

XQuery and Static Typing: Tackling the Problem of Backward Axes

Pierre Genevès, Nils Gesbert

▶ **To cite this version:**

Pierre Genevès, Nils Gesbert. XQuery and Static Typing: Tackling the Problem of Backward Axes. ICFP (International Conference on Functional Programming), ACM SIGPLAN, Aug 2015, Vancouver, Canada. 10.1145/2784731.2784746 . hal-01082635v2

HAL Id: hal-01082635

https://inria.hal.science/hal-01082635v2

Submitted on 7 May 2015 (v2), last revised 29 Dec 2016 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

XQuery and Static Typing: Tackling the Problem of Backward Axes

Pierre Genevès Nils Gesbert

Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
Inria
pierre.geneves@cnrs.fr nils.gesbert@grenoble-inp.fr

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 navigation expressions using 'backward axes,' which return e.g. the siblings of a node, impossible to type.

We discuss how to handle this discrepancy by improving the type system. We describe a logic-based language of extended types able to represent inner tree nodes and show how it can dramatically increase the precision of typing for navigation expressions. We describe how inclusion between these extended types and the classical regular tree types can be decided, allowing a hybrid system combining both type languages. The result is a net increase in precision of typing.

Keywords XQuery, XML, Regular tree types, Mu-calculus

1. Introduction

XQuery is a functional language with some unusual features. The standard which defines it [6, 17] 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 it is 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. 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 $\mathrm{DynEnv} \vdash \mathit{Expr} \Rightarrow \mathit{Val}$, 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[1]; 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 language (Sec. 3.1): 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 can only be 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. 3.2.1) a logic whose formulas denote sets of *focused trees* rather than just of trees, and discuss how it can be combined with the existing types to yield a precise type system.

2. Syntax and Semantics of an XQuery Core

2.1 Values

2.1.1 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 [34]), 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.

[1] Expressions used to make persistent changes to instances in the XQuery data model are defined as a separate extension of the language [33].

[Copyright notice will appear here once 'preprint' option is removed.]

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.

2.1.2 Focused Trees

In order to represent references to nodes of immutable trees, we use *focused trees*, inspired by Huet's Zipper data structure [26] in the manner of [21]. 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.

t	$::=$	$\sigma[tl]$	tree
tl	$::=$	$\epsilon \mid t :: tl$	list of trees
c	$::=$	$\mathsf{Top} \mid (tl, c[\sigma], tl)$	context
f	$::=$	(t, c)	focused tree

A focused tree (t, c) is a pair consisting of a tree t and its context c. The context is Top if the current tree is at the root. Otherwise, it is of the form $(tl, c[\sigma], tl)$ and 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 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, $f\,\langle \mathtt{1} \rangle$ changes the focus to the first child of the current tree, $f\,\langle \mathtt{2} \rangle$ changes the focus to the next sibling of the current tree, $f\,\langle \overline{\mathtt{1}} \rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f\,\langle \overline{\mathtt{2}} \rangle$ changes the focus to the previous sibling. Formally, we have:

DEFINITION 1.

$$(\sigma[t :: tl], c)\,\langle \mathtt{1} \rangle \stackrel{def}{=} (t, (\epsilon, c[\sigma], tl))$$

$$(t, (tl_l, c[\sigma], t' :: tl_r))\,\langle \mathtt{2} \rangle \stackrel{def}{=} (t', (t :: tl_l, c[\sigma], tl_r))$$

$$(t, (\epsilon, c[\sigma], tl))\,\langle \overline{\mathtt{1}} \rangle \stackrel{def}{=} (\sigma[t :: tl], c)$$

$$(t', (t :: tl_l, c[\sigma], tl_r))\,\langle \overline{\mathtt{2}} \rangle \stackrel{def}{=} (t, (tl_l, c[\sigma], t' :: tl_r))$$

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 [17]. 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 replace these pointers with focused trees, as described in the previous subsection, which removes the need for a store[2]. 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. 5).

e	$::=$		expression
		$\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : x$	XML element
	$\mid$	ϵ	empty sequence
	$\mid$	e, e	seq. concatenation
	$\mid$	$\texttt{for}\ \$v\ \texttt{in}\ e\ \texttt{return}\ e$	for loop
	$\mid$	$\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e\ \texttt{return}\ e$	variable binding
	$\mid$	$\texttt{if non-empty}(e)\ \texttt{then}\ e\ \texttt{else}\ e$	existence test
	$\mid$	$\$v/axis::n$	tree navigation
	$\mid$	$\$v$	item variable
	$\mid$	$\$\overline{v}$	sequence variable
n	$::=$	$\sigma \mid *$	label or wildcard
s	$::=$	$\epsilon \mid f :: s$	value sequence

$$\mathcal{E} ::= [] \mid \texttt{<}\sigma\texttt{>}\{\mathcal{E}\}\texttt{</}\sigma\texttt{>} : x \mid \texttt{for}\ \$v\ \texttt{in}\ \mathcal{E}\ \texttt{return}\ e \mid \texttt{if non-empty}(\mathcal{E})\ \texttt{then}\ e\ \texttt{else}\ e \mid \mathcal{E}, e \mid s, \mathcal{E}$$

Figure 1. Navigational core of XQuery.

The navigational XQuery fragment we consider is described by the abstract syntax shown in Fig. 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 $f_1 :: \ldots :: f_n :: \epsilon$.

The XML element construction expression we include in our core syntax represents a combination of XQuery's element constructor and `validate` expressions. In XQuery, indeed, the result of a constructor expression of the form $\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>}$ is always considered untyped (both statically and dynamically) unless it is validated. The `validate` expression may include an explicit type annotation or not; if not, a type corresponding to the element name is looked for in the environment. This expression then checks, at runtime, whether the constructed element conforms to the expected type: if yes, it returns the element with the required dynamic type; if not, a dynamic type error is triggered.

We represent all this by the expression $\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : x$, where x refers to a type u defined in an external environment, as will be defined in Section 3.1. Translation from XQuery into this core is as follows:

- untyped element construction is represented by $\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : \mathsf{AnyElt}$;
- a `validate` expression with explicit type annotation x is represented by $\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : x$;
- for a `validate` expression without annotation, we assume a mapping from Σ to type references x (obtained from e.g. a DTD) is available, and represent this expression as: $\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : x(\sigma)$.

We do not include boolean expressions in our fragment; however XQuery allows writing `if-then-else` expressions where the condition evaluates to a sequence of element nodes. The meaning of such an expression is an emptiness test on the sequence; we include it, and write `if non-empty` explicitly for clarity.

We distinguish variables $\$v$ bound by `for` loops from variables $\$\overline{v}$ bound by `let` expressions. The former are bound to single tree nodes ("items" in XQuery terminology) whereas the latter are bound to possibly empty sequences of nodes. Path navigation expressions can only start from an item variable[3].

[2] Remark that in full XQuery, a focused tree is not enough to uniquely identify a node, because the store may contain several identical trees, whose nodes will have different identifiers. In order to be complete with this respect, we would thus have to use as values not just focused trees, but rather pairs of a focused tree and a tree identifier. However this is not relevant for the fragment we study here, which does not include an identity test.

[3] The expression $\$\overline{v}/axis::n$ exists in XQuery, but its semantics is defined as $\texttt{for}\ \$v\ \texttt{in}\ \$\overline{v}\ \texttt{return}\ \$v/axis::n$ plus a sort operation on the result.

$$
\begin{array}{lcll}
u & ::= & \texttt{element}\ n\ \{\tau\} & \text{unit type} \\
n & ::= & \sigma \mid * & \text{name test} \\
\tau & ::= & u \mid () \mid \tau, \tau \mid (\tau \mid \tau) \mid \tau* \mid x & \text{sequence type}
\end{array}
$$

Figure 3. XQuery Types

2.3 Reduction Semantics

Figure 2 gives reduction rules defining a small-step operational semantics for the focused-tree-based navigational XQuery fragment we consider. These rules rely on the evaluation contexts $\mathcal{E}$ defined in Fig. 1 to allow reduction of a subexpression. In addition to expressions from the syntax of Fig. 1, runtime expressions can contain focused tree and sequence literals (only sequences are values: focused tree literals reduce to sequences by R-SINGLETON) as well as the dynamic type error ω.

Rules R-TREE and R-TREEERROR correspond, as described in the previous section, to a combination of tree construction and dynamic type check: the tree constructed in the premise is the same in both rules, and whether it conforms to the type annotation ($t \in [\![x]\!]$) determines which rule applies; $[\![x]\!]$ will be defined formally in Section 3.1.

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 for 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.

3. Type system

Now that we have a simple formal semantics for a core fragment of XQuery, we want to design a type system able to ensure statically that a given program will never go wrong at runtime. With the reduction rules we have, a syntactically correct program cannot get stuck, and the only kind of expression which can yield a dynamic type error is tree construction <σ>{e}</σ> : x. However, since, in such an expression, e can be any expression and x refers to an external type specification, the problem of type safety is here equivalent to the fully general problem of statically checking that an arbitrary expression will always reduce to a value conforming to an arbitrary specification. This means that if we extend the core to include e.g. type-annotated function definitions we will be able to typecheck them with the same system. We now discuss different type languages and how to construct a type system which is sound and as precise as possible, i.e. accepts only correct programs but as many of them as possible.

3.1 Regular Tree Types

As is customary in the literature ([15, 16, 19, 38] for instance), we use a slight variant of XDuce's type language [23, 24], 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. These references may be mutually recursive, but recursion must be guarded by an element constructor. In other words, if we repeatedly replace all references appearing at top level (i.e. not inside a unit type) with their bindings, this process must terminate after a few iterations and yield a regular expression of unit types. As an additional restriction, these regular expressions must be composed of mutually exclusive unit types and be *1-unambiguous* [7]. This constraint is standard: it comes from XML Schema. In the following, we assume this restriction is respected by all types in E.

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 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.

To give a formal definition, we first define the denotation of a type depending on a function d mapping references to sets of forests. We then define the variable denotation d_E corresponding to a type environment E. The denotation of a type containing references is only defined if an environment providing bindings for all these references is given.

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

Let $E = (x_i = \tau_i)_{i \in I}$. Given two mappings d_1 and d_2 from the x_i to sets of forests, we say that d_1 is smaller than d_2 if: $\forall i \in I, d_1(x_i) \subseteq d_2(x_i)$. The variable denotation d_E corresponding to the type environment E is defined as the smallest mapping such that: $\forall i \in I, d_E(x_i) = [\![\tau_i]\!]_{d_E}$.

In the following, we always assume the environment E is well-formed and contains bindings for all references appearing in the types, and we write $[\![\tau]\!]$ as a shorthand for $[\![\tau]\!]_{d_E}$. We often assume, as well, that references x are implicitly replaced with their bindings at toplevel, so that a type τ is really a regular expression of unit types. We also consider that E always contains the type of all elements, AnyElt, defined as AnyElt $=$ `element` $*$ {AnyElt$*$}.

3.2 Types for Focused Trees

All the definitions we gave about regular tree types up to now are standard. The standard notion of a value (sequence of tree nodes) *matching* a type can be formally defined as follows when nodes are represented as focused trees:

DEFINITION 2. *The* focused-tree interpretation $[\![\tau]\!]\uparrow$ *of a type* τ *is the set* $\{[(t_1, c_1) \ldots (t_n, c_n)] \mid [t_1 \ldots t_n] \in [\![\tau]\!]\}$. *A value* s *is said to* match *type* τ *if* $s \in [\![\tau]\!]\uparrow$.

As we can see, regular tree types naturally denote sequences of trees, and their interpretation is lifted to sequences of focused trees by simply ignoring the context part. The static type system defined in the XQuery standard, and its various improvements proposed in the literature, rely only on this type language, and thus associate to each expression such a regular tree type and nothing else. This means that they do not have any information about the context of the nodes in the sequence the expression will reduce to.

So in such a system, if we consider, for example, the expression `for` $\$v$ `in` e `return` $\$v$`/nsibl::*`, its type has to be deduced from

R-TREE
$$\frac{t = \sigma[t_1 :: t_2 :: \ldots :: t_n :: \epsilon] \qquad t \in [\![x]\!]}{\texttt{<}\sigma\texttt{>}\{[(t_1, c_1), (t_2, c_2), \ldots, (t_n, c_n)]\}\texttt{</}\sigma\texttt{>} : x \longrightarrow [(t, \mathsf{Top})]}$$

R-TREEERROR
$$\frac{t = \sigma[t_1 :: t_2 :: \ldots :: t_n :: \epsilon] \qquad t \notin [\![x]\!]}{\texttt{<}\sigma\texttt{>}\{[(t_1, c_1), (t_2, c_2), \ldots, (t_n, c_n)]\}\texttt{</}\sigma\texttt{>} : x \longrightarrow \omega}$$

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

R-FOREMPTY
$$\texttt{for } \$v \texttt{ in } \epsilon \texttt{ return } e \longrightarrow \epsilon$$

R-SINGLETON
$$f \longrightarrow [f]$$

R-CONCAT
$$[f_1, \ldots, f_n], [f'_1, \ldots, f'_{n'}] \longrightarrow [f_1, \ldots, f_n, f'_1, \ldots f'_{n'}]$$

R-LET
$$\texttt{let } \$\overline{v} \texttt{ := } s \texttt{ return } e \longrightarrow e\left[{}^{s}/_{\$\overline{v}}\right]$$

R-IFF
$$\texttt{if non-empty}(\epsilon) \texttt{ then } e_1 \texttt{ else } e_2 \longrightarrow e_2$$

R-IFT
$$\texttt{if non-empty}(f :: s) \texttt{ then } e_1 \texttt{ else } e_2 \longrightarrow e_1$$

R-NOPARENT
$$(t, \mathsf{Top})/\texttt{parent::}n \longrightarrow \epsilon$$

R-NOCHILD
$$(\sigma[\epsilon], c)/\texttt{child::}n \longrightarrow \epsilon$$

R-NONSIBL
$$(t, (tl, \sigma[c], \epsilon))/\texttt{nsibl::}n \longrightarrow \epsilon$$

R-NOPSIBL
$$(t, (\epsilon, \sigma[c], tl))/\texttt{psibl::}n \longrightarrow \epsilon$$

R-NOANC
$$(t, \mathsf{Top})/\texttt{anc::}n \longrightarrow \epsilon$$

R-SELFSTAR
$$f/\texttt{self::}* \longrightarrow [f]$$

R-SELFMATCH
$$(\sigma[tl], c)/\texttt{self::}\sigma \longrightarrow [(\sigma[tl], c)]$$

R-SELFDIFF
$$\frac{\sigma \neq \sigma'}{(\sigma[tl], c)/\texttt{self::}\sigma' \longrightarrow \epsilon}$$

R-PARENT
$$\frac{f' = f\,\langle\overline{\mathbf{1}}\rangle}{f/\texttt{parent::}n \longrightarrow f'/\texttt{self::}n}$$

R-PSPARENT
$$\frac{f' = f\,\langle\overline{\mathbf{2}}\rangle}{f/\texttt{parent::}n \longrightarrow f'/\texttt{parent::}n}$$

R-CHILD
$$\frac{f' = f\,\langle\mathbf{1}\rangle}{f/\texttt{child::}n \longrightarrow f'/\texttt{self::}n, f'/\texttt{nsibl::}n}$$

R-NSIBL
$$\frac{f' = f\,\langle\mathbf{2}\rangle}{f/\texttt{nsibl::}n \longrightarrow f'/\texttt{self::}n, f'/\texttt{nsibl::}n}$$

R-PSIBL
$$\frac{f' = f\,\langle\overline{\mathbf{2}}\rangle}{f/\texttt{psibl::}n \longrightarrow f'/\texttt{psibl::}n, f'/\texttt{self::}n}$$

R-ANC
$$\frac{f' = f\,\langle\overline{\mathbf{1}}\rangle}{f/\texttt{anc::}n \longrightarrow f'/\texttt{anc::}n, f'/\texttt{self::}n}$$

R-PSANC
$$\frac{f' = f\,\langle\overline{\mathbf{2}}\rangle}{f/\texttt{anc::}n \longrightarrow f'/\texttt{anc::}n}$$

R-DESC
$$f/\texttt{desc::}n \longrightarrow \texttt{for } \$v \texttt{ in } f/\texttt{child::}* \texttt{ return } \$v/\texttt{self::}n, \$v/\texttt{desc::}n$$

R-CONTEXT
$$\frac{e_1 \longrightarrow e_2 \qquad e_2 \neq \omega}{\mathcal{E}[e_1] \longrightarrow \mathcal{E}[e_2]}$$

R-ERROR
$$\frac{e_1 \longrightarrow \omega}{\mathcal{E}[e_1] \longrightarrow \omega}$$

Figure 2. Reduction Rules for the Navigational XQuery Fragment

the regular type τ of e. What we know is that when e reduces to a value $[f_1 \ldots f_n]$, this value will match τ. Looking at the reduction rules, we can see that the final result of the expression depends only of the $f_i\,\langle\mathbf{2}\rangle$; and if $f_i = (t_i, c_i)$, $f_i\,\langle\mathbf{2}\rangle$ depends mainly of c_i, whereas τ only contains information about t_i. It is thus impossible to say anything interesting about the result without having more information on e than its regular tree type. A consequence is that type systems for XQuery based only on this type language give to this expression the most general type (AnyElt$*$), and thus always fail to typecheck it unless no requirement at all was made on the result.

In order to solve this problem, we need to enrich the language of types to also describe the context part of focused trees. We propose to do this using logic formulas.

3.2.1 A Tree Logic

In order to describe sets of focused trees rather than just sets of trees, we use a variant of the logic language defined in [22]. Its syntax is given in Fig. 4, where $a \in \{\mathbf{1}, \mathbf{2}, \overline{\mathbf{1}}, \overline{\mathbf{2}}\}$ 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 [22].

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. We use $\mu X.\varphi$ as an abbreviation for $\mu(X = \varphi)$ in φ. For example, the formula $\mu X.b \vee \langle\overline{\mathbf{2}}\rangle\, X$ means that either the current node or some previous sibling is labeled b.

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

Figure 4. Logic formulas

The interpretation of a logical formula is the set of focused trees such that the formula is satisfied at the current node. We give the formal definition in Fig. 5, where $\mathcal{F}$ is the set of all focused trees and $\texttt{nm}(f)$ is the label at the current node of f.

In the following, we consider only closed formulas and write $\langle\!\langle\varphi\rangle\!\rangle$ for $\langle\!\langle\varphi\rangle\!\rangle_\emptyset$.

3.2.2 Adding formulas to regular tree types

We now have a type language which allows us to describe sets of focused trees. Since the values of the language are *sequences* of focused trees, we still want regular expressions to represent them; we simply enrich the regular expressions of unit types defined in Sec. 3.1 by associating to each unit type a formula. The enriched types are thus regular expressions of pairs of a unit type and a formula, defined as follows:

$$\rho ::= (\varphi, u) \mid () \mid \rho, \rho \mid (\rho \mid \rho) \mid \rho*$$

Note that unit types are *not* enriched in depth, i.e. they are still of the form $\texttt{element}\ n\ \{\tau\}$ where τ does not contain formulas. This is because τ is here actually used to describe a list of trees and not of focused trees: focused trees are of the form $(\sigma[tl], c)$ (Sec. 2.1.2).

$$\langle\!\langle \top \rangle\!\rangle_V \stackrel{\text{def}}{=} \mathcal{F} \qquad \langle\!\langle \langle a \rangle\, \varphi \rangle\!\rangle_V \stackrel{\text{def}}{=} \{ f \,\langle \overline{a} \rangle \mid f \in \langle\!\langle \varphi \rangle\!\rangle_V \}$$

$$\langle\!\langle X \rangle\!\rangle_V \stackrel{\text{def}}{=} V(X) \qquad \langle\!\langle \neg \langle a \rangle \top \rangle\!\rangle_V \stackrel{\text{def}}{=} \{ f \mid f \,\langle a \rangle \text{ undefined} \}$$

$$\langle\!\langle \sigma \rangle\!\rangle_V \stackrel{\text{def}}{=} \{ f \mid \mathtt{nm}(f) = \sigma \} \qquad \langle\!\langle \varphi \vee \psi \rangle\!\rangle_V \stackrel{\text{def}}{=} \langle\!\langle \varphi \rangle\!\rangle_V \cup \langle\!\langle \psi \rangle\!\rangle_V$$

$$\langle\!\langle \neg\sigma \rangle\!\rangle_V \stackrel{\text{def}}{=} \{ f \mid \mathtt{nm}(f) \neq \sigma \} \qquad \langle\!\langle \varphi \wedge \psi \rangle\!\rangle_V \stackrel{\text{def}}{=} \langle\!\langle \varphi \rangle\!\rangle_V \cap \langle\!\langle \psi \rangle\!\rangle_V$$

$$\langle\!\langle \mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi \rangle\!\rangle_V \stackrel{\text{def}}{=}$$
$$\text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, \langle\!\langle \varphi_j \rangle\!\rangle_{V[\overline{T_i/X_i}]} \subseteq T_j \} \quad \text{in}$$
$$\text{let } (U_j) = \left(\textstyle\bigcap_{(T_i)\in S} T_j\right)_{j\in I} \quad \text{in} \quad \langle\!\langle \psi \rangle\!\rangle_{V[\overline{U_i/X_i}]}$$
$$\text{where } V[\overline{T_i/X_i}](X) \stackrel{\text{def}}{=} V(X) \text{ if } X \notin \{X_i\} \text{ and } T_i \text{ if } X = X_i.$$

Figure 5. Interpretation of formulas

In a pair $(\mathtt{element}\ n\ \{\tau\}, \varphi)$, n describes σ, τ describes tl, and c is described only by φ[4]. The list tl is a list of subtrees which are all siblings in the same tree structure; it is very different from a sequence value s where each node in the sequence has its own context independently of the others, although the standard type system does not distinguish the two.

The interpretation of a pair (φ, u) is defined as the set of focused trees which match both components, i.e. :

$$[\![(\varphi, u)]\!] = \{(t, c) \mid t \in [\![u]\!] \text{ and } (t, c) \in \langle\!\langle \varphi \rangle\!\rangle\}$$

From this, the interpretation of regular expressions of pairs in terms of sets of sequences of focused trees is then defined in the obvious manner.

3.3 Typing rules

We present in Figures 7 and 8 a type system for the navigational core of XQuery which makes use of the additional information in our enriched types.

The rules use type environments E, which contain the possibly mutually recursive definitions of named types (see Sec. 3.1), and typing environments Γ which map sequence variables to sequence types and iteration variables to **single pairs** of a formula and a unit type (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.

Rules T-ITEMVAR, T-SEQVAR, T-EMPTY, T-LET and T-SEQ are straightforward. Rule T-FOR uses an auxiliary judgement, taken from [19]. We write $E; \Gamma \vdash \mathtt{for}\ \$v : \rho\ \mathtt{return}\ e : \rho'$ if, in environment Γ, when the bound variable of an iteration $\$v$ has type ρ then the body e of the iteration has type ρ'. This typing of `for` expressions is more precise than the one found in the standard type-system [17] (as explained in Section 5).

Rule T-TREE involves a subtyping check: recall that the tree constructor includes a `validate` operation, and we want this rule to detect whether this operation will succeed at runtime or not. Note that in this subtyping check, the right-hand type is not formula-enriched. Indeed, it comes from the type specification of the element being constructed, and as we can see in Rules R-TREE/R-TREEERROR (Fig. 2), the context of this element is always Top and the original context of the component nodes is erased. The check we have to make is thus between a focused-tree sequence type and a *tree* sequence type, ignoring the contexts in the left-hand type. We detail this in the next section.

The `if-then-else` expression can be typed with more or less precision depending on what appears in the condition. To get the best precision, the four rules presented in Fig. 7 must be tried in the order they are listed. T-IFANY is the most general one and simply gives to the expression the disjunction of the types of the `then` and `else` clauses; this is the usual rule for conditional expressions, and the one in the standard type system. T-IFEMPTY and T-IFNONEMPTY are straightforward improvements in the case of the emptiness check, which do not need enriched types; the auxiliary predicate *nullable*() used in T-IFNONEMPTY indicates whether the empty sequence belongs to the denotation of the regular expression (which is a simple linear syntactic check).

The improvements in precision that formulas allow are in rules T-IFAXIS and T-AXIS, where navigation expressions appear. They make use of the auxiliary functions defined on Fig. 8. k translates a node test n into a formula depending whether it is a wild-card or specific label. The next two functions, `navigate-`*axis*(χ) and `has-`*axis*(χ), are functions from formulas to formulas and in

[4] which can additionally contain information about both σ and tl as well

$$\mathsf{filter}(\mathtt{()}, n) = \mathtt{()}$$
$$\mathsf{filter}(\mathtt{element} * \{\tau\}, n) = \mathtt{element} * \{\tau\}$$
$$\mathsf{filter}(\mathtt{element}\ n\ \{\tau\}, *) = \mathtt{element}\ n\ \{\tau\}$$
$$\mathsf{filter}(\mathtt{element}\ \sigma\ \{\tau\}, \sigma) = \mathtt{element}\ \sigma\ \{\tau\}$$
$$\mathsf{filter}(\mathtt{element}\ \sigma\ \{\tau\}, \sigma') = \mathtt{()} \text{ if } \sigma \neq \sigma'$$
$$\mathsf{filter}(\tau_1 \mid \tau_2, n) = \mathsf{filter}(\tau_1, n) \mid \mathsf{filter}(\tau_2, n)$$
$$\mathsf{filter}((\tau_1, \tau_2), n) = \mathsf{filter}(\tau_1, n), \mathsf{filter}(\tau_2, n)$$
$$\mathsf{filter}(\tau*, n) = \mathsf{filter}(\tau, n)*$$

$$\mathsf{children}(\mathtt{element}\ n\ \{\tau\}) = \tau$$

$$\mathsf{children}(u_1 \mid u_2) = \mathsf{children}(u_1) \mid \mathsf{children}(u_2)$$
$$\mathsf{children}(\tau_1, \tau_2) = \mathsf{children}(\tau_1), \mathsf{children}(\tau_2)$$
$$\mathsf{children}(\tau*) = \mathsf{children}(\tau)*$$

The descendant-or-self function dos first computes the set of all unit types which may appear as descendants of the original type; this (finite) set is then converted into a regular type by putting all these unit types in a big disjunction, to which a Kleene star is added. Formally :

$$\mathsf{setdos}(\mathtt{element}\ n\ \{\tau\}) = \{\mathtt{element}\ n\ \{\tau\}\} \cup \mathsf{setdos}(\tau)$$
$$\mathsf{setdos}(\mathtt{()}) = \emptyset$$
$$\mathsf{setdos}(\tau_1 \mid \tau_2) = \mathsf{setdos}(\tau_1) \cup \mathsf{setdos}(\tau_2)$$
$$\mathsf{setdos}(\tau_1, \tau_2) = \mathsf{setdos}(\tau_1) \cup \mathsf{setdos}(\tau_2)$$
$$\mathsf{setdos}(\tau*) = \mathsf{setdos}(\tau)$$

$$\mathsf{dos}(\tau) = (u_1 \mid \cdots \mid u_n) * \text{ where } \{u_1, \ldots, u_n\} = \mathsf{setdos}(\tau)$$

As remarked by Colazzo and Sartiani [15], for non-recursive types a more precise definition of dos can be given:

$$\mathsf{dos}(\mathtt{()}) = \mathtt{()}$$
$$\mathsf{dos}(\mathtt{element}\ n\ \{\tau\}) = \mathtt{element}\ n\ \{\tau\}, \mathsf{dos}(\tau)$$
$$\mathsf{dos}(\tau_1 \mid \tau_2) = \mathsf{dos}(\tau_1) \mid \mathsf{dos}(\tau_2)$$
$$\mathsf{dos}(\tau_1, \tau_2) = \mathsf{dos}(\tau_1), \mathsf{dos}(\tau_2)$$
$$\mathsf{dos}(\tau*) = \mathsf{dos}(\tau)*$$

This is what we implemented in our prototype: when dos needs to be called, the definition used is decided depending whether the type is recursive.

Figure 6. Auxiliary functions of the XQuery standard type system

T-ItemVar
$$E;\Gamma,\$v:(\varphi,u)\vdash \$v:(\varphi,u)$$

T-SeqVar
$$E;\Gamma,\$\overline{v}:\rho\vdash \$\overline{v}:\rho$$

T-Empty
$$E;\Gamma\vdash \epsilon:()$$

T-Let
$$\frac{E;\Gamma\vdash e_1:\rho_1 \qquad E;\Gamma,\$\overline{v}:\rho_1\vdash e_2:\rho_2}{E;\Gamma\vdash \texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2:\rho_2}$$

T-Tree
$$\frac{(x=\texttt{element}\ n\ \{\tau\})\in E \qquad n=*\vee n=\sigma \qquad E;\Gamma\vdash e:\rho \qquad \rho<:\tau}{E;\Gamma\vdash (\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>}:x):(\mathrm{form}(x),\texttt{element}\ n\ \{\tau\})}$$

T-Seq
$$\frac{E;\Gamma\vdash e_1:\rho_1 \qquad E;\Gamma\vdash e_2:\rho_2}{E;\Gamma\vdash e_1,e_2:\rho_1,\rho_2}$$

T-IfEmpty
$$\frac{E;\Gamma\vdash e:() \qquad E;\Gamma\vdash e_2:\rho_2}{E;\Gamma\vdash \texttt{if non-empty}(e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\rho_2}$$

T-IfNonEmpty
$$\frac{E;\Gamma\vdash e:\rho \qquad \neg \mathit{nullable}(\rho) \qquad E;\Gamma\vdash e_1:\rho_1}{E;\Gamma\vdash \texttt{if non-empty}(e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\rho_1}$$

T-IfAxis
$$\frac{\begin{array}{c}E;\Gamma,\$v:(\varphi,u)\vdash \$v/\mathit{axis}::n:\mathsf{AnyElt}*\\ E;\Gamma,\$v:(\varphi\wedge \texttt{has-}\mathit{axis}(k(n)),u)\vdash e_1:\rho_1 \qquad E;\Gamma,\$v:(\varphi\wedge\neg\texttt{has-}\mathit{axis}(k(n)),u)\vdash e_2:\rho_2\end{array}}{E;\Gamma\vdash \texttt{if non-empty}(\$v/\mathit{axis}::n)\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\rho_1\mid\rho_2}$$

T-IfAny
$$\frac{E;\Gamma\vdash e:\mathsf{AnyElt}* \qquad E;\Gamma\vdash e_1:\rho_1 \qquad E;\Gamma\vdash e_2:\rho_2}{E;\Gamma\vdash \texttt{if non-empty}(e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\rho_1\mid\rho_2}$$

T-For
$$\frac{E;\Gamma\vdash e_1:\rho_1 \qquad E;\Gamma\vdash \texttt{for}\ \$v:\rho_1\ \texttt{return}\ e_2:\rho_2}{E;\Gamma\vdash \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2:\rho_2}$$

T-Axis
$$E;\Gamma,\$v:(\varphi,u)\vdash \$v/\mathit{axis}::n:\texttt{follow-axis}(\mathit{axis},(\varphi,u),n)$$

Auxiliary judgement for `for` loops [19]:

$$\frac{E;\Gamma,\$v:(\varphi,u)\vdash e:\rho'}{E;\Gamma\vdash \texttt{for}\ \$v:(\varphi,u)\ \texttt{return}\ e:\rho'} \qquad E;\Gamma\vdash \texttt{for}\ \$v:()\ \texttt{return}\ e:() \qquad \frac{E;\Gamma\vdash \texttt{for}\ \$v:\rho\ \texttt{return}\ e:\rho'}{E;\Gamma\vdash \texttt{for}\ \$v:\rho*\ \texttt{return}\ e:\rho'*}$$

$$\frac{E;\Gamma\vdash \texttt{for}\ \$v:\rho_1\ \texttt{return}\ e:\rho_1' \qquad E;\Gamma\vdash \texttt{for}\ \$v:\rho_2\ \texttt{return}\ e:\rho_2'}{E;\Gamma\vdash \texttt{for}\ \$v:\rho_1,\rho_2\ \texttt{return}\ e:\rho_1',\rho_2'}$$

$$\frac{E;\Gamma\vdash \texttt{for}\ \$v:\rho_1\ \texttt{return}\ e:\rho_1' \qquad E;\Gamma\vdash \texttt{for}\ \$v:\rho_2\ \texttt{return}\ e:\rho_2'}{E;\Gamma\vdash \texttt{for}\ \$v:\rho_1\mid\rho_2\ \texttt{return}\ e:\rho_1'\mid\rho_2'}$$

Figure 7. Typing Rules for the Navigational XQuery Fragment.

$$\begin{array}{rl} k(*) &= \top \\ \texttt{navigate-self}(\chi) &= \chi \\ \texttt{navigate-child}(\chi) &= \mu Z.\,\langle\bar{1}\rangle\,\chi\vee\langle\bar{2}\rangle\,Z \\ \texttt{navigate-nsibl}(\chi) &= \mu Z.\,\langle\bar{2}\rangle\,\chi\vee\langle\bar{2}\rangle\,Z \\ \texttt{navigate-psibl}(\chi) &= \mu Z.\,\langle 2\rangle\,\chi\vee\langle 2\rangle\,Z \\ \texttt{navigate-parent}(\chi) &= \langle 1\rangle\,\mu Z.\chi\vee\langle 2\rangle\,Z \\ \texttt{navigate-desc}(\chi) &= \mu Z.\,\langle\bar{1}\rangle\,(\chi\vee Z)\vee\langle\bar{2}\rangle\,Z \\ \texttt{navigate-anc}(\chi) &= \langle 1\rangle\,\mu Z.\chi\vee\langle 1\rangle\,Z\vee\langle 2\rangle\,Z \end{array} \qquad \begin{array}{rl} k(\sigma) &= \sigma \\ \texttt{has-self}(\chi) &= \chi \\ \texttt{has-child}(\chi) &= \texttt{navigate-parent}(\chi) \\ \texttt{has-nsibl}(\chi) &= \texttt{navigate-psibl}(\chi) \\ \texttt{has-psibl}(\chi) &= \texttt{navigate-nsibl}(\chi) \\ \texttt{has-parent}(\chi) &= \texttt{navigate-child}(\chi) \\ \texttt{has-desc}(\chi) &= \texttt{navigate-anc}(\chi) \\ \texttt{has-anc}(\chi) &= \texttt{navigate-desc}(\chi) \end{array}$$

$$\begin{array}{rl} \texttt{go-self}(u) &= u \\ \texttt{go-child}(u) &= \mathsf{children}(u) \\ \texttt{go-desc}(u) &= \mathsf{dos}(\mathsf{children}(u)) \\ \texttt{go-parent}(u) &= () \mid \mathsf{AnyElt} \\ \texttt{go-}\mathit{axis}(u) &= \mathsf{AnyElt}* \text{ for } \mathit{axis}\in\{\texttt{anc, psibl, nsibl}\} \end{array} \qquad \begin{array}{rl} \mathsf{distrib}(\chi,()) &= () \\ \mathsf{distrib}(\chi,u) &= (\chi,u) \\ \mathsf{distrib}(\chi,\tau_1,\tau_2) &= (\mathsf{distrib}(\chi,\tau_1),\mathsf{distrib}(\chi,\tau_2)) \\ \mathsf{distrib}(\chi,\tau_1\mid\tau_2) &= (\mathsf{distrib}(\chi,\tau_1)\mid\mathsf{distrib}(\chi,\tau_2)) \\ \mathsf{distrib}(\chi,\tau*) &= \mathsf{distrib}(\chi,\tau)* \end{array}$$

$$\frac{\psi=\texttt{navigate-}\mathit{axis}(\varphi\wedge\mathrm{form}(u))\wedge k(n) \qquad \tau=\mathsf{filter}(n,\texttt{go-}\mathit{axis}(u)) \qquad \varphi\wedge\neg\texttt{has-}\mathit{axis}(k(n))\text{ is satisfiable}}{\texttt{follow-axis}(\mathit{axis},(\varphi,u),n)=\mathsf{distrib}(\psi,\tau)}$$

$$\frac{\psi=\texttt{navigate-}\mathit{axis}(\varphi\wedge\mathrm{form}(u))\wedge k(n) \qquad \tau=\mathsf{filter}(n,\texttt{go-}\mathit{axis}(u)) \qquad \varphi\wedge\neg\texttt{has-}\mathit{axis}(k(n))\text{ is unsatisfiable}}{\texttt{follow-axis}(\mathit{axis},(\varphi,u),n)=(\mathsf{distrib}(\psi,\tau))\setminus\{()\}}$$

Figure 8. Auxiliary functions used to typecheck axes

some sense dual from each other. The first one constructs a formula which is true at all nodes reached by navigating *axis* from a node where χ is true; in a nutshell, this formula says that if we perform the navigation in the reverse direction then we must reach a node satisfying χ. The second one constructs a formula which is true if navigating *axis* from the current node reaches at least one node where χ is true. T-IFAXIS only uses `has-`*axis*(χ); it is an improvement over T-IFANY, possible in the case where the condition is a navigation expression. The `then` and `else` subexpressions can be checked with a refined environment because knowing whether the navigation expression yields an empty result or not gives additional information on $\$v$.

The function `go-`*axis*() is similar to `navigate-`*axis*() in purpose but works on unit types instead of formulas; this function corresponds to the standard XQuery type system. It uses operations children and dos ('descendant-or-self'), which are discussed e.g. in [15] together with filter. Their definition is recalled in Fig. 6, p. 5. The important point to notice is that when *axis* is not `self`, `child` or `desc`, the result of this function is extremely imprecise and basically useless, due to the fact that the original type contains no information on the context – in this case, having the formulas is crucial.

The other functions are used to re-combine formulas and unit types after performing navigation: indeed, `go-`*axis*(u) yields a regular expression whereas `navigate-`*axis*(χ) yields a single formula, which then must be distributed to all unit types in the regular expression, using distrib. This is done by the `follow-axis` function, which computes the type resulting from a navigation expression, and is used in T-AXIS. This function adds a further refinement: it is sometimes possible to detect by a satisfiability check (last premise) that the navigation expression cannot yield the empty sequence. In this case (second `follow-axis` rule), the empty sequence is removed from the type obtained (which is a simple operation on regular expressions).

3.4 Comparing classical and formula-based types

As we saw, in order to deal with the context-erasing tree construction operation, we need to be able to decide when an enriched type ρ is a subtype of a classical type τ. We define the subtyping relation semantically as $\rho <: \tau$ if $[\![\rho]\!] \subseteq [\![\tau]\!]\uparrow$. In order to decide this relation, we first need to compare unit types.

For this, we rely on the function $\mathrm{form}(u)$, which translates a classical unit type into a downward-only formula which is true at any tree node matching this unit type, regardless of its context. This function is formally defined on Fig. 9.

The definition uses an auxiliary operation $(\!|\tau|\!)^{\mathcal{V}}_{\varphi}$ which translates a regular expression τ of unit types into a *single* formula. This operation additionally updates a pair $\mathcal{V}$ of mappings used for downward recursion. The formula represents the set of focused trees such that there is a sequence of siblings starting at the current node which matches τ and ends with a node satisfying φ. This parameter φ allows us to write a recursive definition in the case of concatenation: when translating τ_1, τ_2, the regular expression τ_2 is translated first and the result is used to build a formula that the last node in the sequence matching τ_1 will have to satisfy. The predicate *nullable*(τ) is used for checking whether $\epsilon \in [\![\tau]\!]$.

The treatment of recursion in this translation needs specific care. Indeed, in classical types the same type reference may occur in different sequences, but because the translation of types into formulas integrates the tail of the sequence, this reference will not correspond to the same formula each time. It is therefore not possible to simply translate type references into fixpoint variables. However, just expanding references every time they are encountered would not terminate: we still need to introduce recursion in the translation. Since recursion in the original type must be guarded by element constructors, what we do is to translate *unit types* with fixpoint variables: the translation operation updates a pair $\mathcal{V} = (U, V)$ where U is a mapping from unit types to variables and V a mapping from variables to formulas. Whenever a unit type $u =$ `element` $n\ \{\tau\}$ which is not already in U is encountered, a fresh variable X is created and associated with u in U. The content model type τ is then translated using this updated U and the result of the translation is associated to X in V. At the end of the whole translation, a fixpoint formula is generated from all the bindings in V and the result formula.

The environment E and the substitution of references x with their bindings are left implicit in the definition. The number of such substitutions which needs to be done during the translation is finite since:

- the guardedness constraint on recursion in E implies that any sequence type reduces after a finite number of such substitutions to a type without references at toplevel;
- the translation never looks into the same unit type twice, so references in its content need only be expanded once.

LEMMA 1 (Translation correctness). *Let u be a unit type. Then* $[\![\mathrm{form}(u)]\!] = [\![u]\!]\uparrow$.

Proof. Immediately follows from the correctness of the translation from unranked regular expression types into binary form (proved in Appendix A of [24]), and a straightforward translation of the latter into logical formulas, as first introduced in [21]. □

This translation allows us to compare a formula φ and a unit type u by testing the satisfiability of the formula $\varphi \wedge \neg\mathrm{form}(u)$; indeed, $\langle\!\langle \varphi \wedge \neg\mathrm{form}(u) \rangle\!\rangle = \emptyset$ if and only if any focused tree satisfying φ matches u. Furthermore, it allows us to convert a regular expression ρ of pairs (φ, u) into a regular expression R of only formulas (replacing (φ, u) with $\varphi \wedge \mathrm{form}(u)$). Our problem is then just to decide inclusion between a regular expression of formulas and a regular expression of unit types. For this, we need to write them as regular expressions on a common alphabet, which will allow us to use a standard inclusion check.

Because we are interested in an inclusion check where the formula-based type is on the left, we take as alphabet the set of unit types appearing in the right-hand type plus a single symbol representing everything else.

DEFINITION 3. *Let ω be a constant which is not a unit type.*

Let τ be a regular expression of unit types. Let $U(\tau)$ be the set of unit types appearing at toplevel in τ. We assume that all unit types in $U(\tau)$ are mutually exclusive[5]*: $\forall (u_1, u_2) \in U(\tau)^2, [\![u_1]\!] \cap [\![u_2]\!] = \emptyset$. We define the alphabet of τ as follows: $\Sigma(\tau) = U(\tau) \cup \{\omega\}$.*

For a given τ, we extend the function $\mathrm{form}()$ to ω as follows: $\mathrm{form}(\omega) = \bigwedge_{u \in U(\tau)} \neg\mathrm{form}(u)$. This gives us a function from $\Sigma(\tau)$ to formulas such that the sets $\langle\!\langle \mathrm{form}(\alpha) \rangle\!\rangle$ form a partition of $\mathcal{F}$ when α ranges over $\Sigma(\tau)$.

Given a regular expression r on the alphabet $\Sigma(\tau)$, we define its interpretation in terms of set of values as $[\![r]\!]_\tau = [\![R]\!]$, where R is obtained by replacing all α in r by $\mathrm{form}(\alpha)$. Lemma 1 means that if r does not contain ω, this interpretation coincides with its interpretation as a type, i.e. we have $[\![r]\!]\uparrow = [\![r]\!]_\tau$.

In order to decide $R <: \tau$, we translate R into a regular expression on $\Sigma(\tau)$. There is no reason why the denotation of a formula in R should be included in a single unit type from $\Sigma(\tau)$; rather, it may have a nonempty intersection with several of them. Thus a formula will in general be translated by a choice expression.

[5] as required of types declared by the user: see Sec. 3.1.

$$\frac{(\!|u|\!)^{(\emptyset,\emptyset)}_{\top} = (\psi, (U, V))}{\mathrm{form}(u) = \mu(X = V(X))_{X \in \mathrm{dom}(V)} \text{ in } \psi} \qquad \frac{(\!|\tau_1|\!)^{\mathcal{V}}_{\varphi} = (\psi_1, \mathcal{V}_1) \qquad (\!|\tau_2|\!)^{\mathcal{V}_1}_{\varphi} = (\psi_2, \mathcal{V}_2)}{(\!|\tau_1 \mid \tau_2|\!)^{\mathcal{V}}_{\varphi} = (\psi_1 \vee \psi_2, \mathcal{V}_2)} \qquad \frac{X \text{ is fresh} \qquad (\!|\tau|\!)^{\mathcal{V}}_{\varphi \vee \langle \mathbf{2} \rangle X} = (\psi, \mathcal{V}')}{(\!|\tau*|\!)^{\mathcal{V}}_{\varphi} = (\mu X.\psi, \mathcal{V}')}$$

$$\frac{}{(\!|()|\!)^{\mathcal{V}}_{\varphi} = (\bot, \mathcal{V})} \qquad \frac{(\!|\tau_2|\!)^{\mathcal{V}}_{\varphi} = (\psi, \mathcal{V}') \qquad \varphi' = \begin{cases} \varphi \vee \langle \mathbf{2} \rangle \psi & \text{if } \mathit{nullable}(\tau_2) \\ \langle \mathbf{2} \rangle \psi & \text{otherwise} \end{cases} \qquad (\!|\tau_1|\!)^{\mathcal{V}'}_{\varphi'} = (\psi', \mathcal{V}'') \qquad \psi'' = \begin{cases} \psi \vee \psi' & \text{if } \mathit{nullable}(\tau_1) \\ \psi' & \text{otherwise} \end{cases}}{(\!|\tau_1, \tau_2|\!)^{\mathcal{V}}_{\varphi} = (\psi'', \mathcal{V}'')}$$

$$\frac{u \notin \mathrm{dom}(U) \qquad X \text{ is fresh} \qquad u = \texttt{element}\ n\ \{\tau\} \qquad (\!|\tau|\!)^{(U + \{u \mapsto X\}, V)}_{\neg \langle \mathbf{2} \rangle \top} = (\psi, (U', V')) \qquad \delta(\tau, \psi) = \begin{cases} \neg \langle \mathbf{1} \rangle \top \vee \langle \mathbf{1} \rangle \psi & \text{if } \mathit{nullable}(\tau) \\ \langle \mathbf{1} \rangle \psi & \text{otherwise} \end{cases}}{(\!|u|\!)^{(U,V)}_{\varphi} = (X \wedge \varphi, (U', V' + \{X \mapsto k(n) \wedge \delta(\tau, \psi)\}))} \qquad \frac{U(u) = X}{(\!|u|\!)^{(U,V)}_{\varphi} = (X \wedge \varphi, (U, V))}$$

Figure 9. Translation of a unit type into a formula

We use the following notation: if $r_1 \dots r_n$ are regular expressions, we write $\bigcup_{i \in \{1 \dots n\}} r_i$ for the regular expression $(r_1 \mid r_2 \mid \dots \mid r_n)$.

DEFINITION 4. *Let τ be a regular expression of unit types. For any formula φ, we define:*
$\Sigma_\tau(\varphi) = \{\alpha \in \Sigma(\tau) \mid \langle\!\langle \varphi \wedge \mathrm{form}(\alpha) \rangle\!\rangle \neq \emptyset\}$.

The regular expression on $\Sigma(\tau)$ corresponding to a formula φ is defined by $\mathrm{reg}_\tau(\varphi) = \bigcup_{\alpha \in \Sigma_\tau(\varphi)} \alpha$. This transformation is extended to regular expressions of formulas R in the obvious way.

The expression $\mathrm{reg}_\tau(R)$ is an over-approximation of R, however it is precise enough that we can replace R by $\mathrm{reg}_\tau(R)$ for the specific purpose of comparing it to τ:

LEMMA 2. *Let R and τ be a focused-tree type and a classical type respectively. Then $R <: \tau$ if and only if $[\![\mathrm{reg}_\tau(R)]\!]_\tau \subseteq [\![\tau]\!]_\tau$.*

Proof. For the right-to-left implication, we first notice that for any φ we have $\langle\!\langle \varphi \rangle\!\rangle \subseteq \bigcup_{\alpha \in \Sigma_\tau(\varphi)} \langle\!\langle \mathrm{form}(\alpha) \rangle\!\rangle$. Indeed, we have $\bigcup_{\alpha \in \Sigma(\tau)} \langle\!\langle \mathrm{form}(\alpha) \rangle\!\rangle = \mathcal{F}$ and $\Sigma_\tau(\varphi)$ is obtained from $\Sigma(\tau)$ by removing all α such that $\langle\!\langle \varphi \rangle\!\rangle \cap \langle\!\langle \mathrm{form}(\alpha) \rangle\!\rangle$ is empty: $\langle\!\langle \varphi \rangle\!\rangle$ must be included in the union of the remaining ones. By a straightforward induction, this yields $[\![R]\!] \subseteq [\![\mathrm{reg}_\tau(R)]\!]_\tau$, which allows us to conclude: $[\![\mathrm{reg}_\tau(R)]\!]_\tau \subseteq [\![\tau]\!]_\tau$ implies $[\![R]\!] \subseteq [\![\tau]\!]_\tau$.

For the left-to-right implication, we suppose $R <: \tau$, i.e. $[\![R]\!] \subseteq [\![\tau]\!]\uparrow$. As remarked above, this is equivalent to $[\![R]\!] \subseteq [\![\tau]\!]_\tau$ because τ does not contain ω. What we need to prove is that any word on $\Sigma(\tau)$ which matches $\mathrm{reg}_\tau(R)$ also matches τ (then $[\![\mathrm{reg}_\tau(R)]\!]_\tau \subseteq [\![\tau]\!]_\tau$ is immediate by definition of $[\![\cdot]\!]_\tau$). So let $\alpha_1 \dots \alpha_n$ be a word on $\Sigma(\tau)$ which matches $\mathrm{reg}_\tau(R)$. From the way $\mathrm{reg}_\tau(R)$ is constructed, we can deduce that there exists a word $\varphi_1 \dots \varphi_n$ of formulas which matches the regular expression R and is such that $\alpha_i \in \Sigma_\tau(\varphi_i)$ for all i. For all i, since $\alpha_i \in \Sigma_\tau(\varphi_i)$, there exists a focused tree $f_i \in \langle\!\langle \varphi_i \wedge \mathrm{form}(\alpha_i) \rangle\!\rangle$, by definition of $\Sigma_\tau(\varphi_i)$. We have $[f_1 \dots f_n] \in [\![R]\!]$, and since $[\![R]\!] \subseteq [\![\tau]\!]_\tau$, there must exist a word $\beta_1 \dots \beta_n$ on $\Sigma(\tau)$ which matches τ and verifies $f_i \in \langle\!\langle \mathrm{form}(\beta_i) \rangle\!\rangle$ for all i. But since the $\langle\!\langle \mathrm{form}(\alpha) \rangle\!\rangle$ are pairwise disjoint and we already have $f_i \in \langle\!\langle \mathrm{form}(\alpha_i) \rangle\!\rangle$, the only possibility is $\beta_i = \alpha_i$. Hence $\alpha_1 \dots \alpha_n$ matches τ. □

This result yields the following procedure to decide a relation $R <: \tau$:

1. Compute $\Sigma(\tau)$.
2. For all φ in R, compute $\Sigma_\tau(\varphi)$ using a logical solver.
3. Compute $\mathrm{reg}_\tau(R)$.
4. Test regexp inclusion between $\mathrm{reg}_\tau(R)$ and τ.

3.5 Complexity of the type system

Let $|R|$ be the number of formulas in R and $|\tau|$ the number of unit types in τ. In terms of complexity, the most expensive step is step 2; all the others are polynomial with respect to $|R| + |\tau|$ (due, in the case of step 4, to the fact that τ is 1-unambiguous; note that the size of $\mathrm{reg}_\tau(R)$ is at worst $|R| \times |\tau|$). Step 2 involves a polynomial $(|R| \times (|\tau| + 1))$ number of exponential-time satisfiability tests. The exponent is linear with respect to the size of the formula tested, which is at worst one φ in R plus all $\mathrm{form}(u)$ for u in τ. $\mathrm{form}(u)$ has the same size as the classical binary representation of the regular tree type u defined in [24]. Thus the cost is simple-exponential overall.

Notice that this is a worst-case complexity. Our approach is specifically designed to issue many calls to the exponential-time logical solver but with logical formulas of small size. Therefore, in practice the execution time is much less than if we had a single exponential-time test in terms of the whole problem instance size. This "divide and conquer" principle is further illustrated and quantified with concrete examples in Section 4.

3.6 Soundness of the type system

In this section, we prove the soundness of our type system. Classically, it relies on a subject-reduction lemma.

The type system we described up to here is for top-level expressions only. Runtime expressions can in addition be focused-tree or sequence literals. In order to state our lemma, we add the following four rules:

$$\text{T-ITEM}\ \frac{f \in [\![(\varphi, u)]\!]}{E; \Gamma \vdash f : (\varphi, u)} \qquad \text{T-VALSEQ}\ \frac{s \in [\![\rho]\!]}{E; \Gamma \vdash s : \rho} \qquad \text{T-SUB}\ \frac{E; \Gamma \vdash e : \rho \qquad [\![\rho]\!] \subseteq [\![\rho']\!]}{E; \Gamma \vdash e : \rho'}$$

$$\text{T-VALAXIS}\ \frac{f \in [\![(\varphi, u)]\!]}{E; \Gamma \vdash f/\mathit{axis}::n : \texttt{follow-axis}(\mathit{axis}, (\varphi, u), n)}$$

As opposed to the others, these rules do not give a way to infer the type from the environment and the expression, but they will be used only in the proofs since these expressions cannot appear in programs.

Our type system thus enriched enjoys the following properties.

LEMMA 3 (Substitution). *Let* e *be an expression containing a sequence variable $\$\overline{v}$, and suppose $E; \Gamma, \$\overline{v} : \rho_1 \vdash \mathrm{e} : \rho_2$. Let $s \in [\![\rho_1]\!]$. Then $E; \Gamma \vdash \mathrm{e}\,[^s/_{\$\overline{v}}] : \rho_2$.*

Similarly, let e *be an expression containing an item variable $\$v$, and suppose $E; \Gamma, \$v : (\varphi, u) \vdash \mathrm{e} : \rho$. Let $f \in [\![(\varphi, u)]\!]$. Then $E; \Gamma \vdash \mathrm{e}\,[^f/_{\$v}] : \rho$.*

Proof. In the case of the sequence variable, the typing derivation for e directly yields a typing derivation for $e\,[^s/_{\$\overline{v}}]$ by replacing all

occurrences of T-SEQVAR with T-VALSEQ. In the case of the item variable, we prove the result by induction on the typing derivation for e and distinguish cases depending on the last rule used. For most rules, the result is immediate from the induction hypothesis. The exceptions are T-ITEMVAR, which can be directly replaced with T-ITEM, T-AXIS, which can be directly replaced with T-VALAXIS, and T-IFAXIS. In this last case, ρ is of the form $\rho_1 \mid \rho_2$, e is of the form $\texttt{if non-empty}(\$v/axis::n)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, and we have:

$$E; \Gamma, \$v : (\varphi \wedge \texttt{has-}axis(k(n)), u) \vdash e_1 : \rho_1 \quad (1)$$

$$E; \Gamma, \$v : (\varphi \wedge \neg\texttt{has-}axis(k(n)), u) \vdash e_2 : \rho_2 \quad (2)$$

We distinguish two cases:

- if $f \in \langle\!\langle \texttt{has-}axis(k(n)) \rangle\!\rangle$, then $f \in [\![(\varphi \wedge \texttt{has-}axis(k(n)), u)]\!]$, since we already know $f \in [\![(\varphi, u)]\!]$. Thus, by induction hypothesis and (1) we have:

  $$E; \Gamma \vdash e_1 \left[{}^{f}/_{\$v}\right] : \rho_1 \quad (3)$$

  The formula $(\varphi \wedge \texttt{has-}axis(k(n))) \wedge \neg\texttt{has-}axis(k(n))$ is trivially unsatisfiable, therefore, according to the definitions, $\texttt{follow-axis}(axis, (\varphi \wedge \texttt{has-}axis(k(n)), u), n)$ is not nullable (see Fig. 8, last rule: the empty sequence is removed from the result).

  Therefore, from T-VALAXIS and (3) we can derive $E; \Gamma \vdash \texttt{if non-empty}(f/axis::n)\ \texttt{then}\ e_1 \left[{}^{f}/_{\$v}\right]\ \texttt{else}\ e_2 \left[{}^{f}/_{\$v}\right] : \rho_1$ using T-IFNONEMPTY. This last expression is actually $e \left[{}^{f}/_{\$v}\right]$, and since $[\![\rho_1]\!] \subseteq [\![\rho_1 \mid \rho_2]\!]$ we can conclude by T-SUB.

- if $f \notin \langle\!\langle \texttt{has-}axis(k(n)) \rangle\!\rangle$, then we have:
  $$f \in [\![(\varphi \wedge \neg\texttt{has-}axis(k(n)), u)]\!].$$
  Thus, by induction hypothesis and (2) we have:

  $$E; \Gamma \vdash e_2 \left[{}^{f}/_{\$v}\right] : \rho_2 \quad (4)$$

  We can check that $\texttt{navigate-}axis(\neg\texttt{has-}axis(k(n))) \wedge k(n)$ is always unsatisfiable. We will not detail it, but intuitively, such a formula says that, starting from the current node which satisfies $k(n)$ and following the path corresponding to *axis*, you will reach a node such that, by following the same path in reverse from there, you cannot reach a node satisfying $k(n)$. This is impossible since in our data model you can always go back to your starting point by following the same path in reverse.

  Therefore, the formula $\texttt{navigate-}axis(\neg\texttt{has-}axis(k(n)) \wedge \varphi \wedge \text{form}(u)) \wedge k(n)$ is also unsatisfiable (by an induction on the definition of navigate, we can see that it implies the previous formula). This implies that $[\![\texttt{follow-axis}(axis, (\varphi \wedge \neg\texttt{has-}axis(k(n)), u), n)]\!] \subseteq [\![()]\!]$. Thus, by T-VALAXIS and T-SUB we can derive $E; \Gamma \vdash f/axis::n : ()$. From there and (4), we can conclude using T-IFEMPTY and T-SUB.

LEMMA 4 (Subject reduction). *Let* e *be a runtime expression (as defined in Section 2.3), let E be a well-formed environment defining all references in* e*, and suppose we have $E; \emptyset \vdash \mathrm{e} : \rho$. Then either:*

- e *is a value s and $s \in [\![\rho]\!]$, or*
- *there exists* e' *such that* $\mathrm{e} \longrightarrow \mathrm{e}'$*, and* $E; \emptyset \vdash \mathrm{e}' : \rho$.

Proof. The proof is by induction on nested contexts, i.e., for expressions of the form $\mathcal{E}[e']$, we assume the lemma is true for e' in order to prove it for $\mathcal{E}[e']$.

If $e = \mathcal{E}[e']$ and e' is not a value, then we can see that in all cases (T-TREE, T-SEQ, T-FOR, T-IFANY, T-IFEMPTY, T-IFNONEMPTY[6]), the typing judgement for e has a typing judgement for e' as one of its premises. Then the induction hypothesis tells us that e' reduces (since it is not a value) to an expression e'' satisfying the same typing judgement, which cannot be ω since ω is not typable. Thus e reduces by R-CONTEXT to $\mathcal{E}[e'']$, and the typing rule which typed e also types $\mathcal{E}[e'']$.

We now treat all cases where either e is not of the form $\mathcal{E}[e']$ or e' is a value s.

- if $e = f$, the only possibility is that the judgement is the result of T-ITEM, thus $\rho = (\varphi, u)$ and $f \in [\![(\varphi, u)]\!]$. The expression reduces by R-SINGLETON to $[f]$.
- if $e = s$, the judgement must be the result of T-VALSEQ; the first branch of the alternative in the lemma is then immediate.
- if $e = \texttt{<}\sigma\texttt{>}\{s\}\texttt{</}\sigma\texttt{>} : x$, the judgement must be the result of T-TREE; thus $\rho = \text{form}(x)$, $x = \texttt{element}\ n\ \{\tau\}$ where n matches σ, and $E; \emptyset \vdash s : \rho$ with $\rho <: \tau$. By induction hypothesis, $s \in [\![\rho]\!]$. Since $\rho <: \tau$, we also have $s \in [\![\tau]\!]\uparrow$. Write $s = [(t_1, c_1) \dots (t_n, c_n)]$, and let $t = \sigma[[t_1 \dots t_n]]$. By definition of $[\![\tau]\!]\uparrow$, we have $[t_1 \dots t_n] \in [\![\tau]\!]$, hence $[t] \in [\![\texttt{element}\ n\ \{\tau\}]\!] = [\![x]\!]$. Therefore e reduces by R-TREE to $s' = [(t, \mathsf{Top})]$. Since $[t] \in [\![x]\!]$, we have $s' \in [\![x]\!]\uparrow$, thus $s' \in [\![\text{form}(x)]\!]$; therefore we can conclude $E; \emptyset \vdash s' : \text{form}(x)$ by T-VALSEQ.
- if $e = s_1, s_2$, the judgement must be the result of T-SEQ, so $\rho = \rho_1, \rho_2$ with $s_1 \in [\![\rho_1]\!]$ and $s_2 \in [\![\rho_2]\!]$. e reduces by R-CONCAT to a value s which is the concatenation of s_1 and s_2 and therefore is in $[\![\rho_1, \rho_2]\!]$. We conclude by T-VALSEQ.
- if $e = \texttt{for}\ \$v\ \texttt{in}\ s\ \texttt{return}\ e_2$, the judgement must be the result of T-FOR and we have $E; \emptyset \vdash \texttt{for}\ \$v : \rho_1\ \texttt{return}\ e_2 : \rho$ for some ρ_1 such that $s \in [\![\rho_1]\!]$. Then either:
  - $s = \epsilon$, in which case $\rho = ()$, e reduces to ϵ by R-FOREMPTY, and we can conclude by T-VALSEQ;
  - or $s = f :: s'$. We only give a proof sketch in this case. Since $s \in [\![\rho_1]\!]$, there exist (φ, u) and ρ_1' such that $f \in [\![(\varphi, u)]\!]$, $s' \in [\![\rho_1']\!]$, and $[\![(\varphi, u), \rho_1']\!] \subseteq [\![\rho_1]\!]$ (standard regular expression property). Now from the rules for for expressions we can deduce that there exist ρ_2' and ρ_2'' such that $[\![\rho_2', \rho_2'']\!] \subseteq [\![\rho]\!]$ and :

    $$E; \$v : (\varphi, u) \vdash e_2 : \rho_2' \quad (1)$$

    $$\text{and } E; \emptyset \vdash \texttt{for}\ \$v : \rho_1'\ \texttt{return}\ e_2 : \rho_2''. \quad (2)$$

    e reduces to: $e' = e_2 \left[{}^{f}/_{\$v}\right], \texttt{for}\ \$v\ \texttt{in}\ s'\ \texttt{return}\ e_2$ (by R-FOR). From the substitution lemma and (1) we deduce:

    $$E; \emptyset \vdash e_2 \left[{}^{f}/_{\$v}\right] : \rho_2' \quad (3)$$

    and we conclude:

    $$\dfrac{\dfrac{\dfrac{\dfrac{s' \in [\![\rho_1']\!]}{E; \emptyset \vdash s' : \rho_1'}\ \text{(T-ValSeq)} \qquad (2)}{E; \emptyset \vdash \texttt{for}\ \$v\ \texttt{in}\ s'\ \texttt{return}\ e_2 : \rho_2''}\ \text{(T-For)} \qquad (3)}{E; \emptyset \vdash e' : \rho_2', \rho_2''}\ \text{(T-Seq)}}{E; \emptyset \vdash e' : \rho}\ \text{(T-Sub)}$$

- if $e = \texttt{if non-empty}(s)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, the judgement must be the result of T-IFANY, T-IFEMPTY or T-IFNONEMPTY. In the case of T-IFANY, the expression reduces to either e_1 and e_2, and both ρ_1 and ρ_2 are such that $[\![\rho_i]\!] \subseteq [\![\rho_1 \mid \rho_2]\!]$, so we can conclude by T-SUB. In the case of T-IFEMPTY and T-IFNONEMPTY, the result is straightforward.
- if $e = f/axis::n$, the judgement must be the result of T-VALAXIS and T-AXIS. The result is mostly straightforward by a case analysis on the reduction rule which applies; as there are 16 cases, we leave this as an exercise for the reader.

[6] It cannot be T-IFAXIS because that would require $\$v/axis::n$ to be typable with an empty Γ.

- $e = \$v$, $e = \$v/axis::n$ or $e = \$\overline{v}$ are impossible since they can only be typed with a non-empty Γ.

We can now prove our main soundness theorem.

THEOREM 1 (Soundness). *Let* e *be an expression and let* E *be a well-formed environment comprising definitions for all type references* x *appearing in* e.

If there exists ρ *such that* $E;\emptyset \vdash \mathrm{e} : \rho$, *then there is a finite reduction sequence* $\mathrm{e}\left[{}^{s_1\cdots s_n}/_{\$\overline{v}_1\ldots\$\overline{v}_n}\right] \longrightarrow \cdots \longrightarrow s$ *and we have* $s \in [\![\rho]\!]$.

Proof. The fact that the derivations are finite is actually independent of typing: it is a property of the fragment of XQuery we consider. We can see that rule R-FOR always consumes one element of the sequence, which is finite, and that all navigation steps in a given direction (either 'backward', involving only $\langle\overline{2}\rangle$ and $\langle\overline{1}\rangle$, of 'forward', involving only $\langle 1\rangle$ and $\langle 2\rangle$) reduce to navigation steps which go in the same direction from a node that is strictly further in that direction. It has to end eventually because focused trees are finite and acyclic. The other cases either yield a value or yield an expression smaller than the initial one.

Then the fact that the final result of the reduction sequence matches the expected type is just a straightforward consequence of the substitution lemma (for the initial parameter replacement) and of subject reduction. □

3.7 Extensions to the core

The XQuery navigational core we defined our type system on is quite small; we can ask how it would scale to a larger fragment of the language. In particular, it would be interesting to add function declaration and application, which raises the question of type annotations. Function declarations in XQuery are, indeed, type-annotated, but the annotations are classical XQuery types. These can be straightforwardly added to our system; however, to fully benefit from our improvements in precision, it would be more interesting to specify formula-enriched input and output types for the functions; this requires a user-friendly syntax in which to write the annotations, such as Schematron [27] and RelaxNG [13], which our type language represents an ideal compilation target for.

4. Experimental Results

To evaluate the gain in typing precision and the feasibility of our approach in practice, we implemented a type-checker prototype. Our implementation, written in Scala, relies on two external third-party implementations to which it delegates computations when performing subtype checks:

1. the Haskell implementation of the syntax-directed algorithm for inclusion of (word) regular expressions described in [25];
2. and the Java implementation of the logical satisfiability-testing solver described in [22].

To illustrate the gain in typing precision, we compared our prototype to implementations of existing techniques. Among the numerous XQuery implementations available [37], only very few actually implement XQuery static typing features. We retained the following:

1. Galax [35]: an open-source implementation, by two authors of the XQuery recommendation;
2. XQuantum [14]: a commercial implementation, freely available for one month.

We extensively tested our prototype against those implementations, and we report below on a few examples. These examples are kept very simple for the sake of brevity, and for giving an intuition on how frequent are code patterns for which we can expect a gain in precision using our approach.

Our prototype takes four parameters as input: an XQuery expression (such as the one of Listing 1), input and output types, given under the form of e.g. DTDs, and the name of some element to be considered as root in the input type.

```
1 let $v := /self::* return
2   <body>{if ($v/descendant::table)
3          then <div>Input contains a table.</div>
4          else for $i in $v/body return
5               for $j in $i/* return $j}</body>
```

Listing 1. Sample XQuery with Conditional Statement.

The simple XQuery expression shown in Listing 1 is meant to be applied to some input web page, valid with respect to some type such as the one illustrated on Listing 2 (intentionally simplified here for presentation purposes). One might check that any tree generated by the code snippet of Listing 1 is indeed valid with respect to an output type such as the one of Listing 3 that defines an even simpler content model for the `body` element. This is because the expression of Listing 1 either generates a `div` element or copies the contents of the `body` in the absence of `table` elements.

```
1 <!ELEMENT html (head?,body)>
2 <!ELEMENT body ((div | table)+)>
```

Listing 2. An Excerpt from Input Type (DTD notation).

```
1 <!ELEMENT body ((div)+)>
```

Listing 3. An Excerpt from Output Type (DTD notation).

Such a static type check fails with the XQuantum and Galax implementations that both report false alarms[7]. Our prototype succeeds in type-checking the conditional statement of Listing 1, notably because the negation of the condition is propagated for typing the "else" clause. This kind of propagation is typically made possible by the use of recursive logical formulas in our type language. Our prototype type-checked this example in a total time of 158 ms, including 2 external calls for checking regular expression inclusion, and 6 calls to the logical solver for a total solver time of 82 ms.

Our approach brings increased typing precision in any situation where the XQuery expression uses a backward axis (e.g. `parent`, `ancestor`, `preceding`, etc.) or an horizontal axis (e.g. a navigation to any preceding or following sibling). In practice, a key-value store constitutes a very common situation in which horizontal navigation is essential for accessing the value of a given key (or reciprocally a key from a given value). For example, the code of Listing 4 is intended to be used with documents valid with respect to the official Apple DTD that defines 11 elements[8] for representing nested property lists in general, such as ITunes audio libraries in particular. For a given music library, the code generates a list of referenced files with the corresponding track number. This list is expected to be valid with respect to the same DTD.

The XQuantum implementation does not parse the code of Listing 4 because it does not support horizontal/backward axes. The Galax implementation parses the code but is not capable of inferring any precise type information for those axes. Interestingly, the static type-checking of Listing 4 by Galax fails with the

[7] XQuantum fails with:
"`type error: cannot promote element(div, text()) | (element(div, xs:string) | element(table, Tr+))+ to element(div, xs:string)+`" and Galax fails for a very similar example with: "`Expecting type: element div* but expression has type: (element div | element table)*`".

[8] http://www.apple.com/DTDs/PropertyList-1.0.dtd

error "`element dict but expression has type: element dict of type xs:untyped`" unless we surround the constructed element "`dict`" (line 2) with a `validate`$\{\cdot\}$ function call. However, in that case only a dynamic type check (validation) is performed at runtime, but no static type check is done[9]. In contrast, the purpose of our tool is to perform this check at compile-time (once for all) so that validation of the output can be avoided at runtime. Our tool succeeds in static type-checking the code of Listing 4 in a total time of 465 ms, including 2 calls for regular expression inclusion and 380 ms spent in a total of 27 calls to the logical solver.

```
1 let $r := /self::* return
2   <dict>{
3     for $i in $r/descendant::dict return
4       for $j in $i/key[text()="Location"] return
5         let $v:=$j/following-sibling::*[1] return
6           let $p := $j/parent::* return
7             ($p/preceding-sibling::*[1], $v)
8   }</dict>
```

Listing 4. Sample XQuery with Sibling Navigation.

The pattern "`following-sibling::*[1]`", widely found in practice, simply corresponds to "$\langle\bar{2}\rangle\top$" in logic.

The purpose of our prototype implementation is also to give insights on practical costs with current commodity hardware. All reported evaluations were performed on an Intel Core i7 with 16GB of RAM running OS X 10.9.4. Consider the example of Listing 5 for which our approach also provides a gain in typing precision, and whose size is quite representative of a real XQuery function body size.

```
1  let $parts :=
2    for $i in /* return
3      if ($i/descendant::part) then
4        for $part in $i/descendant::part return
5          <part>{$part/title , $i/descendant::author ,
6                 $part/chapter}</part>
7      else
8        <part>{for $j in $i/head return $j/*,
9               for $j in $i/body return $j/chapter}</part>
10 return
11   let $bookcontents :=
12     if (count($parts)=1) then
13       (for $i in $parts return $i/chapter)
14     else $parts
15   return <book>{<title/>,
16                 (for $i in $parts return $i/author),
17                 $bookcontents}</book>
```

Listing 5. Sample XQuery Function Body.

Our prototype succeeds in analysing the code snippet of Listing 5 in the presence of input and output types (with 12 and 10 different element names, respectively) in a total time of 1970 ms, including 13 checks for regular expression inclusion and 1769 ms spent in a total of 42 solver calls.

Notice that the total time spent in the satisfiability-testing solver calls accounts for 90% of the total type-checking time. Our prototype type-checks XQuery expressions of similar or slightly larger size, in the presence of types of small to moderate size in a few seconds (same order of magnitude). The important observation is that the proportion of the time spent in solver calls stays roughly in the 88% to 96% range of the total analysis time. This confirms in practical terms what we know from theory: the dominant cost in the analysis is the time spent in the logical solver.

[9] This can easily be observed by e.g. generating an element which is not a member of the output type.

5. Related work

Static typing for XQuery has been standardized by the W3C [17]. The type-system proposed by the W3C has been inspired by the seminal work from Hosoya and Pierce [23], which is itself based on finite tree automata containment [24]. The current W3C type-system [17] has a polynomial-time complexity (except for nested let clauses, as noticed by Colazzo and Sartiani [15]). A more precise typing of `for` loops than what made it into the standard had been studied by Fernández et al. [19], at a time when XQuery was not yet a standard but still a proposal in early form [18]. Colazzo et al. [16] also separately introduced a very similar type system. Both type systems are more precise than the one of the W3C while having an exponential-time complexity. Colazzo and Sartiani [15] provide an analysis that illustrates in which cases these type systems differ in terms of precision and complexity. None of these type systems supports non-downward navigation in XML trees, despite it being an essential part of the XQuery standard, since the very beginning. This issue, that our proposal addresses, was clearly reported 14 years ago by Fankhauser et al. [18]. Nevertheless, to the best of our knowledge, this problem has only been indirectly considered so far, as we review below, with the exception of Castagna et al. [10].

Benedikt and Cheney [1] introduce a type system for the W3C's XQuery Update Facility language [11], in which the existence of a sound type-checker for XPath backward axes is assumed. In follow-up works, backward axes are either absent (as in e.g. Cheney and Urban [12]) or they are dealt with using earlier work on XPath static analysis [21] (as in e.g. Benedikt and Cheney [2]). The work by Genevès et al. [21] provides an algorithm to decide query containment for a fragment of XPath with backward axes. The query containment problem consists in statically checking whether the set of nodes returned by one XPath query are always contained in the set returned by another query, for any tree. This work also uses the logic language for XML trees which we described in Sec. 3.2.1 and the associated satisfiability solver. However, this paper is limited to XPath and does not consider XQuery. In comparison, we consider a core fragment of XQuery which supports not only XPath but also control flow operators, and, most importantly, the element construction. The element construction, unlike what it might seem at first sight, is far from trivial, as we discuss in Section 3.3. Furthermore, a fundamental difference with Genevès et al. [21] is that the values we consider are sequences of nodes (instead of sets of nodes). Our sequences of nodes may come from different trees. Nodes have a position in the sequence (used for element construction) and also retain their original tree context independently (used for navigation). None of these aspects, which are essential for XQuery, were considered or discussed in this previous work. Finally, our type system is based on regular expressions of formulas, rather than single formulas. The present proposal is a novel approach which allows us to deal with both the sequence and context aspects, which, to the best of our knowledge, no other system does.

Calcagno et al. [8] introduced context logic, a generalisation of separation logic, for reasoning about both (unordered XML) data and contexts. This reasoning is extended by Gardner et al. [20] to ordered XML and while-programs over atomic tree updates, modeled after the DOM tree update library [36]. These works do not consider high-level language constructs similar to the ones found in XQuery. They seem however promising for reasoning about low-level DOM updates, e.g. in JavaScript programs. The non-trivial connection between context logic and modal logic is explored by Calcagno et al. [9].

The XML type-checking problem has also been studied for other domain-specific languages such as CDuce [4], XSLT [28] or with specific transformers like transducers [30–32]. For a recent

survey on type-checking for XML, see Benzaken et al. [5] 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') [29], recently extended into XQ_{H} by Benedikt and Vu [3] who added higher-order functions, and $\mu\mathrm{XQ}$ ('micro XQuery') [16], extended into $\mu\mathrm{XQ}^{+}$ ('mini XQuery') by Colazzo and Sartiani [15]. The papers defining XQ and XQ_{H} focus on the semantics of the language and the complexity of query evaluation. The papers defining $\mu\mathrm{XQ}$ and $\mu\mathrm{XQ}^{+}$ focus on typing and correctness. 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 XQuery fragment is basically $\mathrm{XQ}/\mu\mathrm{XQ}^{+}$ with the upward and sideways axes added.

Very recently, Castagna et al. [10] have defined a larger fragment with backward axes, that they translate into an extension of the ℂDuce language, which they study. However, neither the precision nor the computational complexity obtained for typing XQuery are studied. No implementation is reported.

6. Conclusion

The work presented in this paper is a type-checking system for XQuery, that takes *all* navigation expressions properly into account. This solves an open issue reported 14 years ago [18], and improves the type-system that finally made it into the standard [17].

Our contribution is fourfold. 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 showed how to combine it with the existing one. Fourth, we proposed a sound type-system which offers a net increase in precision.

References

[1] M. Benedikt and J. Cheney. Semantics, types and effects for XML updates. In *DBPL'09*, pages 1–17, 2009. doi: 10.1007/978-3-642-03793-1_1.

[2] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. *Proc. VLDB Endow.*, 3(1-2):906–917, Sept. 2010. doi: 10.14778/1920841.1920956.

[3] M. Benedikt and H. Vu. Higher-order functions and structured datatypes. In *WebDB'12*, pages 43–48, 2012. URL http://db.disi.unitn.eu/pages/WebDB2012/papers/p13.pdf.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP'03*, pages 51–63, 2003. doi: 10.1145/944705.944711.

[5] V. Benzaken, G. Castagna, H. Hosoya, B. C. Pierce, and S. Vansummeren. XML typechecking. In *Encyclopedia of Database Systems*, pages 3646–3650. Springer, 2009.

[6] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language (2nd ed.). W3C Recommendation, 2010. http://www.w3.org/TR/xquery/.

[7] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229 – 253, 1998. doi: 10.1006/inco.1997.2688.

[8] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL'05*, pages 271–282, 2005. doi: 10.1145/1040305.1040328.

[9] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: Completeness and parametric inexpressivity. In *POPL'07*, pages 123–134, 2007. doi: 10.1145/1190216.1190236.

[10] G. Castagna, H. Im, K. Nguyễn, and V. Benzaken. A core calculus for XQuery 3.0. In *ESOP'15*, pages 232–256, 2015. doi: 10.1007/978-3-662-46669-8_10.

[11] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update Facility. W3C WD, http://www.w3.org/TR/xqupdate/, 2006.

[12] J. Cheney and C. Urban. Mechanizing the metatheory of mini-XQuery. In *CPP'11*, pages 280–295, 2011. doi: 10.1007/978-3-642-25379-9_21.

[13] J. Clark and M. Murata. RELAX NG home page. http://relaxng.org/, 2014.

[14] I. Cognetic Systems. XQuantum XML database server, 2014. http://www.cogneticsystems.com/xquery/xquery.html.

[15] D. Colazzo and C. Sartiani. Precision and complexity of XQuery type inference. In *PPDP'11*, pages 89–100, 2011. doi: 10.1145/2003476.2003490.

[16] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of XML queries. In *ICFP'04*, pages 126–137, 2004. doi: 10.1145/1016850.1016869.

[17] D. Draper, M. Dyck, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C recommendation, December 2010. http://www.w3.org/TR/xquery-semantics/.

[18] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML query algebra, W3C working draft, December 2000. http://www.w3.org/TR/2000/WD-query-algebra-20001204/.

[19] M. F. Fernández, J. Siméon, and P. Wadler. A semi-monad for semi-structured data. In *ICDT'01*, pages 263–300, 2001. doi: 10.1007/3-540-44503-X_18.

[20] P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local Hoare reasoning about DOM. In *PODS'08*, 2008. doi: 10.1145/1376916.1376953.

[21] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI'07*, pages 342–351, 2007. doi: 10.1145/1250734.1250773.

[22] P. Genevès, N. Layaïda, A. Schmitt, and N. Gesbert. Efficiently deciding μ-calculus with converse over finite trees. *ACM Transactions on Computational Logic*, 16(2):16:1–16:41, 2015. doi: 10.1145/2724712.

[23] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003. doi: 10.1145/767193.767195.

[24] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005. doi: 10.1145/1053468.1053470.

[25] D. Hovland. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 78(6):1795 – 1813, 2012. doi: 10.1016/j.jcss.2011.12.003.

[26] G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[27] ISO/IEC. Document schema definition language – schematron. http://www.schematron.com, 2012.

[28] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[29] C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31 (4):1215–1256, Dec. 2006. doi: 10.1145/1189769.1189771.

[30] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *PODS'05*, pages 283–294, 2005. doi: 10.1145/1065167.1065203.

[31] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT'07*, pages 254–268, 2007. doi: 10.1007/11965893_18.

[32] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 66(1):66–97, 2003.

[33] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. XQuery update facility 1.0, W3C recommendation, March 2011. http://www.w3.org/TR/xquery-update-10/.

[34] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML Query Language. W3C Last Call Working, July 2013. http://www.w3.org/TR/xquery-30/.

[35] J. Siméon, M. Fernández, et al. Implementation of xquery 1.0, 2008. http://galax.sourceforge.net/.

[36] W3C. Document object model (DOM), W3C recommendation, 2004. http://www.w3.org/DOM/.

[37] W3C. Xml query (xquery) implementations, 2014. http://www.w3.org/XML/Query/#implementations.

[38] P. Wadler. XQuery: A typed functional language for querying XML. In *Advanced Functional Programming*, pages 188–212, 2002.