Precision and Complexity of XQuery Type Inference

Dario Colazzo*
LRI - Université Paris Sud/INRIA
Carlo Sartiani
Dipartimento di Matematica e Informatica - Università della Basilicata

1 Introduction

XQuery [6] is a functional and Turing-complete XML data manipulation 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 so to return new documents. Defined by the W3C Consortium, XQuery is also statically and strongly typed.

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, like `for`-iteration, for instance. 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.

An alternative and more precise approach for typing XQuery has been proposed in [4] and used as a basis for other proposals [2, 1, 3]. This type system has more precise type inference, at the price of an increased computational complexity.

Though these approaches are well-known 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, guided by existing classifications of schemas used in practice, could have a practical relevance as well, since it would provide important information to implementation designers.

In this paper we start this missing comparative analysis. Besides providing a clean and simple formalization of main typing mechanisms, we will formally study the complexity of the two approaches, show in which cases the W3C excessively over-approximates inferred types, and show that in many of these cases the type system proposed in [4] infers quite precise types still at a reasonable time cost.

2 Type Language

Our type language, which is an abstract model of the type language of XQuery, describes forests of *unranked*, *node-labeled* trees. Forests and trees obey the data model shown in following grammar, where () is the empty forest, b identifies atomic values, $l[f]$ represents an element labeled by l with the nodes in f as its children, and f, f denotes the ordered concatenation of forests.

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

As usual, the concatenation operator is associative and () is its neutral element: $(), f = f, () = f$.

Our type language is shown below, where () is the type for the empty sequence value, B denotes the type for base values, types T, U and $T \mid U$ are, respectively, ordered product and union types, and, finally, $T*$, $T+$, and $T?$ are the usual repetition and optional types. Vertical recursion is supported through *type environments* and type variables.

$$\begin{array}{lll} T & ::= & () \mid \mathsf{B} \mid l[T] \mid T,T \\ & & \mid T \mid T \mid T* \mid T+ \mid T? \\ & & \mid X \end{array}$$

$$\mathsf{B} ::= \texttt{String}$$

*This work has been partially funded by *Agence Nationale de la Recherche*, decision ANR-08-DEFIS-004.

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

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

Figure 1: The grammar of miniXQuery.

As usual, we restrict to $l[]$-guarded type environments, which are environments where only $l[]$-guarded vertical recursion is allowed.

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, hence expressive enough to capture the main type mechanisms of DTD and XML Schema.

As usual the semantics of type is defined as the minimal function that satisfies the following set of monotone equations:

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

3 miniXQuery

miniXQuery is a minimal language modeling the FLWR core of XQuery. miniXQuery contains `for`, `let`, `where`, and `return` clauses, and allows the user to specify both the child and the descendants-or-self axes. The predicate language comprises variable comparisons only. The syntax of miniXQuery is shown in Figure 1. The semantics of the language and the required auxiliary functions are shown in Table 3.1 and Figure 2.

There, ρ is a substitution assigning a forest to each free variable in the query; we make the assumption that each ρ 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 $\prod_{t \in trees(f)} A(t)$, yielding the forest $A(t_1), \ldots, A(t_n)$, when $f = t_1, \ldots, t_n$, and $()$ when $f = ()$.

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

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

Figure 2: Auxiliary functions and predicate evaluation.

4 The W3C Type System

4.1 Complexity

The first result we want to discuss here is stated in the following theorem.

Theorem 4.1 *Given an input type T and a miniXQuery query Q, $E; \Gamma \vdash_m Q : U$ can be computed, according to the W3C type system, in polynomial time and space.*

For reasons of space, we cannot prove here this result. However, we can illustrate it by looking at some of the most crucial type rules of the W3C type system.

Consider the rule (TypeFor) shown below.

(TypeFor)

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

The rule 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 $Prime_E(T_1)$. $Prime_E(T_1)$ returns the union of the uppermost base or tree

Table 3.1. *miniXQuery semantics*

$[\![b]\!]_\rho \triangleq b$ $\quad [\![x]\!]_\rho \triangleq \rho(x)$ $\quad [\![\overline{x}]\!]_\rho \triangleq \rho(\overline{x})$

$[\![()]\!]_\rho \triangleq ()$ $\quad [\![Q_1, Q_2]\!]_\rho \triangleq [\![Q_1]\!]_\rho, [\![Q_2]\!]_\rho$ $\quad [\![l[Q]]\!]_\rho \triangleq l[[\![Q]\!]_\rho]$

$[\![\overline{x}\ \texttt{child} :: \mathit{NodeTest}]\!]_\rho \triangleq \mathit{childr}([\![\overline{x}]\!]_\rho) :: \mathit{NodeTest}$

$[\![\overline{x}\ \texttt{dos} :: \mathit{NodeTest}]\!]_\rho \triangleq \mathit{dos}([\![\overline{x}]\!]_\rho) :: \mathit{NodeTest}$

$[\![\texttt{let}\ x ::= Q_1\ \texttt{return}\ Q_2]\!]_\rho \triangleq [\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho}$

$[\![\texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2]\!]_\rho \triangleq \prod_{t \in trees([\![Q_1]\!]_\rho)} [\![Q_2]\!]_{\rho, \overline{x} \mapsto t}$

$[\![\texttt{let}\ x ::= Q_1\ \texttt{where}\ P\ \texttt{return}\ Q_2]\!]_\rho \triangleq$ if $P(\rho, x \mapsto [\![Q_1]\!]_\rho)$ then $[\![Q_2]\!]_{\rho, x \mapsto [\![Q_1]\!]_\rho}$ else ()

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

types inside T 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 $\{?, +, *\}$ $(T_2\ .\ Quant_E(T_1))$.

It should be observed that Q_2 is visited only once by the inference rule, hence the complexity is not affected by the size of the inferred type for Q_1.

Consider now the rule (TypeDos) shown below.

(TypeDOS)

$$\frac{\begin{array}{l}\overline{x} : T \in \Gamma\ \wedge\ (T = m_1[T_1'] \mid m_2[T_2'] \mid \ldots \mid m_n[T_n'] \vee T = B) \\ E; \Gamma \vdash_m \overline{x}\ \mathit{child} : T_1 \\ E; \Gamma, \overline{x} : \mathit{Prime}_E(T_1) \vdash_m \overline{x}\ \mathit{child} : T_2 \\ E; \Gamma, \overline{x} : \mathit{Prime}_E(T_2) \vdash_m \overline{x}\ \mathit{child} : T_3 \\ \ldots \\ E; \Gamma, \overline{x} : \mathit{Prime}_E(T_n) \vdash_m \overline{x}\ \mathit{child} : T_{n+1} \\ E; \Gamma \vdash_m \mathit{Prime}_E(T_{n+1}) <: \mathit{Prime}_E(T_1) \mid \ldots \mid \mathit{Prime}_E(T_n) \\ U' = (\mathit{Prime}_E(T) \mid \mathit{Prime}_E(T_1) \mid \ldots \mid \mathit{Prime}_E(T_n))* \\ E \vdash_m U' \doteq_{\mathit{NodeTest}} U\end{array}}{E; \Gamma \vdash_m\ \overline{x}\ \texttt{dos} ::\ \mathit{NodeTest} : U}$$

This rule 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 $E; \Gamma \vdash_m \mathit{Prime}_E(T_{n+1}) <: \mathit{Prime}_E(T_1) \mid \ldots \mid \mathit{Prime}_E(T_n)$ is just a formal way to express the termination of the search in the parse tree and it does not involve any subtyping operation. The rule returns a type consisting of the star-guarded union of the types collected during the exploration.

It is quite easy to see that no type is visited twice and that the repeated application of $\mathit{Prime}_E(\cdot)$ can be computed in polynomial time by using some form of memoization.

4.2 Precision

To illustrate the precision issues that affects the W3C type system, we focus our attention on the following type rule.

(TypeChildNodeTest)

$$\frac{\begin{array}{l}\overline{x} : T \in \Gamma\ \wedge\ (T = m[T'] \vee T = B) \\ E \vdash_m \mathit{content}(T) \doteq_{\mathit{NodeTest}} U\end{array}}{E; \Gamma \vdash_m \overline{x}\ \texttt{child} :: \mathit{NodeTest} : U}$$

This rule infers an output type for a $\overline{x}$ `child` :: *NodeTest* filter by extracting the children types of T and filtering them according to *NodeTest*.

Consider now the following query:

```
for x̄ in l[b], m[b], n[b]
return for ȳ in x̄ child :: node()
       return x̄, ȳ
```

When invoked on this query, the type inference system first infers a type for `l[b], m[b], n[b]`. This type $(l[B], m[B], n[B])$ is then passed to $\mathit{Prime}_E(\cdot)$, whose result is $(l[B] \mid m[B] \mid n[B])$.

The inference system, then, infers a type for the inner query. In this case, the rule (TypeChildNodeTest) is applied and the result is just $B \mid B \mid B$.

Hence, the inferred output type for the query is: $((l[B] \mid m[B] \mid n[B]), (B))+$.

This type is a gross over-approximation of the actual type of the output, which is the following: $l[B], B, m[B], B, n[B], B$. This approximation is justified by the need to confine space and time complexity in the polynomial realm.

In the following section we will see how precision of type inference can be improved.

5 A More precise type system

The more precise type system proposed in [4] distinguishes for its rule to type `for`-iterations:

(TypeFor)

$$\frac{E; \Gamma \vdash Q_1 : T_1 \quad E; \Gamma \vdash \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : T_2}{E; \Gamma \vdash \texttt{for}\ \overline{x}\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2 : T_2}$$

As in the W3C case, the rule first computes the inferred type for Q_1; this type is then iterated on by a type analysis allowing the system to prove the premise $E;\Gamma \vdash \overline{x}\ \texttt{in}\ T_1\ \rightarrow\ Q_2 : T_2$. This analysis is performed according to several rules; for reasons of space, we only show below the most significant ones.

(TypeInSeq)

$$\frac{E;\Gamma \vdash \overline{x}\ \texttt{in}\ T_i\ \rightarrow\ Q : U_i \quad i = 1,2}{E;\Gamma \vdash \overline{x}\ \texttt{in}\ T_1, T_2\ \rightarrow\ Q : U_1, U_2}$$

(TypeInStar)

$$\frac{E;\Gamma \vdash \overline{x}\ \texttt{in}\ T\ \rightarrow\ Q : U}{E;\Gamma \vdash \overline{x}\ \texttt{in}\ T*\ \rightarrow\ Q : U*}$$

(TypeInEl)

$$\frac{E;\Gamma \vdash Q : U}{E;\Gamma \vdash \overline{x}\ \texttt{in}\ l[T]\ \rightarrow\ Q : l[U]}$$

The rules for auxiliary expressions $\overline{x}\ \texttt{in}\ T_1 \rightarrow Q_2$ are purely structural on T_1, and allow for a very precise type analysis, since possible over-approximations introduced by $Quant_E(T_1)$ are avoided. For example, when T_1 is a product T', T'', we have that, according to the W3C specification, $Quant_E(T_1)$ is $*$ (or $+$ in some cases). As a consequence, a $*$ type is always inferred for the whole `for` query. Instead, the above rule TypeInSeq infers a more precise sequence type T'_2, T''_2. In general a $*$-type is inferred much less frequently, typically only when T_1 contains a top level $*$ type. As a consequence, we have a more precise type inference.

On the other hand, for nested `for`-expressions this rule may lead to $O(n^k)$ time complexity, where k is the number of nested `for` expressions, and n is the maximal size of the type T_1^i inferred for the left hand side query of `for`-expressions. In practical cases, both k and n are rather small, hence the type inference time remains reasonable. This is testified by some tests that we made on real case and quite complex queries [5].

Concerning the typing of descendant-or-self axis, the type system proposed in [4] essentially adopts the same approach as the one proposed by the W3C. However, for non-recursive types, we can do better, as shown in [5]. Given a non-recursive type T (w.l.o.g. we can assume T not containing variables), we can define type inference for the descendant-or-self axis as follows:

Definition 5.1

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

It is straightforward to prove that for any non-recursive type T:

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

As it can be seen, the inference of $*$ types is reduced as much as possible, and this leads to a better precision wrt to the W3C approach.

6 Conclusions

This work is a first step towards a comparative analysis of the two current major proposals for XQuery type inference. In future work we aim at formally characterizing classes of queries and schemas for which the two systems differs in terms of precision. At the same time we will provide classes for which the two systems ensure the same degree of precision.

References

[1] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based XML projection. In *VLDB*, 2006.

[2] J. Cheney. Flux: functional updates for XML. In *ICFP*, pages 3–14, 2008.

[3] J. Cheney. Regular expression subtyping for XML query and update languages. In *ESOP*, 2008.

[4] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for Path Correctness of XML Queries. In *ICFP*, 2004.

[5] D. Colazzo and C. Sartiani. Detection of corrupted schema mappings in xml data integration systems. *ACM Trans. Internet Techn.*, 9(4), 2009.

[6] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, Jan. 2007. W3C Recommendation.