodeTest)
$$\frac{\begin{array}{l} \mathit{WF}(E;\Gamma \vdash_m \overline{x}\ \texttt{child} :: \mathit{NodeTest} : U) \\ \overline{x} : T \in \Gamma \\ (T = T'_1 \mid \ldots \mid T'_n)\ \wedge (T'_i = m_i[T''_i] \vee T'_i = B \vee T'_i = ()) \\ E \vdash_m \mathit{content}(T) \rightarrow_{\mathit{NodeTest}} U \end{array}}{E;\Gamma \vdash_m \overline{x}\ \texttt{child} :: \mathit{NodeTest} : U}$$

(TypeDOS)
$$\frac{\begin{array}{l} \mathit{WF}(E;\Gamma \vdash_m \overline{x}\ \texttt{dos} :: \mathit{NodeTest} : U) \\ \overline{x} : T \in \Gamma \\ (T = T'_1 \mid \ldots \mid T'_n)\ \wedge (T'_i = m_i[T''_i] \vee T'_i = B \vee T'_i = ()) \\ E;\Gamma \vdash_m \overline{x}\ \mathit{child} : T_1 \\ E;\Gamma, \overline{x} : \mathit{Prime}_E(T_1) \vdash_m \overline{x}\ \mathit{child} : T_2 \\ E;\Gamma, \overline{x} : \mathit{Prime}_E(T_2) \vdash_m \overline{x}\ \mathit{child} : T_3 \\ \ldots \\ E;\Gamma, \overline{x} : \mathit{Prime}_E(T_n) \vdash_m \overline{x}\ \mathit{child} : T_{n+1} \\ E \vdash_m \mathit{Prime}_E(T_{n+1}) \leqslant \mathit{Prime}_E(T_1) \mid \ldots \mid \mathit{Prime}_E(T_n) \\ U' = (\mathit{Prime}_E(T) \mid \mathit{Prime}_E(T_1) \mid \ldots \mid \mathit{Prime}_E(T_n))* \\ E \vdash_m U' \rightarrow_{\mathit{NodeTest}} U \end{array}}{E;\Gamma \vdash_m \ \overline{x}\ \texttt{dos} ::\ \mathit{NodeTest} : U}$$

Figure 9. Common type rules.

(TypeInEmpty)
$$\frac{\mathit{WF}(E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ (\,)\ \rightarrow\ Q : (\,))}{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ (\,)\ \rightarrow\ Q : (\,)}$$

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

(TypeInEl)
$$\frac{E;\Gamma,\ \overline{x} : m[T] \vdash_\mu Q : U}{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ m[T]\ \rightarrow\ Q : U}$$

(TypeInConc)
$$\frac{\begin{array}{l} E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q : T'_1 \\ E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_2\ \rightarrow\ Q : T'_2 \end{array}}{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_1, T_2\ \rightarrow\ Q : T'_1, T'_2}$$

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

(TypeInPlus)
$$\frac{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q : U}{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T+\ \rightarrow\ Q : U+}$$

(TypeInOpt)
$$\frac{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q : U}{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T?\ \rightarrow\ Q : U?}$$

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

(TypeInUnion)
$$\frac{\begin{array}{l} E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q : T'_1 \\ E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_2\ \rightarrow\ Q : T'_2 \end{array}}{E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_1 \mid T_2\ \rightarrow\ Q : T'_1 \mid T'_2}$$

Figure 11. Case analysis type rules.

(XTypeFor)
$$\frac{E;\Gamma \vdash_\mu Q_1 : T_1 \quad E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : T_2}{E;\Gamma \vdash_\mu \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : T_2}$$

(XTypeForWhere)
$$\frac{E;\Gamma \vdash_\mu Q_1 : T_1 \quad E;\Gamma \vdash_\mu \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : T_2}{E;\Gamma \vdash_\mu \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2 : (T_2)?}$$

Figure 12. μXQ+ specific type rules.

Theorem 7.7 (Equality) *If* $(\Gamma,\ E)$ *is one-type and conflict-free, and* Q *is tagged-path, then* $E;\Gamma \vdash_m Q : T$ *and* $E;\Gamma \vdash_\mu Q : T'$*, where* $[\![T]\!]_E = [\![T']\!]_E$.

It is interesting to observe that all these conditions are necessary to guarantee the equality of the inferred types.

Example 7.8 Consider the following query, where $\overline{db}$ has type $T = m[\mathsf{B}] \mid m[l[\mathsf{B}], n[\mathsf{B}], l[\mathsf{B}]]$.

```
for $\overline{x}$ in $\overline{db}$ child :: $l$
return $\overline{x}$
```

This query is tagged-path; however, the environments are not one-type and conflict-free. Rule (TypeChildNodeTest) computes $content(T) = \mathsf{B} \mid (l[\mathsf{B}], n[\mathsf{B}], l[\mathsf{B}])$. $content(T)$ is then filtered by using the rules of Figure 8, and $T_1 = () \mid l[\mathsf{B}], l[\mathsf{B}] = (l[\mathsf{B}], l[\mathsf{B}])?$ is returned as type for the navigational step.

Rule (TypeFor) binds $Prime_E(T_1) = l[\mathsf{B}]$ to $\overline{x}$ and returns $T_{w3c} = (l[\mathsf{B}]?+) = l[\mathsf{B}]*$ as result type.

Rule (XTypeFor), instead, navigates over $(l[\mathsf{B}], l[\mathsf{B}])?$ and returns $T_\mu = (l[\mathsf{B}], l[\mathsf{B}])?$ as result type.

As expected, $[\![T_{w3c}]\!]_E \neq [\![T_\mu]\!]_E$.

■

Example 7.9 Consider the query of the previous example and assume that $\overline{db}$ has type $T = m[l[\mathsf{B}], n[\mathsf{B}], l[\mathsf{B}]]$.

In this case, the type environments are one-type, but they fail to meet the conflict-freedom restriction.

Rule (TypeChildNodeTest) computes $content(T) = l[\mathsf{B}], n[\mathsf{B}], l[\mathsf{B}]$. $content(T)$ is then filtered by using the rules of Figure 8, and $T_1 = l[\mathsf{B}], l[\mathsf{B}]$ is returned as type for the navigational step.

Rule (TypeFor) binds $Prime_E(T_1) = l[\mathsf{B}]$ to $\overline{x}$ and returns $T_{w3c} = l[\mathsf{B}]+$ as result type.

Rule (XTypeFor), instead, navigates over $l[\mathsf{B}], l[\mathsf{B}]$ and returns $T_\mu = l[\mathsf{B}], l[\mathsf{B}]$ as result type.

As expected, the inferred types have different semantics. ■

Example 7.10 Consider now the query of the examples above, where $\overline{db}$ has type $T = m[l[\mathsf{B}], n[\mathsf{B}]]$. Unlike the previous examples, T is conflict-free, so we expect that the two type systems return the same type.

In this case, Rule (TypeChildNodeTest) returns $T_1 = l[\mathsf{B}]$. Hence, Rule (TypeFor) returns $T_{w3c} = l[\mathsf{B}]$ as result type, since $Quant_E(T_1) = 1$.

Rule (XTypeFor), instead, navigates over $l[\mathsf{B}]$ and returns $T_\mu = l[\mathsf{B}]$ as result type, as we expected. ■

As shown in the following example, situations similar to those described in the previous examples are common in real-world scenarios.

Example 7.11 Consider the following query on the well known XMark DTD [16], returning a `region` element for each region, containing the name and the number of items of the region; we assume that $\overline{x}$ is bound to the root element of an XMark instance; we use the XQuery function $name()$ returning the tag name of an XML element, and the XQuery counting function $count()$; we assume both functions returning values of type B.

```
for $\overline{y}$ in $\overline{x}$ child :: $regions$
return for $\overline{z}$ in $\overline{y}$ child :: $node()$
        return region[
                        name[$name(\overline{z})$]
                        total[$count(\overline{z}$ child :: $item)$]]
```

In the XMark DTD, the content type of `regions` is the sequence type $(africa, asia, australia, europe, namerica, samerica)$. As a consequence, the W3C type system infers the following type: $(region[name[\mathsf{B}], total[\mathsf{B}]])+$.

However, the exact type for for this query is $region[C], region[C], region[C], region[C], region[C], region[C]$, with $C = name[\mathsf{B}],\ total[\mathsf{B}]$. Actually this type is inferred by the μXQ+ type system, thanks to the type case analysis performed on `for`-iterations. ■

In the light of the analysis made up to now, and inspired by Theorem 7.7, we can design a new type system which mixes up advantages of the two systems being compared here. We write $E;\Gamma \vdash_H Q : T$ to indicate that the type T has been inferred from Q, Γ and E in the hybrid system. The rules of the hybrid systems are essentially those of the μXQ+ type system, with the only difference that `for`-iterations are typed by the following rules (`for-where` queries are typed in a similar way).

$$\text{(TypeForHybrid1)}$$
$$\frac{\begin{array}{l} Q_1 \text{ is tagged-path } \wedge\ Q_2 \text{ is tagged-path} \\ (E,\Gamma) \text{ is one-type and conflict-free} \\ E;\Gamma \vdash_H Q_1 : T_1 \\ E;\Gamma,\ \overline{x} : Prime_E(T_1) \vdash_H Q_2 : T_2 \end{array}}{E;\Gamma \vdash_H \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : T_2.Quant_E(T_1)}$$

$$\text{(TypeForHybrid2)}$$
$$\frac{\begin{array}{l} Q_1 \text{ is not tagged-path } \vee\ Q_2 \text{ is not tagged-path} \vee \\ (E,\Gamma) \text{ is not one-type and conflict-free} \\ E;\Gamma \vdash_H Q_1 : T_1 \\ E;\Gamma \vdash_H \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : T_2 \end{array}}{E;\Gamma \vdash_H \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : T_2}$$

The resulting hybrid system has the same precision of the μXQ+ type system, but lower time consumption since the costly μXQ+ case analysis is invoked only if strictly necessary

Theorem 7.12 (Precision of the Hybrid system) *For each well formed* (Γ, E, Q)*, if* $E;\Gamma \vdash_\mu Q : T_\mu$ *and* $E;\Gamma \vdash_H Q : T_H$ *then* $[\![T_\mu]\!]_E = [\![T_H]\!]_E$.

7.2 Extensions

We found in the previous section that, under some specific circumstances, the two type systems return the same type for a given query.

In this section we will see that some restrictions on the type environment allow for an improvement of inference precision without any significant performance penalty.

7.2.1 Non-recursive Types

When the input type contains no recursive type, we can significantly improve the precision of the inferred type for `descendant -or-self` steps without affecting the computational complexity.

Indeed, given a non-recursive type T, we can define type inference for the descendant-or-self axis through the following rule:

$$\text{(TypeDOSNoRec)}$$
$$\frac{\begin{array}{l} \overline{x} : T \in \Gamma\ \wedge\ (T = l_1[T_1'] \mid l_2[T_2'] \mid \ldots \mid l_n[T_n'] \vee T = \mathsf{B}) \\ E \vdash_m dos_E(\Gamma(\overline{x})) \rightarrow_{NodeTest} U \end{array}}{E;\Gamma \vdash_m\ \overline{x}\ \texttt{dos} :: \ NodeTest : U}$$

where the auxiliary function $dos_E()$ mimic `dos` semantics, and is defined as follows.

Definition 7.13

$$\begin{array}{rcl} dos_E(()) & = & () \\ dos_E(B) & = & B \\ dos_E(l[T]) & = & l[T], dos_E(T) \\ dos_E(T?) & = & dos_E(T)? \\ dos_E(T*) & = & dos_E(T)* \\ dos_E(T+) & = & dos_E(T)+ \\ dos_E(T, U) & = & dos_E(T), dos_E(U) \\ dos_E(T \mid U) & = & dos_E(T) \mid dos_E(U) \\ dos_E(X) & = & dos_E(E(X)) \end{array}$$

Example 7.14 Consider the following query, where $\overline{db}$ has type $a[b[], c[()]+]$:

for $\overline{x}$ in $\overline{db}$ dos :: b
return $\$x$

By applying Rule (TYPEDOS), the system infers the type $b[()]*$ for the navigational step; by Rule (TYPEFOR), hence, the result type is just $b[()]*$.

Instead, if we apply the $dos_E()$ function to the type of $\overline{db}$, we have that $dos_E(a[b[()], c[()]+]) =$

$$\begin{array}{rl} & a[b[()], c[()]+], dos_E(b[()], c[()]+) \\ = & a[b[()], c[()]+], dos_E(b[()]), dos_E(c[()]+) \\ = & a[b[()], c[()]+], b[()], (), (c[()], ())+ \end{array}$$

Then, by using Rule (TYPEDOSNOREC), the navigational step $\overline{db}$ dos :: b is typed with $b[()]$ and the result type is $b[()].1 = b[()]$, which is a much more precise type. ∎

The following theorem states the correctness of this approach.

Theorem 7.15 *For any non-recursive type T defined in E:*

$$dos_E(T) = U \quad \Rightarrow \quad \forall f \in [\![T]\!]_E.\ dos(f) \in [\![U]\!]_E$$

Proof. Trivial. ∎

It is quite easy to see that Rule (TYPEDOSNOREC) always infers a subtype of that inferred by Rule (TYPEDOS).

As it can be noted from Example 7.14, $dos_E()$ satisfies the following property.

Property 7.16 *If $dos_E(T) = U$ and U contains a $*$ type at its top level, then there exists $f \in [\![T]\!]_E$ such that $dos(f) = ()$, and for each natural k there exists $f \in [\![T]\!]_E$ such that $|dos(f)| > k$.*

This property is crucial, as it states that $dos_E()$ does not introduce unnecessary $*$ type operators. Hence, unlike the general rule of the W3C system, this rule does not introduce unnecessary approximations.

It is easy to see that the following theorem holds.

Theorem 7.17 *$dos_E(T)$ can be evaluated in time $O((|T||E|)^2)$.*

8. Complexity Analysis

In this section we will analyze the complexity of the XQuery type inference in both the W3C and the μXQ+ type systems. We will prove two main results here. First, we will show that the W3C type inference algorithm may return exponentially larger output types when `let` clauses are nested in a particular way, and that it resorts to polynomial time complexity when no `let` clauses are present. Then, we will see that μXQ+ type inference system is subject to combinatorial explosion even in presence of `for` clauses only, hence leading to a worst case exponential time complexity.

8.1 Assumptions

To analyze the complexity of the type inference algorithms, we need a few assumptions about the complexity model and the basic data structures being used.

To study the complexity of our algorithms we base our analysis on the RAM (*Random Access Machine*) model. As usual, we assume that read/write operations on the RAM memory and on the input/output device are performed in unit time, as well as comparisons and arithmetic operations. Under these hypotheses, the type inference algorithms can be simulated by RAM programs with the same asymptotic complexity.

We assume here to represent E and Γ through arrays. As E is not altered by the inference process, it can be created and initialized at schema creation time, without any significant impact on the complexity of type inference. Γ, instead, is a dynamic structure that can be created and initialized at query parsing time with a $O(|\Gamma Var(\Gamma)|)$ extra cost.

We restrict our analysis to *non-recursive* queries, where one cannot navigate or output the result of a nested query, e.g., we exclude queries like

```
for $\overline{x}$ in $\overline{db}$ child :: $a$ return
    for $\overline{y}$ in { for $\overline{z}$ in $\overline{db}$ dos :: $c$
            where $\overline{x} = \overline{z}$
            return w[$\overline{z}$] } return
        for $\overline{u}$ in $\overline{y}$ child :: $d$ return h[$\overline{u}$]
```

This restriction is met by many queries used in practice and is also enforced in data integration systems using XQuery as language for expressing queries and transformations (see [10]).

8.2 Preliminary definitions

In the following we will use the terms *query size* and *type size* to denote the size of the corresponding abstract syntax tree, as shown in the following definitions.

Definition 8.1 (Query size) *Given a query Q, $|Q|$ is inductively defined as follows:*

$$\begin{array}{ll} |Q| = 1 & |b| = 1 \\ l[Q] = 1 + |Q| & |Q_1, Q_2| = |Q_1| + |Q_2| \\ |\overline{x}\ \texttt{child} :: NodeTest| = 3 & |\overline{x}\ \texttt{dos} :: NodeTest| = 3 \end{array}$$

$$\begin{array}{l} |\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2| = 2 + |Q_1| + |Q_2| \\ |\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2| = 2 + |Q_1| + |Q_2| \\ |\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2| = 2 + |Q_1| + |Q_2| + |P| \\ |\texttt{let}\ x ::= Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2| = 2 + |Q_1| + |Q_2| + |P| \end{array}$$

$$\begin{array}{ll} |\texttt{true}| = 1 & |\texttt{false}| = 1 \\ |P_1\ \texttt{or}\ P_2| = 1 + |P_1| + |P_2| & |\texttt{not}\ P| = 1 + |P| \\ |(P)| = |P| & |\texttt{empty}(\chi)| = 1 + |\chi| \\ |\chi_1\ \delta\ \chi_2| = |\chi_1| + |\chi_2| + 1 & |\chi| = 1 \end{array}$$

Definition 8.2 (Type size) *Given a type T, $|T|$ is inductively defined as follows:*

$$\begin{array}{ll} |()| = 1 & |B| = 1 \\ |l[T]| = 1 + |T| & |T_1, T_2| = 1 + |T_1| + |T_2| \\ |T_1 \mid T_2| = 1 + |T_1| + |T_2| & |T?| = 1 + |T| \\ |T+| = 1 + |T| & |T*| = 1 + |T| \\ |X| = 1 & \end{array}$$

Definition 8.3 (Type environment size) *Given a type environment $E = X_1 = T_1; \ldots; X_n = T_n$, $|E| = \Sigma_{i=1}^{n} |T_i|$ if $E \neq \emptyset$, and $|E| = 1$ if $E = \emptyset$.*

Definition 8.4 (Variable environment size) *Given a variable environment $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, $|\Gamma| = \Sigma_{i=1}^{n} |T_i|$ if $\Gamma \neq \emptyset$, and $|\Gamma| = 1$ if $\Gamma = \emptyset$.*

We will also use the notion of *independent variables*, which is defined as follows.

Definition 8.5 *Given a query Q, $Var(Q)$ is the set of all variables occurring in Q.*

Definition 8.6 *Given a query Q and a for-variable $\overline{x} \in Var(Q)$:*

$$\begin{aligned} dep_Q(\overline{x}) \quad = \quad & \{\overline{y} \mid \texttt{for } \overline{x} \texttt{ in } \overline{y} \texttt{ child} :: \textit{NodeTest is in } Q\} \cup \\ & \{\overline{y} \mid \texttt{for } \overline{x} \texttt{ in } \overline{y} \texttt{ dos} :: \textit{NodeTest is in } Q\} \cup \\ & \{\chi \mid \texttt{for } \overline{x} \texttt{ in } \chi \textit{ is in } Q\} \end{aligned}$$

Definition 8.7 *Given a query Q and a let-variable $x \in Var(Q)$:*

$$\begin{aligned} dep_Q(x) \quad = \quad & \{\overline{y} \mid \texttt{let } x ::= \overline{y} \texttt{ child} :: \textit{NodeTest is in } Q\} \cup \\ & \{\overline{y} \mid \texttt{let } x ::= \overline{y} \texttt{ dos} :: \textit{NodeTest is in } Q\} \cup \\ & \{\chi \mid \texttt{let } x ::= \chi \textit{ is in } Q\} \end{aligned}$$

Definition 8.8 *Given a query Q and a variable $\chi \in Var(Q)$, $dep^*_Q(\chi) = \{\chi' \mid \chi' \in dep_Q(\chi) \vee \exists \chi'' \in dep_Q(\chi).\chi' \in dep^*_Q(\chi'')\}$.*

Definition 8.9 *We say that two variables χ_1, χ_2 are* independent *wrt a query Q if and only $dep^*_Q(\chi_1) \cap dep^*_Q(\chi_2) = \emptyset$.*

Definition 8.10 *Given a query Q, $\{\chi_1, \ldots, \chi_n\} \subseteq Var(Q)$ is a set of independent variables in Q if $\forall i, j = 1, \ldots, n.dep^*_Q(\chi_i) \cap dep^*_Q(\chi_j) = \emptyset$.*

8.3 Complexity of auxiliary functions

A preliminary step in the analysis of the complexity of the W3C type inference system is the analysis of the space and time complexity of auxiliary functions.

Space complexity The following lemmas show the space complexity of the auxiliary functions used in the W3C type system. Due to the presence of type variables to be unfolded, $Prime_E(T)$ and type filtering may return types bigger than the input type. As the proofs are trivial, we omit them.

Lemma 8.11 $|Prime_E(T)| \in O(|T||E|)$.

Lemma 8.12 $|Quant_T(E)| \in O(1)$.

Lemma 8.13 $|content(T)| \in O(|T|)$.

Lemma 8.14 *If $E \vdash_m T \to_l U$, then $|U| \in O(|E||T|)$.*

Time Complexity Once we identified the space complexity of the auxiliary functions, we can analyze their time complexity. In the following, we will use $\mathcal{C}(f)$ to denote the cost of evaluating a given function f according to the RAM complexity model. We omit the proofs of the lemmas, as they are trivial.

Lemma 8.15 $\mathcal{C}(content(T)) \in O(|T|)$.

Lemma 8.16 $\mathcal{C}(E \vdash_m T \to_l U) \in O(|E||T|)$.

Lemma 8.17 $\mathcal{C}(Prime_E(T)) \in O(|E||T|)$

Lemma 8.18 $\mathcal{C}(Quant_E(T)) \in O(|E||T|)$

8.4 Complexity of W3C type rules

To understand the complexity of the W3C type inference system, we must first analyze a few properties of the navigational operators.

Lemma 8.19 (Complexity of Rule (TYPECHILDNODETEST))
$\mathcal{C}(E; \Gamma \vdash_m \overline{x} \texttt{ child} :: \textit{NodeTest} : U) \in O(|\Gamma(\overline{x})||E|)$.

Proof. All operators inside Rule (TYPECHILDNODETEST) have $O(1)$ complexity, except for $E \vdash_m content(T) \to_{\textit{NodeTest}} U$. This operation requires the system to compute $content(T)$ and to filter its result according to *NodeTest*. By Lemma 8.15, $content(T)$ (where $T = \Gamma(\overline{x})$) can be computed in time $O(|\Gamma(\overline{x})|)$. By Lemma 8.16, hence, Rule (TYPECHILDNODETEST) can be evaluated in $O(|\Gamma(\overline{x})||E|)$ time. ■

The following describes the size of the type inferred by Rule (TYPEDOS).

Lemma 8.20 *If $E; \Gamma \vdash_m \overline{x} \texttt{ dos} :: \textit{NodeTest} : U$, then $|U| \in O((|E||\Gamma(\overline{x})|)^2)$.*

Proof. As *NodeTest* = `node()` is the least selective node test, we analyze the size of the U when *NodeTest* = `node()`.

By the element-guardedness of type environment, we know that each type variable is unfolded only once; as a consequence, the typing rule must explore a parse tree of size $O(|E||\Gamma(\overline{x})|)$, the worst case happening when $\Gamma(\overline{x}) = l_1[X_1] \mid \ldots \mid l_n[X_n]$.

Each subtree in the parse tree is added to the result, hence $|U| \in O((|E||\Gamma(\overline{x})|)^2)$. ■

As stated by Lemma 8.20, the typing of `descendant-or-self` may quadratically increase the size of the inferred type. This quadratic increase, however, does not lead to a combinatorial explosion, as stated by the following lemma.

Lemma 8.21 *If $E; \Gamma \vdash_m \overline{x} \texttt{ dos} :: \textit{NodeTest} : U$ and $\Gamma(\overline{x}) = C'[C[T] \mid T]$, where $C[\cdot]$ and $C'[\cdot]$ are type contexts, then $E; \Gamma, \overline{x} : C'[C[T]] \vdash_m \overline{x} \texttt{ dos} :: \textit{NodeTest} : U$.*

Proof. Trivial. ■

The previous lemma states that repeated subtrees give no contribution to the output type of a `descendant-or-self` selector. By exploiting this lemma, we can prove the following result.

Lemma 8.22 *Let Q be a non-recursive query, let $\chi_0, \ldots, \chi_n$ the variables bound by Q. If $S = \{\chi_{i_0}, \ldots, \chi_{i_k}\}$ is the set of all the independent variables in Q, then, for each variable χ_i, $|\Gamma(\chi_i)| \in O((\Upsilon)^2)$, where $\Upsilon = max\{\Gamma(\chi_{i_0}), \ldots, \Gamma(\chi_{i_k})\}$.*

This lemma shows that, while a `descendant-or-self` operation can quadratically increase the size of the inferred type of a variable, no exponential explosion is possible.

This result leads to the following corollary.

Corollary 8.23 *Let $E; \Gamma \vdash_m Q : T$. Let Γ' the variable environment obtained from Γ by inferring a type for each variable $\chi_i \in Var(Q)$ such that $(\chi_i : T_i) \notin \Gamma$.Then, $|\Gamma'| \leqslant \Upsilon^2 n$, where $n = |Var(Q)|$.*

This corollary states that, during the type inference process for Q, the size of the variable environment cannot exceed $n\Upsilon^2$.

Given these results, we can now prove one of the main results of this section.

Lemma 8.24 (Complexity of Rule (TYPEDOS))
$\mathcal{C}(E; \Gamma \vdash_m \overline{x} \texttt{ dos} :: \textit{NodeTest} : U) \in O((|\Gamma(\overline{x})||E|)^3)$.

Proof. To prove the thesis it is necessary to rewrite Rule (TYPEDOS) in a more algorithmic fashion, as shown below.

$$\frac{\begin{array}{l} \overline{x} : T \in \Gamma \ \wedge \ (T = m_1[T'_1] \mid m_2[T'_2] \mid \ldots \mid m_n[T'_n] \vee T = \mathsf{B}) \\ U' = BFSPrime(\overline{x}, \Gamma, E) \\ E \vdash_m U' \to_{\textit{NodeTest}} U \end{array}}{E; \Gamma \vdash_m \overline{x} \texttt{ dos} :: \textit{NodeTest} : U}$$

```
BFSPrime($\overline{x}, \Gamma, E$)
 1  $Queue\ \ Q \leftarrow \emptyset$
 2  $boolean[]\ \ expVar$
 3  $Type\ \ result \leftarrow Prime_E(\Gamma(\overline{x}))$
 4  $Q \leftarrow Prime_E(\Gamma(\overline{x}))$
 5  while $Q\ \ not\ \ empty$
 6  do $Type\ \ Z = S.pop()$
 7      $E; \Gamma, \overline{x} : Z \vdash_m \overline{x}$ child :: $node() : Z'$
 8      if $Z' \neq ()$
 9         then if $Z'$ is not a type variable
10                 then AddPrime($Z', Q$)
11         else  if $not\ \ expVar[Z']$
12                 then $expVar[Z'] = $ true
13                       $Q.push(Prime_E(Z'))$
14                       $result = result \mid Prime_E(Z')$
15  return $result*$
```

```
AddPrime($T, Q$)
 1  if $not\ \ (genTypes\ \ contains\ \ T)$
 2     then $Q.push(Prime_E(T))$
 3           $result = result \mid Prime_E(T)$
```

Figure 13. DFSPrime procedure.

where BFSPrime is defined as shown in Figure 8.4.

The BFSPrime procedure uses several auxiliary data structures. $expVar$ is a boolean array that specifies whether a given type variable has been previously unfolded, and can be initialized in $O(|Def(E)|)$ time. Q is a queue holding type terms to be visited.

AddPrime is an auxiliary procedure that pushes its argument into Q and adds it to the result only if its argument has never seen before (through the $genType$ map).

The algorithm is essentially a variation of the standard breadth first tree search. The number of iterations of the `while` loop is bounded by $|E||\Gamma(\overline{x})|$, as no type variable is unfolded twice and no type term is visited more than once.

During each iteration, Rule (TypeChildNodeTest) is invoked; as proved in Lemma 8.19, this rule can be evaluated in time $O(|\Gamma(\overline{x})||E|)$. Procedure AddPrime can be computed in time $O((|E||\Gamma(\overline{x}))^2)$. As a consequence, Rule (TypeDos) can be evaluated in time $O((|E||\Gamma(\overline{x})|)^3)$. ■

We can now state the main result of this section.

Theorem 8.25 (Complexity of type inference) *If Q has no* `let` *clauses, then $\mathcal{C}(E; \Gamma \vdash_m Q : U) \in O(|Q|(|E|A)^3)$, where $A = max(|Q|, |E|)$.*

This theorem cannot be applied to queries with `let` clauses. Indeed, consider the following query:

```
let $y_1$ := $\overline{x}, \overline{x}$ return
    let $y_2$ := $y_1, y_1$ return
        ...
        let $y_n$ := $y_{n-1}, y_{n-1}$
            return $y_n$
```

If $\overline{x}$ has type $a[\mathsf{B}]$, then y_n will have type $(a[\mathsf{B}], a[\mathsf{B}])^n$, which is exponential in the size of the query.

While this is a very uncommon situation, it shows that the W3C type inference system may degrade to an exponential complexity.

8.5 Complexity of μXQ+ Type Inference

In the previous section we discovered that, according to the W3C type rules, type inference can be performed in polynomial time and space when the query being analyzed has no `let` clauses, and that type inference may be subject to combinatorial explosion when `let` clauses are nested.

The μXQ+ type system differs from the W3C one for the treatment of `for` clauses. In μXQ+, indeed, the inference algorithm iterates over the top level element types of the for-variable, and, for each element type, infers a distinct type for Q_2. As shown in Section 7, this iterative inference approach leads to a great precision improvement over the W3C type system.

In this section we will analyze the complexity of the μXQ+ type inference. To understand the complexity issues that may arise when the μXQ+ type inference approach is used, it is worth to analyze the behaviour of the type system through an example.

Example 8.26 Consider the following query:

```
for $\overline{x}_1$ in $l_1[b], l_2[b], l_3[b]$
return for $\overline{x}_2$ in $l_1[b], l_2[b], l_3[b]$
       return $\overline{x}_1, \overline{x}_2$
```

The type inference algorithm first analyzes the outer `for` clause and infers a type for $l_1[b], l_2[b], l_3[b]$. This type ($T_1 = l_1[\mathsf{B}], l_2[\mathsf{B}], l_3[\mathsf{B}]$) is not bound to $\overline{x}$ nor it is transformed in prime form. Instead, the algorithm invokes Rule (TypeInConc), which analyzes the inner query for three times: one for $l_1[\mathsf{B}]$, one for $l_2[\mathsf{B}]$, and one for $l_3[\mathsf{B}]$.

The inner query is, then, analyzed exactly in same way, hence the expression $\overline{x}_1, \overline{x}_2$ is evaluated nine times.

The inferred output type, hence, is the following:

$$\begin{array}{c} l_1[\mathsf{B}], l_1[\mathsf{B}], l_1[\mathsf{B}], l_2[\mathsf{B}], l_1[\mathsf{B}], l_3[\mathsf{B}], \\ l_2[\mathsf{B}], l_1[\mathsf{B}], l_2[\mathsf{B}], l_2[\mathsf{B}], l_2[\mathsf{B}], l_3[\mathsf{B}], \\ l_3[\mathsf{B}], l_1[\mathsf{B}], l_3[\mathsf{B}], l_2[\mathsf{B}], l_3[\mathsf{B}], l_3[\mathsf{B}] \end{array}$$

As it can be easily seen, this type has size in $O(|T_1|^k)$, where $k = 2$ is the number of nested `for` clauses. ■

This example shows that the size of the inferred type is a polynomial whose exponent is the number of `for` clauses in the query. In the worst case scenario $k \in O(|Q|)$, which implies that the output type may have exponential size in the size of the query. This implies that case analysis may require an exponential space to precisely type a query.

Summarizing, the type inference algorithm of μXQ+ is subject to exponential explosion phenomena when the query being analyzed contains nested `for` and `let` clauses.

9. Related Works

The problem of type inference for XQuery has been studied in several recent works.

In [7], where a variant of the μXQ+ type system has been first proposed, authors deals with the problem of detecting errors wrt the source type.

In [5], Cheney proposes a new type system able to deal with XML updates, while in [4] he describes a new type system mixing a subsumption mechanism and type rules for updates. In both these papers, Cheney adopts the μXQ+ case analysis technique for typing `queries`, but in none of them he compares this approach with that of the W3C system in term of precision and complexity.

In [14], Dan Suciu et al. focus on the problem of verifying whether the output of an XML query conforms to a given type (the so-called *XML type-checking problem*). Suciu et al. show that no precise type can be inferred by using regular type languages, and base their approach on *inverse* type inference (i.e., the problem of finding, given a query Q and output type T, an input type generating T), which can be performed in a sound and complete way by relying on *k-pebble automata*. This solution, however, has *non-elementary* complexity, which strongly affects its practical applicability. Similar approaches have been described in [12, 13].

10. Conclusions

In this paper we analyzed the problem of inferring an output type for an XQuery query. By comparing two type systems we identified the main sources of precision loss, namely the generation on non-regular languages, `for`-iterations, and `descendant-or-self` navigational steps.

We compared these two type systems not only in terms of precision, but also in terms of complexity, finding that type inference can be performed in polynomial time in the W3C system if the query being analyzed has no `let` clauses, while the μXQ+ system may require exponential space.

We also showed that the precision of the W3C can be improved when particular conditions are met, without increasing the computational complexity.

As a future work, we want to further investigate the relation between precision and complexity: in particular, we want to understand how tagged-path, one-type, conflict-freedom restrictions impact on the complexity of the inference algorithms.

Finally, we want to investigate the use numerical occurrence indicators to make the inferred types more succinct.

Acknowledgments

We would like to thank all the anonymous reviewers of this paper for their insightful comments.

References

[1] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based XML projection. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *VLDB*, pages 271–282. ACM, 2006.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, Jan. 2007. W3C Recommendation.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). Technical report, World Wide Web Consortium, 2006. W3C Recommendation.

[4] J. Cheney. Regular expression subtyping for XML query and update languages. In S. Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2008.

[5] J. Cheney. Flux: functional updates for XML. In J. Hook and P. Thiemann, editors, *ICFP*, pages 3–14. ACM, 2008.

[6] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for Path Correctness of XML Queries. In *Proceedings of the 2004 International Conference on Functional Programming (ICFP), Snowbird, Utah, September 19-22, 2004*, 2004.

[7] D. Colazzo., G. Ghelli, P. Manghi., and C. Sartiani. Static analysis for path correctness of XML queries. *Journal of Functional Programming*, 16(4-5):621–661, 2006.

[8] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient asymmetric inclusion between regular expression types. In *In Proceedings of the 12th International Conference on Database Theory (ICDT 2009), March 23-26 2009, Saint-Petersburg, Russia*, 2009.

[9] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.

[10] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW*, pages 556–567, 2003.

[11] H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[12] K. Inaba, H. Hosoya, and S. Maneth. Multi-return macro tree transducers. In O. H. Ibarra and B. Ravikumar, editors, *CIAA*, volume 5148 of *Lecture Notes in Computer Science*, pages 102–111. Springer, 2008.

[13] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In T. Schwentick and D. Suciu, editors, *ICDT*, volume 4353 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2007.

[14] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *PODS*, pages 11–22. ACM, 2000.

[15] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[16] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for XML data management. In *VLDB*, pages 974–985. Morgan Kaufmann, 2002.

[17] J. Siméon and P. Wadler. The essence of XML. In *POPL*, pages 1–13, 2003.

[18] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures Second Edition. Technical report, World Wide Web Consortium, Oct 2004. W3C Recommendation.