XQuery and Static Typing: Tackling the Problem of Backward Axes

Pierre Genevès, Nils Gesbert, Nabil Layaïda

XQuery and Static Typing: Tackling the Problem of Backward Axes

Pierre Genevès[1], Nils Gesbert[2], and Nabil Layaïda[3]

[1] CNRS
[2] Université Grenoble Alpes
[3] INRIA

Abstract. XQuery is a functional language dedicated to XML data querying and manipulation. As opposed to other W3C-standardized languages for XML (e.g. XSLT), it has been intended to feature strong static typing. Currently, however, some expressions of the language cannot be statically typed with any precision. We argue that this is due to a discrepancy between the semantics of the language and its type algebra: namely, the values of the language are (possibly inner) tree nodes, which may have siblings and ancestors in the data. The types on the other hand are regular tree types, as usual in the XML world: they describe sets of trees. The type associated to a node then corresponds to the subtree whose root is that node and contains no information about the rest of the data. This makes navigational expressions using 'backward axes,' which return e.g. the siblings of a node, impossible to type.
We discuss how to solve this discrepancy and propose a compromise : to use extended types representing possibly inner tree nodes in some key parts of a program, and to cut out the subtrees from their original context in the rest.

1 Introduction

XQuery is a functional language with some unusual features. The standard which defines it [1,2] describes, among other things, a formal semantics for a core fragment of the language, rules to compile the full language into its core fragment, and a static type system.

Although Turing-complete, this language is not general-purpose; it is designed for manipulating XML data, in various ways. Its type system is thus built around regular tree types, as usual for XML data. The values of the language, however, are not trees or forests, but *sequences of pointers* to tree nodes. These pointers can point anywhere in the tree, not only at the root, and it is always possible, given such a pointer, to get pointers to its parent and sibling nodes. Furthermore, a sequence may contain pointers into different trees.

The formal semantics from the XQuery standard uses judgements of the form DynEnv $\vdash$ *Expr* $\Rightarrow$ *Val*, where DynEnv is a store of trees. Navigational expressions (e.g. getting the parent of a node) are evaluated by looking up the initial pointer in the store, navigating in there, and returning a pointer to the

destination. However, XQuery is designed as a pure functional language and all the trees in the store are immutable[4]; the only expressions which update the store are those which create a new tree, returning a pointer to its root.

Because of this purity, it is possible to describe the semantics of Core XQuery without using an external store, but only reduction rules for expressions, if we represent tree nodes as *focused trees*, a data structure describing a whole tree 'seen' from a given internal node. We believe that it makes it easier to reason about programs. This will be our first contribution (Sec. 2).

This formalization will allow us to highlight a discrepancy between the semantics of XQuery and its type system (Sec. 4.2): whereas the values manipulated by the language consist of a subtree *and a context*, the types describe only the subtree and say nothing of the context. Because of this, expressions navigating upwards or between siblings are simply given the most general type, which contains no information whatsoever, regardless of the type of the initial node.

In order to solve this discrepancy, we then define (Sec. 5) a logic whose formulas denote sets of *focused trees* rather than just of trees, and discuss how it could be combined with the existing type system.

2 Syntax and Semantics of an XQuery Navigational Core

2.1 Values

Items and Sequences XQuery programs manipulate two 'levels' of values: items and sequences. In full XQuery, item values can be literals of various base types (string, boolean etc.), functions (in XQuery 3.0 [4]), and tree nodes. Base values and function values behave in a fairly standard way in XQuery, so, in order to keep this paper to the point, we consider the fragment where all items are tree nodes. Furthermore, we focus on the structure of XML trees and thus consider them composed of only element nodes (with no text content or attributes). This does not imply a loss of generality since literals and text could be encoded as trees.

In XQuery, sequence values are flat lists of items. Nested sequences do not exist. The result of evaluating an expression is always a sequence.

Tree nodes are pointers to nodes which can be anywhere in a tree, not necessarily at the root. Since the tree data structures manipulated by XQuery are always immutable, we need not however actually represent these node values as pointers into a shared data structure defined in an external environment: we may represent them as *focused trees* which contain all the information we need. We detail this structure in the next subsection.

Focused Trees In order to represent references to nodes of immutable trees, we use *focused trees*, inspired by Huet's Zipper data structure [5] in the manner

[4] Expressions that can be used to make persistent changes to instances in the XQuery data model are defined as a separate extension of the language [3].

of [6]. Focused trees not only describe a tree but also its context: its siblings and its parent, including its parent context recursively.

Formally, we assume an alphabet Σ of labels, ranged over by σ. The syntax of our data model is as follows.

$$
\begin{array}{rll}
t ::= & \sigma[tl] & \text{tree} \\
tl ::= & & \text{list of trees} \\
& \epsilon & \quad \text{empty list} \\
\mid & t :: tl & \quad \text{cons cell} \\
c ::= & & \text{context} \\
& \mathit{Top} & \quad \text{root of the tree} \\
\mid & (tl, c[\sigma], tl) & \quad \text{context node} \\
f ::= & (t, c) & \text{focused tree}
\end{array}
$$

A focused tree (t, c) is a pair consisting of a tree t and its context c. The context $(tl, c[\sigma], tl)$ comprises three components: a list of trees at the left of the current tree in reverse order (the first element of the list is the tree immediately to the left of the current tree), the context above the tree, and a list of trees at the right of the current tree. The context above the tree may be *Top* if the current tree is at the root, otherwise it is of the form $c[\sigma]$ where σ is the label of the enclosing element and c is the context in which the enclosing element occurs.

We now describe how to navigate focused trees, in binary style. There are four directions that can be followed: for a focused tree f, $f\langle 1\rangle$ changes the focus to the first child of the current tree, $f\langle 2\rangle$ changes the focus to the next sibling of the current tree, $f\langle \overline{1}\rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f\langle \overline{2}\rangle$ changes the focus to the previous sibling.

Formally, we have:

Definition 1.

$$
\begin{aligned}
(\sigma[t :: tl], c)\langle 1\rangle &\stackrel{def}{=} (t, (\epsilon, c[\sigma], tl)) \\
(t, (tl_l, c[\sigma], t' :: tl_r))\langle 2\rangle &\stackrel{def}{=} (t', (t :: tl_l, c[\sigma], tl_r)) \\
(t, (\epsilon, c[\sigma], tl))\langle \overline{1}\rangle &\stackrel{def}{=} (\sigma[t :: tl], c) \\
(t', (t :: tl_l, c[\sigma], tl_r))\langle \overline{2}\rangle &\stackrel{def}{=} (t, (tl_l, c[\sigma], t' :: tl_r))
\end{aligned}
$$

When the focused tree does not have the required shape, these operations are not defined.

2.2 Expressions

The standard defining XQuery describes how to compile ('normalize') expressions of the full language into a core fragment, called the XQuery Core [2]. Although this part of the specification has not been updated after XQuery 1.0, it still is a good starting point.

The formal semantics for this core fragment is defined using an external store, with node items being pointers into that store. What we propose to do is to

$e ::=$		expression
	$\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>}$	XML element
$\mid$	ϵ	empty sequence
$\mid$	e, e	sequence concatenation
$\mid$	$\texttt{for}\ \$v\ \texttt{in}\ e\ \texttt{return}\ e$	for loop
$\mid$	$\texttt{if empty}(e)\ \texttt{then}\ e\ \texttt{else}\ e$	existence test
$\mid$	$\$v/axis{::}n$	tree navigation
$\mid$	$\$v$	item variable
$n ::=$		name test
	σ	label
$\mid$	$*$	wildcard
$s ::=$		value sequence
	ϵ	empty sequence
$\mid$	$f :: s$	cons cell
$\mathcal{E} ::=$		evaluation context
	$[]$	context hole
$\mid$	$\texttt{<}\sigma\texttt{>}\{\mathcal{E}\}\texttt{</}\sigma\texttt{>}$	
$\mid$	$\mathcal{E}, e$	
$\mid$	$s, \mathcal{E}$	
$\mid$	$\texttt{for}\ \$v\ \texttt{in}\ \mathcal{E}\ \texttt{return}\ e$	
$\mid$	$\texttt{if empty}(\mathcal{E})\ \texttt{then}\ e\ \texttt{else}\ e$	

Fig. 1. Navigational core of XQuery.

replace these pointers with focused trees, as described in the previous subsection, which removes the need for a store. As the XQuery Core is already quite large, we will consider a much smaller fragment comprising only constructs impacted by this proposal and useful for the discussion, which we call the navigational core. It is worth noting that several other 'core fragments' of XQuery have already been defined and studied in research papers. We will discuss how this one relates to them in the related work section (Sec. 6).

The navigational XQuery fragment we consider is described by the abstract syntax shown in Figure 1, where $axis \in \{\texttt{child}, \texttt{desc}, \texttt{parent}, \texttt{anc}, \texttt{psibl}, \texttt{nsibl}, \texttt{self}\}$. The values of the language are sequences s; we write $[f_1, \ldots, f_n]$ for $f_1 :: \ldots :: f_n :: \epsilon$.

2.3 Reduction Semantics

Figure 4 gives reduction rules defining a small-step operational semantics for the focused-tree-based navigational XQuery fragment we consider.

Note that, because $f\,\langle\overline{1}\rangle$ and $f\,\langle\overline{2}\rangle$ are never both defined for the same f, rules R-Parent and R-PSParent are mutually exclusive, and that R-NoParent can only apply in a case where both $f\,\langle\overline{1}\rangle$ **and** $f\,\langle\overline{2}\rangle$ are undefined. The same is true of R-Anc, R-PSAnc and R-NoAnc, so that the set of rules is almost deterministic. The only ambiguity is the order of concatenation in expressions of the form s_1, s_2, s_3, but in that case note that the result is independent on that order.

$$\text{(R-Tree)}\quad \texttt{<}\sigma\texttt{>}\{[(t_1,c_1),(t_2,c_2),\ldots,(t_n,c_n)]\}\texttt{</}\sigma\texttt{>} \longrightarrow [(\sigma[t_1 :: t_2 :: \ldots :: t_n :: \epsilon], Top)]$$

$$\text{(R-For)}\quad \texttt{for } \$v \texttt{ in } f_1 :: s \texttt{ return } e \longrightarrow e\left[{}^{f_1}/_{\$v}\right], \texttt{for } \$v \texttt{ in } s \texttt{ return } e$$

$$\text{(R-ForEmpty)}\quad \texttt{for } \$v \texttt{ in } \epsilon \texttt{ return } e \longrightarrow \epsilon \qquad \text{(R-Singleton)}\quad f \longrightarrow [f]$$

$$\text{(R-Concat)}\quad [f_1,\ldots,f_n],[f'_1,\ldots,f'_{n'}] \longrightarrow [f_1,\ldots,f_n,f'_1,\ldots f'_{n'}]$$

$$\text{(R-IfT)}\quad \texttt{if empty}(\epsilon) \texttt{ then } e_1 \texttt{ else } e_2 \longrightarrow e_1 \qquad \text{(R-IfF)}\quad \texttt{if empty}(f :: s) \texttt{ then } e_1 \texttt{ else } e_2 \longrightarrow e_2$$

$$\text{(R-NoParent)}\ (t, Top)/\texttt{parent::}n \longrightarrow \epsilon \qquad \text{(R-NoChild)}\ (\sigma[\epsilon], c)/\texttt{child::}n \longrightarrow \epsilon$$

$$\text{(R-NoNSibl)}\ (t,(tl,\sigma[c],\epsilon))/\texttt{nsibl::}n \longrightarrow \epsilon \qquad \text{(R-NoPSibl)}\ (t,(\epsilon,\sigma[c],tl))/\texttt{psibl::}n \longrightarrow \epsilon$$

$$\text{(R-NoAnc)}\ (t, Top)/\texttt{anc::}n \longrightarrow \epsilon \qquad \text{(R-NoDesc)}\ (\sigma[\epsilon], c)/\texttt{desc::}n \longrightarrow \epsilon$$

$$\text{(R-SelfStar)}\quad f/\texttt{self::*} \longrightarrow [f] \qquad \text{(R-SelfMatch)}\quad (\sigma[tl],c)/\texttt{self::}\sigma \longrightarrow [(\sigma[tl],c)]$$

$$\text{(R-SelfDiff)}\ \frac{\sigma \neq \sigma'}{(\sigma[tl],c)/\texttt{self::}\sigma' \longrightarrow \epsilon} \qquad \text{(R-Parent)}\ \frac{f' = f\,\langle\overline{1}\rangle}{f/\texttt{parent::}n \longrightarrow f'/\texttt{self::}n}$$

$$\text{(R-PSParent)}\ \frac{f' = f\,\langle\overline{2}\rangle}{f/\texttt{parent::}n \longrightarrow f'/\texttt{parent::}n}$$

$$\text{(R-Child)}\ \frac{f' = f\,\langle 1\rangle}{f/\texttt{child::}n \longrightarrow f'/\texttt{self::}n, f'/\texttt{nsibl::}n}$$

$$\text{(R-NSibl)}\ \frac{f' = f\,\langle 2\rangle}{f/\texttt{nsibl::}n \longrightarrow f'/\texttt{self::}n, f'/\texttt{nsibl::}n}$$

$$\text{(R-PSibl)}\ \frac{f' = f\,\langle\overline{2}\rangle}{f/\texttt{psibl::}n \longrightarrow f'/\texttt{psibl::}n, f'/\texttt{self::}n}$$

$$\text{(R-Anc)}\ \frac{f' = f\,\langle\overline{1}\rangle}{f/\texttt{anc::}n \longrightarrow f'/\texttt{anc::}n, f'/\texttt{self::}n} \qquad \text{(R-PSAnc)}\ \frac{f' = f\,\langle\overline{2}\rangle}{f/\texttt{anc::}n \longrightarrow f'/\texttt{anc::}n}$$

$$\text{(R-Desc)}\ \frac{f' = f\,\langle 1\rangle}{f/\texttt{desc::}n \longrightarrow f'/\texttt{self::}n, f'/\texttt{desc::}n, f'/\texttt{nsibl::}n} \qquad \text{(R-Context)}\ \frac{e_1 \longrightarrow e_2}{\mathcal{E}[e_1] \longrightarrow \mathcal{E}[e_2]}$$

Fig. 2. Reduction Rules for the Navigational XQuery Fragment

3 Problem Statement

Now that we have a simple formal semantics for a core fragment of XQuery, we want to study the problem of type-checking on this fragment.

However, the reader may have noticed that the reduction rules we have defined cannot actually generate an error value, nor can they get stuck on a syntactically correct expression. Thus, in this small fragment, programs cannot go wrong, independently of typing. We will therefore state our problem not in terms of type *safety* but of type *conformance*: the program always generates a result, but we want to check statically whether this result conforms to what has been specified.

We did not include functions in our fragment, but will take as our use case a function definition with type annotations. We thus add to the syntax *sequence variables* $\$\overline{v}$ to represent the function's parameters, and the problem we want to solve is the following:

Given a typing environment $\Gamma = \$\overline{v}_1 : \tau_1, \ldots, \$\overline{v}_n : \tau_n$*, an expression* e *referring to variables* $\$\overline{v}_1 \ldots \$\overline{v}_n$*, and a result type* τ*, is it the case that, for any sequence values* $s_1 \ldots s_n$ *matching types* $\tau_1 \ldots \tau_n$*, the expression* $\mathrm{e}\left[{}^{s_1 \ldots s_n}/_{\$\overline{v}_1 \ldots \$\overline{v}_n}\right]$ *reduces to a value matching type* τ *?*

As usual, we will require checking procedures for answering this question to be *sound* (give no false positives) but not necessarily *complete* (never give any false negatives). We want them however to be as *precise* as possible (as few false negatives as possible).

We now have to say what types are. The usual formal definition, close to what the standard uses, is given in the next section.

4 The XQuery Static Type System

4.1 Regular Tree Types

As is customary in the literature ([7,8,9,10] for instance), we use a slight variant of XDuce's type language [11,12], described in Fig. 3, to represent (core) XQuery types.

Unit types u, or 'prime types' in the XQuery terminology, correspond to items. Types τ correspond to sequences. In the general case, u would include both element types and base types; since we removed base values from the language fragment we consider, it only includes element types.

A *type environment* E is a mapping from type references x to types τ. An environment must respect some well-formedness constraints, so that badly-founded recursion such as $x = () \mid x$ for example is not allowed. These constraints are detailed formally in [12]. In the following, we do not treat the case of x explicitly: we always assume we have an environment E which defines it and implicitly replace the variable with its binding when needed. We also consider that E always contains the type of all elements, AnyElt, defined as AnyElt = `element` $*$ {AnyElt$*$}.

The semantics of types is defined in terms of sets of forests, i.e. of sequences of trees (called *elements* in the XML context). A value s, which is a sequence

$u ::=$		unit type
	`element` $n\ \{\tau\}$	element
$\mid$	$u \mid u$	choice
$n ::=$		name test
	σ	label
$\mid$	$*$	wildcard
$\tau ::=$		sequence type
	u	unit type
$\mid$	`()`	empty sequence
$\mid$	τ, τ	concatenation
$\mid$	$\tau \mid \tau$	choice
$\mid$	$\tau*$	repetition
$\mid$	x	type reference

Fig. 3. XQuery Types

of *items* (nodes, focused trees in our semantics), *matches* a type if the forest constituted of the subtrees rooted at all nodes of the sequence belongs to the semantics of the type. This is the same forest that is constructed in the tree creation rule R-Tree as the children of the new node. Formally:

$$
\begin{aligned}
[\![\texttt{element}\ \sigma\ \{\tau\}]\!] &= \{[\sigma[tl]] \mid tl \in [\![\tau]\!]\} \\
[\![\texttt{element}\ *\ \{\tau\}]\!] &= \{[\sigma[tl]] \mid \sigma \in \Sigma \text{ and } tl \in [\![\tau]\!]\} \\
[\![\texttt{()}]\!] &= \epsilon \\
[\![\tau, \tau']\!] &= \{[t_1, \ldots, t_n, t'_1, \ldots, t'_m] \mid [t_1 \ldots t_n] \in [\![\tau]\!] \text{ and } [t'_1 \ldots t'_m] \in [\![\tau']\!]\} \\
[\![\tau \mid \tau']\!] &= [\![\tau]\!] \cup [\![\tau']\!] \\
[\![\tau^0]\!] &= \epsilon \\
[\![\tau^{n+1}]\!] &= [\![\tau, \tau^n]\!] \\
[\![\tau*]\!] &= \bigcup_{n \in \mathbb{N}} [\![\tau^n]\!]
\end{aligned}
$$

Note that in some cases the definition of $[\![\tau]\!]$ above loops; in those cases, the semantics is empty ($[\![\tau]\!] = \varnothing$).

4.2 Typing Rules

The rules use type environments E, which contain the possibly mutually recursive definitions of named types (see above), and typing environments Γ which map sequence variables to sequence types, as said in Sec. 3, and in addition map iteration variables to **unit types** (not types in general). Indeed, in the `for` loop, the variable is bound successively to all items in the input sequence, thus its value is an item, not a sequence.

This system constructs deterministically one type for the whole expression from the types of its subexpressions (which is sometimes called forward type

inference). In general, the computed type will not be exactly the one required in our type-checking question, but we can still answer positively if the semantics of the computed type is included in the semantics of the required type, so there is an additional inclusion check (which is straightforward for this regular type language). Here precision of typing thus means computing the smallest possible output type.

In the XQuery standard, typing the `for` loop involves two operations on types, prime and quant. The first one constructs the disjunction of all unit types which appear in a sequence type. The second one determines whether a sequence type can contain sequences with zero, one, or several items. Its result is a multiplier from the set $\{0, 1, +, ?, *\}$.

Typing the node test involves filtering a type to extract only the matching element types and is done by the filter operation. We also define children and dos operations to extract from a unit type all the possible types for, respectively, its children and its descendants plus itself.

The + operation on multipliers q, the prime and quant operators, the application of a multiplier to a sequence type, and the filter, children and dos operations are formally defined as follows:

$$\begin{aligned} 0 + q &= q \\ 1 + q &= + \text{ if } q \neq 0 \\ + + q &= + \\ ? + * &= * \end{aligned}$$

$$\begin{aligned} \mathsf{prime}(u) &= u \\ \mathsf{prime}(()) &= () \\ \mathsf{prime}(\tau_1, \tau_2) &= \mathsf{prime}(\tau_1) \mid \mathsf{prime}(\tau_2) \\ \mathsf{prime}(\tau *) &= \mathsf{prime}(\tau) \end{aligned}$$

$$\begin{aligned} \mathsf{quant}(u) &= 1 \\ \mathsf{quant}(()) &= 0 \\ \mathsf{quant}(\tau_1, \tau_2) &= \mathsf{quant}(\tau_1) + \mathsf{quant}(\tau_2) \\ \mathsf{quant}(\tau *) &= 0 \text{ if } \mathsf{quant}(\tau) = 0, * \text{ otherwise} \end{aligned}$$

$$\begin{aligned} \tau^0 &= () \\ \tau^1 &= \tau \\ \tau^+ &= \tau, \tau * \\ \tau^? &= () \mid \tau \\ \tau^* &= \tau * \end{aligned}$$

$$
\begin{aligned}
\mathsf{filter}(()) &= () \\
\mathsf{filter}(\texttt{element} * \{\tau\}, n) &= \texttt{element} * \{\tau\} \\
\mathsf{filter}(\texttt{element}\ n\ \{\tau\}, *) &= \texttt{element}\ n\ \{\tau\} \\
\mathsf{filter}(\texttt{element}\ \sigma\ \{\tau\}, \sigma) &= \texttt{element}\ \sigma\ \{\tau\} \\
\mathsf{filter}(\texttt{element}\ \sigma\ \{\tau\}, \sigma') &= () \text{ if } \sigma \neq \sigma' \\
\mathsf{filter}(\tau_1 \mid \tau_2, n) &= \mathsf{filter}(\tau_1) \mid \mathsf{filter}(\tau_2) \\
\mathsf{filter}(\tau_1, \tau_2, n) &= \mathsf{filter}(\tau_1), \mathsf{filter}(\tau_2) \\
\mathsf{filter}(\tau*) &= \mathsf{filter}(\tau)*
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{children}(\texttt{element}\ n\ \{\tau\}) &= \tau \\
\mathsf{children}(\tau_1 \mid \tau_2) &= \mathsf{children}(\tau_1) \mid \mathsf{children}(\tau_2) \\
\mathsf{children}(\tau_1, \tau_2) &= \mathsf{children}(\tau_1), \mathsf{children}(\tau_2) \\
\mathsf{children}(\tau*) &= \mathsf{children}(\tau)*
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{dos}(()) &= () \\
\mathsf{dos}(\texttt{element}\ n\ \{\tau\}) &= \texttt{element}\ n\ \{\tau\}, \mathsf{dos}(\tau) \\
\mathsf{dos}(\tau_1 \mid \tau_2) &= \mathsf{dos}(\tau_1) \mid \mathsf{dos}(\tau_2) \\
\mathsf{dos}(\tau_1, \tau_2) &= \mathsf{dos}(\tau_1), \mathsf{dos}(\tau_2) \\
\mathsf{dos}(\tau*) &= \mathsf{dos}(\tau)*
\end{aligned}
$$

$$
\text{(T-SeqVar)}\ \ E; \Gamma, \$\overline{v} : \tau \vdash \$\overline{v} : \tau
\qquad
\text{(T-Empty)}\ \ E; \Gamma \vdash \epsilon : ()
$$

$$
\text{(T-Tree)}\ \frac{E; \Gamma \vdash e : \tau}{E; \Gamma \vdash \texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : \texttt{element}\ \sigma\ \{\tau\}}
\qquad
\text{(T-Seq)}\ \frac{E; \Gamma \vdash e_1 : \tau_1 \qquad E; \Gamma \vdash e_2 : \tau_2}{E; \Gamma \vdash e_1, e_2 : \tau_1, \tau_2}
$$

$$
\text{(T-If)}\ \frac{E; \Gamma \vdash e_1 : \tau_1 \qquad E; \Gamma \vdash e_2 : \tau_2}{E; \Gamma \vdash \texttt{if empty}(e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1 \mid \tau_2}
$$

$$
\text{(T-For)}\ \frac{E; \Gamma \vdash e_1 : \tau_1 \qquad E; \Gamma, \$v : \mathsf{prime}(\tau_1) \vdash e_2 : \tau_2}{E; \Gamma \vdash \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2 : \tau_2^{\mathsf{quant}(\tau_1)}}
$$

$$
\text{(T-Self)}\ \ E; \Gamma, \$v : u \vdash \$v/\texttt{self::}n : \mathsf{filter}(u, n)
$$

$$
\text{(T-Child)}\ \ E; \Gamma, \$v : u \vdash \$v/\texttt{child::}n : \mathsf{filter}(\mathsf{children}(u), n)
$$

$$
\text{(T-Desc)}\ \ E; \Gamma, \$v : u \vdash \$v/\texttt{desc::}n : \mathsf{filter}(\mathsf{dos}(\mathsf{children}(u)), n)
$$

$$
\text{(T-Parent)}\ \ E; \Gamma, \$v : u \vdash \$v/\texttt{parent::}n : () \mid \mathsf{AnyElement}
$$

$$
\text{(T-OthAxis)}\ \frac{axis \in \{\texttt{anc, psibl, nsibl}\}}{E; \Gamma, \$v : u \vdash \$v/axis\texttt{::}n : \mathsf{AnyElement}*}
$$

Fig. 4. Standard Typing Rules for the Navigational XQuery Fragment

We can see that this type system lacks precision in several places. One of them is Rule T-For, which gives to the sequence resulting from the loop a *homogeneous type*, disregarding completely the fact that, since the input sequence

may contain specific types in a specific order, it should be possible in most cases to deduce ordering information about the different item types which can occur in the output. This source of imprecision is well known and has been adressed in several research papers already, e.g. [8], [10]. Note that it can be adressed while staying within the standard type language presented thus far: the problem is that the type system does not fully exploit the information it has. We will not expand on this point further in this paper.

The other blatant source of imprecision is Rule T-OthAxis, and to a very slightly lesser extent Rule T-Parent. Indeed, type AnyElement* is the top type of the algebra; any value whatsoever matches that type[5]. Thus the type given to navigation expressions with axes `psibl, nsibl` or `ancestor` contains no information at all, and it is not much better for `parent`. This means that typechecking such an expression will always fail (unless no requirement was made on the result) and that the programmer using it will be forced to by-pass static checking with a type cast.

This source of imprecision is also well known, it has however not been adressed yet as far as we know, except, indirectly (*via* a translation into another language), in [13]. The reason why it has not, we think, is because it is more fundamental than the problem of `for` loops. If we look at the information available to the type system, i.e. the expression and the typing environment, there is one small improvement we can easily make: if the expression is of the form $\$v/axis::\sigma$, then we know that the result must contain only nodes labelled σ—this is still very limited information. Can we do better? If we look at the semantics of Fig. 4, we know that at some point, when the surrounding `for` loop is unfolded, $\$v$ will be replaced with a focused tree $f = (t, c)$. Then, depending on the axis and the shape of f, either $f\langle\overline{2}\rangle$, $f\langle\overline{1}\rangle$ or $f\langle 2\rangle$ will be computed to yield the result. The definitions of these operations (Def. 1) show that the first component (the tree) of the result is taken from c, in the case of $\langle 2\rangle$ and $\langle\overline{2}\rangle$, or is constructed from both t and a part of c in the case of $\langle\overline{1}\rangle$.

Now what is the type information we have? the environment Γ maps iteration variables to unit types, which describe nonempty sets of trees. The fact that $\$v$ has type u means that when, during evaluation, $\$v$ is replaced with (t, c), t will be an element of $[\![u]\!]$—it says nothing on what c will possibly be. Therefore it is impossible to know anything about the result of an expression such as $\$v/$`nsibl::*` using only this kind of type environment. The problem is highlighted by our focused-tree-based formalization: item values can be represented as pairs of which only the first component has a type—there is a clear discrepancy between the semantics and the type system.

It is worth noting that in the case of $\langle\overline{1}\rangle$ we do not have exactly zero information on the result, due to the fact that t, for which we have a type, appears as the first child of the new tree. $\langle\overline{1}\rangle$ is used in the reduction rules for the `parent` and `anc` axes. This fact would allow us to improve the T-Parent rule slightly: we know that, if the parent exists, it has at least one child, which has type u. We can also use this information when typing the `anc` axis.

[5] Admittedly, if we had base types or function types, it would exclude them.

We summarise below the very limited improvements which are possible for the typing of navigation expressions within the standard type environment:

$$\text{(T-Parent')}\quad E; \Gamma, \$v : u \vdash \$v/\texttt{parent::}n : \texttt{()} \mid \texttt{element}\ n\ \{\mathsf{AnyElement}*, u, \mathsf{AnyElement}*\}$$

$$\text{(T-Anc)}\ \frac{E' = E, x = u \mid \texttt{element}\ *\ \{\mathsf{AnyElement}*, x, \mathsf{AnyElement}*\}}{E'; \Gamma, \$v : u \vdash \$v/\texttt{anc::}n : \texttt{()} \mid \texttt{element}\ n\ \{\mathsf{AnyElement}*, x, \mathsf{AnyElement}*\}}$$

$$\text{(T-OthAxis')}\ \frac{axis \in \{\texttt{psibl, nsibl}\}}{E; \Gamma, \$v : u \vdash \$v/axis\texttt{::}n : \texttt{element}\ n\ \{\mathsf{AnyElement}*\}*}$$

If we want to do better, we have to change either the type language or the semantics. We first try to do the former, in the next section.

5 Types for Focused Trees

5.1 A Tree Logic

In order to describe sets of focused trees rather than just sets of trees, we use the logic language defined in [6]. It is a sub-logic of the alternation free modal μ-calculus with converse; its syntax is given in Fig. 5, where $a \in \{1, 2, \overline{1}, \overline{2}\}$ are *programs*, corresponding to the four directions in which trees can be navigated.

Our main reasons for choosing this formalism are: it is expressive enough to support all XQuery types, it is succinct (types are represented as formulas of linear size compared to their regular expression syntax), and the satisfiability problem for a logical formula of size n can be efficiently decided with an optimal $2^{\mathcal{O}(n)}$ worst-case time complexity bound with the solver of [6].

Formulas include the truth predicate, atomic propositions (denoting the label of the node in focus), disjunction and conjunction of formulas, formulas under an existential modality (denoting the existence of a node, in the direction denoted by the program, satisfying the sub-formula), and a fixpoint operator.

$\varphi, \psi ::=$		formula
	$\top$	true
$\mid$	$\sigma \quad \mid \quad \neg\sigma$	atomic proposition (negated)
$\mid$	X	variable
$\mid$	$\varphi \vee \psi$	disjunction
$\mid$	$\varphi \wedge \psi$	conjunction
$\mid$	$\langle a \rangle \varphi \quad \mid \quad \neg \langle a \rangle \top$	existential (negated)
$\mid$	$\mu(X_i = \varphi_i)_{i \in I}$ in ψ	(least) n-ary fixpoint

Fig. 5. Logic formulas

The logic allows expressing recursion through the fixpoint binder. The recursive formula $\langle 1 \rangle (\mu X.a \vee \langle 1 \rangle X \vee \langle 2 \rangle X)$ states the existence of some node labelled with "a" at an arbitrary depth in the subtree. The meaning of the recursive formula $\mu X.b \vee \langle \overline{2} \rangle X$ is that either the current node is labeled b or some previous sibling of the current node is labeled b.

The interpretation of a logical formula is the set of focused trees such that the formula is satisfied at the current node. The semantics of this logic is intuitively explained through examples in [14] and formally defined in [6]. We give the formal definition in Fig. 6, where $\mathcal{F}$ is the set of all focused trees and $\mathtt{nm}(f)$ is the label at the current node of f.

$$\begin{aligned}
[\![\top]\!]_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![\sigma]\!]_V &\stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) = \sigma\} \\
[\![X]\!]_V &\stackrel{\text{def}}{=} V(X) & [\![\neg\sigma]\!]_V &\stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) \neq \sigma\} \\
[\![\varphi \vee \psi]\!]_V &\stackrel{\text{def}}{=} [\![\varphi]\!]_V \cup [\![\psi]\!]_V & [\![\varphi \wedge \psi]\!]_V &\stackrel{\text{def}}{=} [\![\varphi]\!]_V \cap [\![\psi]\!]_V
\end{aligned}$$

$$\begin{aligned}
[\![\langle a\rangle\, \varphi]\!]_V &\stackrel{\text{def}}{=} \{f\,\langle\overline{a}\rangle \mid f \in [\![\varphi]\!]_V \wedge f\,\langle\overline{a}\rangle \text{ defined}\} \\
[\![\neg\,\langle a\rangle\, \top]\!]_V &\stackrel{\text{def}}{=} \{f \mid f\,\langle a\rangle \text{ undefined}\}
\end{aligned}$$

$$[\![\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi]\!]_V \stackrel{\text{def}}{=} \text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, [\![\varphi_j]\!]_{V[\overline{T_i/X_i}]} \subset T_j\} \quad \text{in}$$
$$\text{let } (U_j) = \left(\textstyle\bigcap_{(T_i)\in S} T_j\right)_{j\in I} \quad \text{in} \quad [\![\psi]\!]_{V[\overline{U_i/X_i}]}$$

where $V[\overline{T_i/X_i}](X) = V(X)$ if $X \notin \{X_i\}$ and T_i if $X = X_i$.

Fig. 6. Interpretation of formulas

The lemma 4.2 of [6] says that the interpretation of a fixpoint formula is equal to the union of the interpretations of all its finite unfoldings (where unfolding is defined as usual). A consequence (detailed in [6]) is that the logic is closed under negation, i. e. for any closed φ, $\neg\varphi$ can be expressed in the syntax using De Morgan's relations and this definition:

$$\begin{aligned}
\neg\,\langle a\rangle\, \varphi &\stackrel{\text{def}}{=} \neg\,\langle a\rangle\, \top \vee \langle a\rangle\, \neg\varphi \\
\neg\mu(X_i = \varphi_i) \text{ in } \psi &\stackrel{\text{def}}{=} \mu(X_i = \neg\varphi_i\{\overline{[X_i/\neg X_i]}\}) \text{ in } \neg\psi\{\overline{[X_i/\neg X_i]}\}
\end{aligned}$$

In the following, we consider only closed formulas and write $[\![\varphi]\!]$ for $[\![\varphi]\!]_\varnothing$.

5.2 Adding Formulas to the Type System

We now have a language which allows us to describe sets of focused trees. If we take the type language of Sec. 4 and replace unit types with formulas, we obtain a type language for sequences of focused trees.

Recall that typechecking in the standard type system involved inferring a type for the expression using the rules then checking an inclusion between this type and the expected type. With formulas, the inclusion test translates into a satisfiability test (namely, $[\![\varphi]\!] \subseteq [\![\psi]\!]$ if and only if $\varphi \wedge \neg\psi$ is unsatisfiable), and as mentioned above, we have an efficient decision procedure for this test in this language. The remaining question is whether we can adapt the typing rules to this modified type language.

Looking at the type system, the change of type language does not affect the structural rules (T-SeqVar, T-Empty, T-Seq, T-If, T-For) since none of them looks inside unit types, so this will only change the rules for tree construction and navigation expressions. Let us start with the parent rule. It becomes this:

$$(\text{T-Parent''})\ \ \Gamma, \$v : \varphi \vdash \$v/\texttt{parent::}n : k(n) \wedge \langle 1 \rangle\, \mu X.(\varphi \vee \langle 2 \rangle\, X)$$

where $k(\sigma) \stackrel{\text{def}}{=} \sigma$ and $k(*) \stackrel{\text{def}}{=} \top$.

This type says that we can go from the node at focus to a node where φ is satisfied by going down once and then right a certain number of times. This is as precise as we can do.

Unfortunately, things do not go as perfectly well for other axes (except of course `self` which is trivial), because the other axes generate *sequences* of nodes. Thus the type of the expression should not be a single formula but a regular expression of formulas. Ideally, this regular expression would reflect somehow the ordering of differently-typed nodes in the sequence, but it is certainly not possible to be exact unless the sequence has a statically known length limit—indeed, every node in the sequence corresponds to a different focused tree and it is possible to set any of them apart from the others using a carefully crafted (and possibly large) formula. There is no most precise regular expression that we can look for. The problem of keeping the correct amount of ordering information is not at all straightforward and is beyond the scope of this paper; we intend to investigate it in future work.

The simplest way to infer a type for the other axes is to consider the sequence homogeneous. Note that it makes the new rules for `child` and `desc` slightly *less* precise than the standard rules (unless the resulting sequence is then iterated over, in which case the ordering information is lost anyway in the standard system).

We give below the typing rules for other axes using logic formulas:

$$(\text{T-Child''})\ \ E; \Gamma, \$v : \varphi \vdash \$v/\texttt{child::}n : (k(n) \wedge \mu X.(\langle \overline{1} \rangle\, \varphi \vee \langle \overline{2} \rangle\, X))*$$

$$(\text{T-Desc''})\ \ E; \Gamma, \$v : \varphi \vdash \$v/\texttt{desc::}n : (k(n) \wedge \mu X.(\langle \overline{1} \rangle\, (\varphi \vee X) \vee \langle \overline{2} \rangle\, X))*$$

$$(\text{T-Anc''})\ \ E; \Gamma, \$v : \varphi \vdash \$v/\texttt{anc::}n : (k(n) \wedge \langle 1 \rangle\, \mu X.(\varphi \vee \langle 1 \rangle\, X \vee \langle 2 \rangle\, X))*$$

$$(\text{T-Psibl})\ \ E; \Gamma, \$v : \varphi \vdash \$v/\texttt{psibl::}n : (k(n) \wedge \mu X.(\langle 2 \rangle\, \varphi \vee \langle 2 \rangle\, X))*$$

$$(\text{T-Nsibl})\ \ E; \Gamma, \$v : \varphi \vdash \$v/\texttt{nsibl::}n : (k(n) \wedge \mu X.(\langle \overline{2} \rangle\, \varphi \vee \langle \overline{2} \rangle\, X))*$$

Perhaps surprisingly, T-Tree, which was completely straightforward in the standard system, is the most difficult rule to adapt to formula types. Indeed, the semantics of the tree construction expression (rule R-Tree from Fig. 4) shows that the context part of the input focused trees is cut off and only the subtree part is kept. This looks deceptively simple because we represented focused trees as pairs, but logic formulas describe the focused trees as a whole, they do not

separate the context and subtree components. Thus we do not know how to infer a formula describing the new element from a regular expression of formulas describing the input node sequence.

We would like to emphasize the fact that, in an actual XQuery implementation, R-Tree is indeed the most complicated operation in our small fragment: the other ones mostly manipulate pointers to navigate in a data structure. This operation, in contrast, involves the deep copy of some specific, possibly overlapping parts of possibly several structures, and the assembling of the results into a brand new structure. If we take that into account, the fact that this operation is difficult to type when we use formula-based types which try to closely describe the structure of values is less surprising.

However, this shows limits of the purely formula-based approach. Since the standard approach and the formula-based approach are weak on different parts of the language, we can try combining the two.

5.3 Combining the two approaches

We saw that logic formulas are closer to the semantics of the XQuery language than standard XQuery types, and allow a reasonably precise typing of expressions involving non-downward navigation. However they are not well suited for use as the main types in a whole program, for at least two reasons. One is the difficulty of typing the element construction expression. Another one is type annotations: we stated our type-checking problem in terms of conformance to a specification. XQuery programs manipulate XML documents and, in general, both the input and output types will be specified as regular types, which are the standard for XML documents.

It therefore makes sense to investigate the possibility of a hybrid system mixing the two. It is possible, and quite straightforward, to translate standard unit types into logic formulas describing exactly the same set of trees, with or without the constraint that the node described by the unit types is at the root, as needed. We then can use the satisfiability solver to check whether a unit type is included in a logic type, as well as the converse. This means that the type-checking question described in Sec. 3 can be asked using standard regular types and answered using logic.

Furthermore, we can take advantage of the intrinsic compositionality of a type system: mostly, i.e. for all structural rules, this type-checking question is answered by splitting it into subquestions. We can answer some of the subquestions using logic and some of them using the standard type system.

6 Related work

Static typing for XQuery has been standardized by the W3C [2] and improved by [10]. [2] describes a type system which is polynomial (except for nested let clauses). Colazzo goes one step further by a thorough analysis of precision and complexity of this type-system and by introducing his own, more precise (but

exponential) type system. The type-system proposed by the W3C has been inspired by the seminal work found in [11], which is itself based on finite tree automata containment [12]. A more precise typing of `for` loops than what made it into the standard had been studied in [8]. None of these type systems supports non-downward navigation in XML trees, despite it being a part of the XQuery standard.

In the programming languages community, the XML type-checking problem has also been studied for other particular domain-specific languages such as CDuce [15], XSLT [16] or with specific transformers like transducers [17,18]. For a recent survey of related works on type-checking for XML, see [19] and references thereof.

The language fragment we decided to study formally is inspired by what can be found in the literature. Other formally-studied fragments include XQ ('core XQuery') [20], recently extended into XQ_{H} by Benedikt and Vu who added higher-order functions [21], and μXQ ('micro XQuery') [9], extended into $\mu\mathrm{XQ}^{+}$ ('mini XQuery') in [10]. The papers defining XQ and XQ_{H} focus on the semantics of the language and the complexity of query evaluation. The papers defining μXQ and $\mu\mathrm{XQ}^{+}$ focus on typing and correctness. There does not seem to be a significant difference between XQ and $\mu\mathrm{XQ}^{+}$, although the set of expressions is not exactly the same. μXQ does not include conditionals (`where` or, equivalently, `if-then-else` expressions), and XQ_{H} adds λ-abstractions and function application. None of these fragments includes axes other than `child` and `desc`. This allows XQ and XQ_{H} to have a formal semantics where items are simply trees without the need for a store, because it is not possible in these fragments to go from a node to its parent or siblings. Our 'navigational XQuery' fragment is basically XQ/$\mu\mathrm{XQ}^{+}$ with the upward and sideways axes added.

Very recently, Castagna et al. [13] have defined a larger fragment which they call ${\mathrm{XQ}_{\mathrm{H}}}^{+}$; it adds to XQ_{H} value-switch and type-switch expressions, as well as non-downward axes as we do. They do not study this fragment directly: rather, they define a translation from it to an extension of the $\mathbb{C}$Duce language, and study that extension, whose semantics is described by a fairly complicated system—not surprisingly since it has to integrate with $\mathbb{C}$Duce's pattern-matching, overloaded functions and other advanced features. In contrast, we tried to focus on just the unusual features of XQuery and keep things as simple as we can. Still, this work is probably the closest, in terms of objective, to the current paper, since it does treat backward axes in some way.

The logic language for XML trees which we described in Sec. 5 was described in more detail in [6], where it was used to check properties of XPath expressions. The typing rules we give in Sec. 5.2 are inspired by the way XPath subexpressions were translated into logic in that paper.

7 Conclusion

The work presented in this paper is not an accomplished type-checking system for XQuery, but what we hope will prove a useful basis and starting point to, finally, take *all* navigation expressions properly into account in such systems.

Our contribution is threefold. First, we defined a novel focused-tree-based operational semantics for a fragment of the XQuery language; this fragment was kept small here to concentrate on the core issues but can be easily extended. Second, we formulated the difficulty of typing XQuery expressions with backward axes in terms of a discrepancy between the language's semantics and type algebra, and demonstrated that this difficulty cannot be overcome without changing, at least locally, one of the two. Third, we proposed a logic-based type language to represent the missing information and discussed how to combine it with the existing one.

The last step of this work illustrates the design of type-checking systems based on rich tree logics, where subtyping is typically decided using a logical satisfiability solver (such as the one proposed in [6]). One advantage of this approach is to inherit from the well-studied characteristics of a generic logical formalism (in particular: expressivity, worst-case complexity, robustness with respect to extensions), as well as from the availability of implementations.

One direction for further research would be to do backward type inference [22], i.e. construct types that subexpressions must match from the required type of the whole expression, then check whether the initial typing environment matches the requirements. The logic-based type language seems natural for this purpose since it provides a succinct notation for the inferred subtypes and their contexts.

Another direction of research is to extend XQuery's type algebra for higher order functions and parametric polymorphism in the light of recent breakthroughs on the subject [23,24]. All these features combined will represent a much more powerful type system usable in practice and more attractive than W3C's proposed one, which gives too many false negatives. This will also revive programmers' interest in the language as the range of detectable errors in XQuery compilers will become much more accurate and wider.

References

1. Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language (Second edition). W3C Recommendation (2010) `http://www.w3.org/TR/xquery/`.
2. Draper, D., Dyck, M., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: XQuery 1.0 and XPath 2.0 formal semantics, W3C recommendation (December 2010) http://www.w3.org/TR/xquery-semantics/.
3. Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., Siméon, J.: XQuery update facility 1.0, W3C recommendation (March 2011) http://www.w3.org/TR/xquery-update-10/.

4. Robie, J., Chamberlin, D., Dyck, M., Snelson, J.: XQuery 3.0: An XML Query Language. W3C Last Call Working (July 2013) http://www.w3.org/TR/xquery-30/.
5. Huet, G.P.: The zipper. J. Funct. Program. **7**(5) (1997) 549–554
6. Genevès, P., Layaïda, N., Schmitt, A.: Efficient static analysis of XML paths and types. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. PLDI '07, New York, NY, USA, ACM (2007) 342–351
7. Wadler, P.: XQuery: A typed functional language for querying XML. In Jeuring, J., Jones, S.L.P., eds.: Advanced Functional Programming. Volume 2638 of Lecture Notes in Computer Science., Springer (2002) 188–212
8. Fernández, M.F., Siméon, J., Wadler, P.: A semi-monad for semi-structured data. In den Bussche, J.V., Vianu, V., eds.: ICDT. Volume 1973 of Lecture Notes in Computer Science., Springer (2001) 263–300
9. Colazzo, D., Ghelli, G., Manghi, P., Sartiani, C.: Types for path correctness of XML queries. In Okasaki, C., Fisher, K., eds.: ICFP, ACM (2004) 126–137
10. Colazzo, D., Sartiani, C.: Precision and complexity of XQuery type inference. In: Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark. (2011) 89–100
11. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Trans. Internet Technol. **3**(2) (May 2003) 117–148
12. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Trans. Program. Lang. Syst. **27**(1) (2005) 46–90
13. Castagna, G., Im, H., Nguyen, K., Benzaken, V.: A core calculus for XQuery 3.0. http://www.pps.univ-paris-diderot.fr/~gc/papers/xqueryduce.pdf (apr 2013)
14. Genevès, P., Layaïda, N., Quint, V.: Impact of XML schema evolution. ACM Trans. Internet Technol. **11** (July 2011) 4:1–4:27
15. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. In: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming. ICFP '03, New York, NY, USA, ACM (2003) 51–63
16. Kirkegaard, C., Møller, A., Schwartzbach, M.I.: Static analysis of XML transformations in Java. IEEE Transactions on Software Engineering **30**(3) (March 2004) 181–192
17. Maneth, S., Berlea, A., Perst, T., Seidl, H.: XML type checking with macro tree transducers. In: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. PODS '05, New York, NY, USA, ACM (2005) 283–294
18. Maneth, S., Perst, T., Seidl, H.: Exact XML type checking in polynomial time. In Schwentick, T., Suciu, D., eds.: ICDT. Volume 4353 of Lecture Notes in Computer Science., Springer (2007) 254–268
19. Benzaken, V., Castagna, G., Hosoya, H., Pierce, B.C., Vansummeren, S.: XML typechecking. In Liu, L., Özsu, M.T., eds.: Encyclopedia of Database Systems. Springer US (2009) 3646–3650
20. Koch, C.: On the complexity of nonrecursive XQuery and functional query languages on complex values. ACM Trans. Database Syst. **31**(4) (December 2006) 1215–1256
21. Benedikt, M., Vu, H.: Higherorder functions and structured datatypes. In: WebDB: Proceedings of the 15th International Workshop on the Web and Databases. (2012) 43–48

22. Milo, T., Suciu, D., Vianu, V.: Typechecking for XML transformers. J. Comput. Syst. Sci. **66**(1) (2003) 66–97
23. Gesbert, N., Genevès, P., Layaïda, N.: Parametric polymorphism and semantic subtyping: the logical connection. In: ICFP'11, Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ACM (sep 2011) 107–116
24. Castagna, G., Xu, Z.: Set-theoretic foundation of parametric polymorphism and subtyping. In: ICFP'11, Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ACM (sep 2011) 94–106