Backward Type Inference for XML Queries

Hyeonseung Im, Pierre Genevès, Nils Gesbert, Nabil Layaïda

Backward Type Inference for XML Queries

Hyeonseung Im[1], Pierre Genevès[2], Nils Gesbert[2], Nabil Layaïda[2*]

[1]Kangwon National University
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble France

March 29, 2017

Abstract

Although XQuery is a statically typed, functional query language for XML data, some of its features such as upward and horizontal XPath axes are typed imprecisely. The main reason is that while the XQuery data model allows us to navigate upwards and between siblings from a given XML node, the type model, *e.g.,* regular tree types, can describe only the subtree structure of the given node. Recently, Castagna et al. (2015) and Genevès and Gesbert (2015) independently propose a precise forward type inference system for XQuery using an extended type language that can describe not only a given XML node but also its context. In this paper, as a complementary method to forward type inference systems, we propose a novel backward type inference system for XQuery, using the type language proposed by Genevès and Gesbert (2015). Our backward type inference system provides an exact typing result for XPath axes and a sound typing result for XQuery expressions.

Introduction

XQuery Boag et al. (2010) is a statically typed, functional, World Wide Web Consortium (W3C) standard query language for XML data. Its type language is based on regular tree types Hosoya et al. (2005) and its static and dynamic semantics are formally defined Draper et al. (2010). One of the key features of XQuery is its use of XPath Clark and DeRose (1999); Berglund et al. (2010) to navigate and extract XML data. Although XPath navigational expressions greatly facilitate XML manipulation, they are also a main source of undesired, imprecise type inference in the XQuery formal semantics. Specifically, when upward or horizontal XPath axes such as `parent` and `following-sibling` are used, the formal semantics simply deduces the most general type (*e.g.,* AnyElt for `parent` and AnyElt$*$ for `following-sibling` where AnyElt denotes the type of all XML elements), which essentially conveys no information, regardless of the type of the initial XML node. In the end, in the recent recommendations of XPath 3.0 Robie et al. (2014a) and XQuery 3.0 Robie et al. (2014b), static typing became "implementation defined" and hence optional.

The over-approximation in type inference is in particular due to the discrepancy between the XQuery data model and the type model. Specifically, in XQuery, values are *sequences of pointers* to XML tree nodes and each pointer can point anywhere in the corresponding tree. Moreover, given such a pointer, it is always possible to obtain a pointer to its parent or sibling node, thus allowing us to navigate upwards and between siblings. In clear contrast, given a pointer value, its type (*e.g.,* a regular tree type) can describe only the subtree structure to which the pointer points, but not its context, *i.e.,* part of the whole tree except the subtree pointed by the pointer value. Therefore, with this type language, only downward axes such as `child` and `desc` can be precisely typed at best (*e.g.,* Colazzo and Sartiani (2011)).

To alleviate this limitation, recently, Castagna et al. (2015) and Genevès and Gesbert (2015) independently developed a more precise type system for XQuery using an extended type language and in the style of forward type inference, where the input type is given and an output type is inferred. Specifically, they extend regular tree types with zipper types, inspired by Huet (1997), and modal logic formulas Genevès et al. (2007), respectively, both of which essentially allow us to describe not only an XML node but also its context. Hence their type systems deduce a more precise type using the context information for both upward and horizontal axes.

In this paper, as a complementary method to forward type inference such as in Castagna et al. (2015); Genevès and Gesbert (2015), we propose a novel backward type inference system for XQuery. Given an XQuery expression

*This research was partially supported by ANR Project CLEAR (ANR-16-CE25-0010).

e and an expected output type ρ_o, backward inference computes the pre-image ρ_i of ρ_o under e such that it is guaranteed that for any XML document of type ρ_i, e always produces a document of type ρ_o. Since in many cases XQuery is used to extract a small fragment of a large XML document (unless it is used to publish a webpage from a given XML document) and it is much easier for the user to specify a smaller type (*i.e.,* the output type), backward inference is more effective than forward inference in these cases. Furthermore, whereas in forward inference output types may not be regular and thus need to be approximated, in backward inference exact typing may be done, *e.g.,* Maneth et al. (2005), although practical implementation may not be feasible.

To develop a backward type inference system, we first define the syntax and semantics of an XQuery core by representing XML nodes as *focused trees* Genevès et al. (2007) (Section 1), following the work by Genevès and Gesbert (2015). A focused tree is a variant of the Zipper data structure Huet (1997), which describes a whole tree "seen" from a given internal node, that is, a subtree and its context. As focused trees support functional navigation in any direction from a given tree node, we can simplify the semantics of the XQuery core, without resorting to an external store for node pointers as in the XQuery formal semantics. With focused trees, our semantics is a straightforward extension of the one given in Colazzo and Sartiani (2011) with non-downward XPath axes.

As for our type language, we use formula-enriched sequence types Genevès and Gesbert (2015), which combine the usual regular tree types with tree logic formulas Genevès et al. (2007) to describe both a tree node and its context (Section 2). Then, using formula-enriched sequence types, we define an exact backward type inference for XPath axes (Section 3). That is, given an XPath axis and an output type ρ, if our inference system infers an input type ρ', the result of evaluating the axis is of type ρ if and only if an input focused tree is of type ρ'. Then, building on the inference rules for XPath axes, we define a sound backward inference system for the XQuery core (Section 4). While our inference system is practically implementable, exact typing is not possible in the presence of arbitrary `for`-expressions with a formula-enriched sequence type as an output type, and therefore we introduce an approximation.

We summarize the main contributions as follows:

- We formulate a novel backward type inference system for a large fragment of XQuery, including all the XPath axis expressions. In particular, we show that our backward inference system for XPath axes is exact.
- We prove soundness of our backward type inference system for the XQuery core, from which we can obtain a practical typechecking algorithm. We also formally analyze the complexity of our inference system, and show that it is double exponential in terms of the given expression.

1 Syntax and Semantics of an XQuery Core

In this section, we introduce an XQuery Core, a minimal XQuery fragment supporting all the navigational XPath axis expressions. Our XQuery core is basically an extension of miniXQuery proposed by Colazzo and Sartiani (2011) with non-downward axes.

1.1 Focused Trees

We first define XML values as a set of focused trees, inspired by Huet's zipper data structure Huet (1997). A focused tree is an XML node with its context: the siblings and the parent of the node, including the parent's context recursively. Intuitively a context records the path covered when traversing an XML tree from its root to a certain node. Thus focused trees allow us to easily navigate XML trees in any direction: both forward and backward navigation.

Below we formally define the syntax of our data model. We assume an alphabet Σ of labels, ranged over by σ.

$$
\begin{array}{lrcl}
\text{Trees} & t & ::= & \sigma[tl] \\
\text{Tree lists} & tl & ::= & \epsilon \mid t :: tl \\
\text{Contexts} & c & ::= & \mathsf{Top} \mid (tl; c[\sigma]; tl) \\
\text{Focused trees} & f & ::= & (t, c)
\end{array}
$$

A focused tree (t, c) is a pair consisting of a focused node (or a tree) t and its context c. A context c is Top if the focused node is at the root. Otherwise it is a triple $(tl_l; c[\sigma]; tl_r)$: tl_l is a list of the left siblings of the current focused node in reverse order (the first element of the list is the tree immediately to the left of the current node), $c[\sigma]$ the context above the current node where σ is the label of the parent, and tl_r a list of the right siblings.

Expressions	e	::=	$\epsilon \mid \texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u \mid e, e \mid \texttt{for}\ \$v\ \texttt{in}\ e\ \texttt{return}\ e$
		$\mid$	$\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e\ \texttt{return}\ e \mid \texttt{if}\ \texttt{nempty}(e)\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid$	$\$v/axis\texttt{::}n \mid \$var$
Variables	$\$var$	::=	$\$v \mid \$\overline{v}$
Axis names	$axis$	::=	$\texttt{self} \mid \texttt{child} \mid \texttt{desc} \mid \texttt{nsibl} \mid \texttt{parent} \mid \texttt{anc} \mid \texttt{psibl}$
Name tests	n	::=	$\sigma \mid *$
Values	s	::=	$\epsilon \mid f :: s$

Figure 1: Syntax of XQuery core

We now describe how to navigate a focused tree in a binary fashion. Given a focused tree f, forward navigation $f\langle 1\rangle$ and $f\langle 2\rangle$ respectively change the focus to the leftmost child and to the next right sibling of the current focused node. Conversely backward navigation $f\langle \bar{1}\rangle$ and $f\langle \bar{2}\rangle$ respectively change the focus to the parent and the preceding left sibling of the current node. In particular, $f\langle \bar{1}\rangle$ is defined only if the current node is the leftmost node, *i.e.,* it has no left sibling. Definition 1.1 formally defines the navigation of focused trees.

Definition 1.1 (Navigation of focused trees)**.**

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

If the focused tree does not have the required shape, these operations are undefined.

1.2 XQuery Core

Figure 1 defines the abstract syntax of a simplified navigational fragment of XQuery core, defined in the XQuery 1.0 and XPath 2.0 Formal Semantics Draper et al. (2010). In the XQuery core, navigational (*i.e., structural*) expressions are well separated from data value expressions (*e.g.,* ordering and node identity testing) which make typechecking undecidable (see for instance Alon et al. (2003)). Since the full language of XQuery can be compiled into the XQuery core and we are mainly interested in typechecking, we consider only navigational expressions in this paper.

Basically, an XQuery expression e is inductively defined as either an empty sequence ϵ, an XML element constructor ($\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u$), a sequence concatenation (e_1, e_2), a for-loop expression $\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2$, a let-binding $\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2$, a conditional expression $\texttt{if}\ \texttt{nempty}(e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, a navigational axis expression $\$v/axis\texttt{::}n$, or a variable $\$var$.

First of all, we assume that an XML element constructor $\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>}$ is always annotated with a type u (the precise definition of u is given in Section 2.1). In XQuery, the result of a construction expression of the form $\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>}$ is considered untyped (both statically and dynamically) unless it is validated, *e.g.,* using an `validate` expression. The `validate` expression checks if the constructed XML element conforms to the expected type at runtime, and if not, it raises a dynamic type error. Our element constructor is thus a combination of XQuery's untyped element constructor and `validate` expression. For untyped element constructors in XQuery, *i.e.,* without `validate`, we assume that they are annotated with AnyElt which is the type of all XML elements.

As for other expressions, in a for-loop expression, an item variable $\$v$ is bound to a single element node (or a single "item" in the XQuery terminology), whereas in a let-binding expression, a sequence variable $\$\overline{v}$ is bound to a possibly empty sequence of nodes. In a conditional expression $\texttt{if}\ \texttt{nempty}(e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, if the condition e evaluates to a non-empty sequence of nodes, then e_1 is evaluated; otherwise, e_2 is evaluated. An axis expression $\$v/axis\texttt{::}n$ extracts the nodes reachable from the current node $\$v$ through *axis* and that also satisfy name test n. Path navigation can start only from an item variable. A name test n is either a node label σ or a wildcard pattern $*$ that matches any label. For path navigation, we consider only `self`, `child`, `desc`, `nsibl`, `parent`, `anc`, and `psibl` axes[1] because other axes can easily be encoded. We use the following syntactic sugar:

$$\$v/\texttt{desc-or-self::}n \quad \equiv \quad \$v/\texttt{self::}n,\ \$v/\texttt{desc::}n$$

[1]We use abbreviated names instead of the full name of the XPath axes. In particular, `nsibl` corresponds to `following-sibling`.

$$
\begin{array}{rcll}
[\![\epsilon]\!]_\eta & = & \epsilon & \\
[\![\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u]\!]_\eta & = & f & \text{if } \begin{cases} [\![e]\!]_\eta = [(t_1, c_1), \ldots, (t_n, c_n)] \\ f = (\sigma[t_1 :: \ldots :: t_n :: \epsilon], \mathsf{Top}) \\ f \in [\![u]\!] \end{cases} \\
[\![e_1, e_2]\!]_\eta & = & [\![e_1]\!]_\eta, [\![e_2]\!]_\eta & \\
[\![\$var]\!]_\eta & = & \eta(\$var) & \\
[\![\texttt{for } \$v \texttt{ in } e_1 \texttt{ return } e_2]\!]_\eta & = & \Pi_{f_1,\ldots,f_n} [\![e_2]\!]_{\eta, \$v \mapsto f_i} & \text{if } [\![e_1]\!]_\eta = [f_1, \ldots, f_n] \\
[\![\texttt{for } \$v \texttt{ in } e_1 \texttt{ return } e_2]\!]_\eta & = & \epsilon & \text{if } [\![e_1]\!]_\eta = \epsilon \\
[\![\texttt{let } \$\bar{v} \texttt{ := } e_1 \texttt{ return } e_2]\!]_\eta & = & [\![e_2]\!]_{\eta, \$\bar{v} \mapsto [\![e_1]\!]_\eta} & \\
[\![\texttt{if nempty}(e) \texttt{ then } e_1 \texttt{ else } e_2]\!]_\eta & = & [\![e_1]\!]_\eta & \text{if } [\![e]\!]_\eta = f, s \\
[\![\texttt{if nempty}(e) \texttt{ then } e_1 \texttt{ else } e_2]\!]_\eta & = & [\![e_2]\!]_\eta & \text{if } [\![e]\!]_\eta = \epsilon \\
[\![\$v/\mathit{axis}::n]\!]_\eta & = & [\![\eta(\$v)/\mathit{axis}::n]\!] &
\end{array}
$$

$$
\begin{array}{rcll}
[\![f/\texttt{self::}n]\!] & = & [f] & \text{if } \texttt{name}(f) \leq n \\
[\![f/\texttt{self::}n]\!] & = & \epsilon & \text{if } \texttt{name}(f) \not\leq n \\
[\![f/\texttt{child::}n]\!] & = & [\![f'/\texttt{self::}n]\!], [\![f'/\texttt{nsibl::}n]\!] & \text{if } f' = f\,\langle 1 \rangle \\
[\![f/\texttt{child::}n]\!] & = & \epsilon & \text{if } f = (\sigma[\epsilon], c) \\
[\![f/\texttt{parent::}n]\!] & = & [\![f'/\texttt{self::}n]\!] & \text{if } f' = f\,\langle \bar{1} \rangle \\
[\![f/\texttt{parent::}n]\!] & = & [\![f'/\texttt{parent::}n]\!] & \text{if } f' = f\,\langle \bar{2} \rangle \\
[\![f/\texttt{parent::}n]\!] & = & \epsilon & \text{if } f = (t, \mathsf{Top}) \\
[\![f/\texttt{nsibl::}n]\!] & = & [\![f'/\texttt{self::}n]\!], [\![f'/\texttt{nsibl::}n]\!] & \text{if } f' = f\,\langle 2 \rangle \\
[\![f/\texttt{nsibl::}n]\!] & = & \epsilon & \text{if } f = (t, (tl; \sigma[c]; \epsilon)) \\
[\![f/\texttt{psibl::}n]\!] & = & [\![f'/\texttt{psibl::}n]\!], [\![f'/\texttt{self::}n]\!] & \text{if } f' = f\,\langle \bar{2} \rangle \\
[\![f/\texttt{psibl::}n]\!] & = & \epsilon & \text{if } f = (t, (\epsilon; \sigma[c]; tl)) \\
[\![f/\texttt{anc::}n]\!] & = & [\![f'/\texttt{anc::}n]\!], [\![f'/\texttt{self::}n]\!] & \text{if } f' = f\,\langle \bar{1} \rangle \\
[\![f/\texttt{anc::}n]\!] & = & [\![f'/\texttt{anc::}n]\!] & \text{if } f' = f\,\langle \bar{2} \rangle \\
[\![f/\texttt{anc::}n]\!] & = & \epsilon & \text{if } f = (t, \mathsf{Top}) \\
[\![f/\texttt{desc::}n]\!] & = & \Pi_{f_1,\ldots,f_m} [\![f_i/\texttt{self::}n]\!], [\![f_i/\texttt{desc::}n]\!] & \text{if } [\![f/\texttt{child::}*]\!] = [f_1, \ldots, f_m] \\
[\![f/\texttt{desc::}n]\!] & = & \epsilon & \text{if } [\![f/\texttt{child::}*]\!] = \epsilon
\end{array}
$$

Auxiliary definitions: $\texttt{name}((\sigma[tl], c)) = \sigma$ and $n_1 \leq n_2$ if and only if $n_1 = n_2$ or $n_2 = *$

Figure 2: Semantics of the navigational fragment of XQuery core

An XQuery expression *e* evaluates to a value s, which is defined as a sequence of focused trees as in Genevès and Gesbert (2015). This definition of values allows us to define the semantics in a compositional way. We write $[f_1, \ldots, f_n]$ for $f_1 :: \ldots :: f_n :: \epsilon$ and s_1, s_2 for a sequence concatenation of s_1 and s_2. In XQuery, all values are sequences and a single item (or tree) is considered a singleton sequence that contains that item (or tree). Hence in the rest of the paper we use f and $[f]$ interchangeably.

1.3 Semantics

Figure 2 defines the semantics of the navigational fragment of XQuery core. The semantics of the language is defined by the following denotation function:

$$[\![\_]\!] \ : \ \mathit{Substitution} \rightarrow \mathit{Expression} \rightarrow \mathit{Value}$$

A substitution η is a partial map from variables to values. In Figure 2, we assume that each η is well-formed, meaning that a for-loop variable $\$v$ is only bound to a single focused tree.

While most of the rules are straightforward and compositional, we took special care for element constructor ($\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u$) which combines a tree construction and a dynamic typecheck. First, suppose that the inner expression *e* evaluates to a sequence $[f_1, \ldots, f_n]$ of focused trees, where $f_i = (t_i, c_i)$. Note that each focused tree has its own context. Then we embed them into a new XML tree, namely $\sigma[f_1 :: \ldots :: f_n :: \epsilon]$, whose context is Top. When navigating it, we need to update the context with respect to the new tree node. Therefore, we remove the old context from each focused tree f_i and obtain $f = (\sigma[t_1 :: \ldots :: t_n :: \epsilon], \mathsf{Top})$ as in Castagna et al. (2015); Genevès and Gesbert (2015). Finally, we check if the constructed tree f conforms to the annotated type ($f \in [\![u]\!]$), and if so, return it as the result of evaluating ($\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u$); $[\![u]\!]$ is formally defined in Definition 2.2.

To evaluate a for-loop expression `for` $\$v$ `in` e_1 `return` e_2 with substitution η, we first evaluate $[\![e_1]\!]_\eta$. If the result is not an empty sequence, say $[f_1, \ldots, f_n]$, then for each focused tree f_i, we evaluate the for-loop body e_2 with an extended substitution $\eta, \$v \mapsto f_i$. Finally we concatenate the results of evaluating $[\![e_2]\!]_{\eta, \$v \mapsto f_i}$ for $i = 1, \ldots, n$ in order. In contrast, if $[\![e_1]\!]_\eta$ evaluates to an empty sequence, then the for-loop expression also evaluates to an empty sequence.

To evaluate an axis expression $\$v/axis::n$, we analyze the shape of the focused tree bound to the for-loop variable $\$v$. The definition of $[\![f/axis::n]\!]$ follows from the intuition behind the axis *axis*. For example, $[\![f/\texttt{self::}n]\!]$ evaluates to a singleton sequence $[f]$ if and only if the label of f matches the name test n, denoted by $\texttt{name}(f) \leq n$. The semantics of `child` is defined using `self` and `nsibl` applied to the child node. Note that $f\langle\bar{1}\rangle$ and $f\langle\bar{2}\rangle$ are never both defined for the same f and thus the definitions for the semantics of `parent` are mutually exclusive (the same is true for `anc`). $[\![f/\texttt{nsibl::}n]\!]$ and $[\![f/\texttt{psibl::}n]\!]$ recursively apply `nsibl` and `psibl` to the next and preceding sibling of f, respectively, if there exists such a node. $[\![f/\texttt{desc::}n]\!]$ applies `self` and `desc` recursively to each child node of f and concatenates the results into a sequence.

2 Type Language

As in the recent work by Genevès and Gesbert (2015), our type language is based on regular tree types Hosoya et al. (2005) and a tree logic, which is a sub-logic of the alternation free modal μ-calculus with converse Genevès et al. (2007). In this section, we first briefly introduce regular tree types and the tree logic, together with their semantics in terms of sets of XML values. Then we introduce our type language, regular tree types enriched with tree logic formulas.

2.1 Regular Tree Types

We use a slight variant of XDuce's regular expression type language Hosoya and Pierce (2003) to type sequences of XML trees (or *elements*), which is expressive enough to capture standard XML types such as DTD and XML Schema Murata et al. (2005). Formally we define our regular tree types as follows.

Definition 2.1 (Regular tree types)**.**

$$
\begin{array}{lrcl}
\textit{Unit types} & u & ::= & \texttt{element}\ \mathrm{n}\ \{\tau\} \\
\textit{Name tests} & \mathrm{n} & ::= & \sigma \mid * \\
\textit{Sequence types} & \tau & ::= & u \mid \texttt{()} \mid \tau, \tau \mid (\tau \mid \tau) \mid \tau^* \mid x
\end{array}
$$

A sequence type τ is a regular expression over unit types, where a unit type u, or a "prime type" in the XQuery terminology, corresponds to an XML element. (In general, u may also include primitive types such as Int or String, but for simplicity, we consider only element types.) As usual, we use the following abbreviations: $\tau^+ \equiv \tau, \tau^*$ and $\tau^? \equiv \texttt{()} \mid \tau$. (We use $\equiv$ both for syntactic equivalence and syntactic sugar.)

While the Kleene star * operator supports horizontal recursion, we use *type environments* and type variables to support vertical recursion. A type environment E is a mapping from type variables x to types τ. The variables bound in E may be defined in a mutually recursive way, but recursion must be guarded by an element type to ensure well-formedness of types, *i.e.,* contractiveness of recursive types Amadio and Cardelli (1993). We also assume that regular expressions defined by E are composed of mutually exclusive unit types and *1-unambiguous* Brüggemann-Klein and Wood (1998), which is standard and comes from XML Schema.

As usual, the semantics of regular tree types is defined as sets of forests, *i.e.,* sets of sequences of trees, and the subtyping relation is semantically defined as the set inclusion relation.

Definition 2.2. *Given a type environment E, the semantics of types is defined as the smallest function $[\![\_]\!]_E$ that satisfies the following set of equations:*

$$
\begin{array}{rrr}
[\![x]\!]_E = [\![E(x)]\!]_E & [\![\texttt{()}]\!]_E = \{\epsilon\} & [\![\tau \mid \tau']\!]_E = [\![\tau]\!]_E \cup [\![\tau']\!]_E \\
[\![\tau^0]\!]_E = \{\epsilon\} & [\![\tau^{n+1}]\!]_E = [\![\tau, \tau^n]\!]_E & [\![\tau^*]\!]_E = \bigcup_{n \in \mathbb{N}} [\![\tau^n]\!]_E \\
\multicolumn{3}{l}{[\![\texttt{element}\ \sigma\ \{\tau\}]\!]_E = \{[\sigma[tl]] \mid tl \in [\![\tau]\!]_E\}} \\
\multicolumn{3}{l}{[\![\texttt{element}\ *\ \{\tau\}]\!]_E = \{[\sigma[tl]] \mid \sigma \in \Sigma \textit{ and } tl \in [\![\tau]\!]_E\}} \\
\multicolumn{3}{r}{[\![\tau, \tau']\!]_E = \{[t_1, \ldots, t_n, t'_1, \ldots, t'_m] \mid [t_1, \ldots, t_n] \in [\![\tau]\!]_E \textit{ and } [t'_1, \ldots, t'_m] \in [\![\tau']\!]_E\}}
\end{array}
$$

Then, a type τ_1 is a subtype of τ_2, denoted by $\tau_1 <: \tau_2$, if and only if $[\![\tau_1]\!]_E \subseteq [\![\tau_2]\!]_E$.

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

2.1.1 Limitations

The regular tree type language we gave above is standard and used to define the static type system in the XQuery standard and its various improvements in the literature. In such a type system, an XQuery expression is associated with a regular tree type, and the notion of a value (*i.e.,* a sequence of tree nodes) *matching* a type can be defined as follows when nodes are represented as focused trees Genevès and Gesbert (2015):

Definition 2.3. *The* focused-tree interpretation $[\![\tau]\!]\uparrow$ *of a type* τ *is defined as the set:*

$$\{[(t_1, c_1) \dots (t_n, c_n)] \mid [t_1 \dots t_n] \in [\![\tau]\!]\}$$

A value s *is said to* match *a type* τ *if* $s \in [\![\tau]\!]\uparrow$.

As shown in the above definition, regular tree types denote sequences of trees, and their interpretation is lifted to sequences of focused trees by simply ignoring the context part. In other words, using regular tree types, the type system cannot properly address expressions that analyze the shape of the context of a given focused tree: given f of type τ, we cannot deduce a precise type for $f\,\langle\bar{1}\rangle$, $f\,\langle\bar{2}\rangle$, and $f\,\langle 2\rangle$ because when $f = (t, c)$, τ only contains information about t, but those expressions require information about c. More specifically, consider an expression `for` $\$v$ `in` *e* `return` $\$v/\texttt{psibl::}*$. Suppose that *e* is of type τ and reduces to $[f_1, \dots, f_n]$. Then, according to the reduction semantics, we compute $f_i/\texttt{psibl::}*$ for each f_i and concatenate their results. Since evaluating $f_i/\texttt{psibl::}*$ depends only on $f_i\,\langle\bar{2}\rangle$, analyzing τ is not enough to deduce any meaningful type for $f_i/\texttt{psibl::}*$, and thus for the entire for-expression. Therefore, most previous type systems for XQuery built upon the regular tree type language simply give to this expression the most general type AnyElt^*.

In the next section, we propose to use a tree logic to solve this problem as in Genevès and Gesbert (2015).

2.2 A Tree Logic

To describe sets, *i.e.,* types, of focused trees rather than just sets of trees, we use a variant of the logic language defined in Genevès et al. (2007). The tree logic, defined below, is expressive enough to support all XQuery types, and the satisfiability problem for a logical formula of size n can efficiently be decided with an optimal $2^{\mathcal{O}(n)}$ worst-case time complexity bound Genevès et al. (2015).

Definition 2.4 (Logic formulas)**.**

$$\begin{array}{rcl} a & ::= & \langle 1\rangle \mid \langle 2\rangle \mid \langle\bar{1}\rangle \mid \langle\bar{2}\rangle \\ \varphi, \psi & ::= & \top \mid \sigma \mid \neg\sigma \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \langle a\rangle\,\varphi \mid \neg\,\langle a\rangle\,\top \mid X \mid \mu(X_i = \varphi_i)_{i\in I}\ \textit{in}\ \psi \end{array}$$

$a \in \{1, 2, \bar{1}, \bar{2}\}$ are *programs*, corresponding to the four directions where trees can be navigated. A program is used in an existential formula $\langle a\rangle\,\varphi$, denoting the existence of a subtree at the direction of a that satisfies the sub-formula φ. Other formulas include the truth predicate $\top$, atomic propositions σ (denoting the label of the focused tree), disjunction and conjunction of formulas, and least n-ary fixed points. We also use the following abbreviations: $\bot$ to mean $\neg\top$, $[a]\,\varphi$ for $\neg\,\langle a\rangle\,\top \vee \langle a\rangle\,\varphi$, and $\mu X.\varphi$ for $\mu(X = \varphi)$ in φ. The universal modality $[a]\,\varphi$ encodes that a subtree at the direction of a does not exist, or else it satisfies φ.

The semantics of a logical formula is defined as the set of focused trees such that the formula is satisfied at the current node. We use the following interpretation function:

$$\langle\!|-|\!\rangle : \mathit{Formula} \to \mathit{Substitution} \to \mathit{FocusedTreeSet}$$

where a substitution V is a finite map from recursion variables to sets of focused trees. In the definition below, we use $\mathcal{F}$ to denote the set of all focused trees and $\texttt{name}(f)$ to denote the label at the current node of f.

Definition 2.5 (Interpretation of formulas)**.**

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

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

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

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

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

In the rest of the paper, we consider only closed formulas and write $\langle\!\langle \psi \rangle\!\rangle$ for $\langle\!\langle \psi \rangle\!\rangle_\emptyset$.

2.3 Formula-Enriched Sequence Types

Although now we can describe sequences of trees with regular tree types and sets of focused trees with logical formulas, we still cannot type *sequences* of focused trees, which is the values of our XQuery core. To this end, as in Genevès and Gesbert (2015), we simply enrich the regular expressions of unit types in Definition 2.1 by associating a formula to each unit type. The enriched types, which we call formula types, are thus regular expressions of pairs of a unit type and a formula, as defined below.

Definition 2.6 (Formula types)**.**

$$\rho \quad ::= \quad (\varphi, u) \mid () \mid \rho, \rho \mid (\rho \mid \rho) \mid \rho^+$$

Basically a formula type $(\varphi, \mathtt{element}\ n\ \{\tau\})$ describes a focused tree of the form $(\sigma[tl], c)$ where n describes σ, τ describes tl, and φ describes c as well as both σ and tl, respectively. Note that the context information is described only by the formula. The interpretation of a pair (φ, u) is defined as the set of singleton sequences of focused trees which match both φ and u:

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

From this, the semantics of formula types in terms of sets of sequences of focused trees is defined in the obvious manner. Then the subtyping relation $\rho_1 <: \rho_2$ is also semantically defined as the set inclusion relation $[\![\rho_1]\!] \subseteq [\![\rho_2]\!]$.

In Section 1.2, we assumed that every XML element constructor was annotated with not a formula type (φ, u) but a unit type u. The reason is that an element constructor always reduces to a single tree node whose context is Top, and thus there is no need to use a formula type for the annotation. We simply consider u to be $(\top, u)$.

3 Inference for XPath Axes

In this section, we present a backward type inference system for XPath axes only, and based on this we will develop a type inference system for the XQuery core in Section 4.

In backward type inference, we are given an expression e and an output type ρ_o of XML values that e may produce. Then we infer an input type ρ_i such that for any XML document t of type ρ_i, $e(t)$ always produces a sequence of XML nodes of type ρ_o. When considering XPath axes, this means that we infer the type describing a set of input focused trees such that when applied to an axis, each input tree produces a sequence of nodes that has the output type ρ_o. More precisely, since XPath axes can only be applied to a for-loop variable in our XQuery core, we infer from the given axis *axis* and output type ρ, a single formula type (φ, u) (possibly their union) that the input tree, *i.e.,* the for-loop variable, must satisfy. In particular, we design the inference rules in such a way that the following invariant holds: if (φ, u) is the inferred input type, a subtype relation $\varphi <: u$ holds, that is, for any t and c, if $(t, c) \in \langle\!\langle \varphi \rangle\!\rangle$, then $t \in [\![u]\!]$. By ensuring this invariant, we obtain a sound and complete inference system for XPath axes.

$$\textsc{Self-Empty}\quad \frac{}{(\neg k(n), \mathsf{AnyElt}) \leftarrow \texttt{self::}n,\ ()} \qquad \textsc{Self-Formula}\quad \frac{}{(\varphi \wedge k(n) \wedge \mathrm{form}(u),\ u) \leftarrow \texttt{self::}n,\ (\varphi, u)}$$

$$\textsc{Self-Seq1}\quad \frac{\neg \mathit{nullable}(\rho_i) \qquad \mathit{nullable}(\rho_j) \qquad \rho' \leftarrow \texttt{self::}n,\ \rho_i}{\rho' \leftarrow \texttt{self::}n,\ (\rho_1, \rho_2)}\ (i \neq j)$$

$$\textsc{Self-Seq2}\quad \frac{\mathit{nullable}(\rho_1) \qquad \mathit{nullable}(\rho_2) \qquad \rho'_i \leftarrow \texttt{self::}n,\ \rho_i}{\rho'_1 \mid \rho'_2 \leftarrow \texttt{self::}n,\ (\rho_1, \rho_2)}\ (i = 1, 2) \qquad \textsc{Self-Or}\quad \frac{\rho'_i \leftarrow \texttt{self::}n,\ \rho_i}{\rho'_1 \mid \rho'_2 \leftarrow \texttt{self::}n,\ (\rho_1 \mid \rho_2)}\ (i = 1, 2)$$

$$\textsc{Self-Plus}\quad \frac{\rho' \leftarrow \texttt{self::}n,\ \rho}{\rho' \leftarrow \texttt{self::}n,\ \rho^+} \qquad \textsc{Parent}\quad \frac{\rho' \leftarrow \texttt{self::}n,\ \rho}{\texttt{child-type}(\rho') \leftarrow \texttt{parent::}n,\ \rho}$$

Auxiliary definitions:

$$\begin{array}{rcl@{\qquad}rcl}
k(*) & = & \top & k(\sigma) & = & \sigma \\
\mathit{nullable}(\,()\,) & = & \mathsf{true} & \mathit{nullable}(\rho_1, \rho_2) & = & \mathit{nullable}(\rho_1) \wedge \mathit{nullable}(\rho_2) \\
\mathit{nullable}((\varphi, u)) & = & \mathsf{false} & \mathit{nullable}(\rho_1 \mid \rho_2) & = & \mathit{nullable}(\rho_1) \vee \mathit{nullable}(\rho_2) \\
\mathit{nullable}(\rho^+) & = & \mathit{nullable}(\rho) & & & \\
\mathit{Prime}(\,()\,) & = & () & \mathit{Prime}(\tau_1, \tau_2) & = & \mathit{Prime}(\tau_1) \mid \mathit{Prime}(\tau_2) \\
\mathit{Prime}(u) & = & u & \mathit{Prime}(\tau_1 \mid \tau_2) & = & \mathit{Prime}(\tau_1) \mid \mathit{Prime}(\tau_2) \\
\mathit{Prime}(\tau^+) & = & \mathit{Prime}(\tau) & & & \\
\mathsf{distrib}(\chi, \,()\,) & = & () & \mathsf{distrib}(\chi, (\tau_1, \tau_2)) & = & (\mathsf{distrib}(\chi, \tau_1), \mathsf{distrib}(\chi, \tau_2)) \\
\mathsf{distrib}(\chi, u) & = & (\chi, u) & \mathsf{distrib}(\chi, \tau_1 \mid \tau_2) & = & (\mathsf{distrib}(\chi, \tau_1) \mid \mathsf{distrib}(\chi, \tau_2)) \\
\mathsf{distrib}(\chi, \tau^+) & = & \mathsf{distrib}(\chi, \tau)^+ & & &
\end{array}$$

$$\begin{array}{rcl}
\texttt{child-type}(\rho_1 \mid \rho_2) & = & \texttt{child-type}(\rho_1) \mid \texttt{child-type}(\rho_2) \\
\texttt{child-type}((\varphi, \texttt{element}\ n\ \{\tau\})) & = & \mathsf{distrib}(\texttt{has-parent}(\varphi), \mathit{Prime}(\tau)) \\
\texttt{has-parent}(\chi) & = & \mu Z.\, \langle \bar{1} \rangle\, \chi \vee \langle \bar{2} \rangle\, Z
\end{array}$$

Figure 3: Inference rules for `self` and `parent`

Formally, the subtype relation $\varphi <: u$ between formula φ and unit type u can be checked in two steps. First, we translate u into a downward-only formula which is true at any tree node matching this unit type, regardless of its context. Technically, this translation can be done using an auxiliary function $\mathrm{form}(u)$, which is defined and proved correct in Genevès and Gesbert (2015). (For its precise definition, we refer the reader to Figure 10 in Genevès and Gesbert (2015).) Next, we test the satisfiability of the formula $\varphi \wedge \neg \mathrm{form}(u)$, for example, using the decision procedure presented in Genevès et al. (2015); in fact, $\langle\!\langle \varphi \wedge \neg \mathrm{form}(u) \rangle\!\rangle = \emptyset$ if and only if any focused tree matching φ also satisfies u, *i.e.,* $\langle\!\langle \varphi \rangle\!\rangle \subseteq [\![u]\!]\uparrow$.

Below we present inference rules using a judgment of the form $\rho_i \leftarrow \mathit{axis}\texttt{::}n,\ \rho_o$ where input type ρ_i is always of the form $(\varphi_1, u_1) \mid \ldots \mid (\varphi_n, u_n)$. We first look into the inference rules for `self` and `parent`.

3.1 Inference Rules for `self` and `parent`

3.1.1 Self

Figure 3 shows inference rules for `self`. Basically `self::`*n* returns a singleton sequence containing the input tree if it satisfies name test *n*; otherwise it returns an empty sequence. Conversely, if the output type is `()`, it means that the input tree fails the name test and thus has type $\neg k(n)$ (rule SELF-EMPTY). Here $k(n)$ is the translation of *n* into a corresponding formula.

If the output type is a single formula type (φ, u), it means that the input tree has that type: more precisely, the input tree should satisfy both φ and $k(n)$, and at the same time should have type u (rule SELF-FORMULA). All these constraints are encoded in the formula $\varphi \wedge k(n) \wedge \mathrm{form}(u)$ where we translate the unit type u into a formula using the function $\mathrm{form}(u)$. Using this function, we guarantee that the inferred formula is a subtype of the inferred unit type. Note that in the rule SELF-FORMULA, $\varphi \wedge k(n) \wedge \mathrm{form}(u) <: u$ holds. In addition, when $u = \texttt{element}\ \sigma\ \{\tau\}$, inference fails if $n = \sigma'$ and $\sigma \neq \sigma'$. In this case, there is no input tree that when applied to `self::`n, produces a tree of type `element` $n'\ \{\tau\}$ because no tree node can have different labels at the same time.

If the output type is a sequence type (ρ_1, ρ_2), at least one type needs to be nullable (*i.e.,* the interpretation of the type includes an empty sequence ϵ) since `self::`n returns at most one tree as output. The type of the input tree is then the type inferred from the non-nullable part of the output type (rule SELF-SEQ1). If both ρ_1 and ρ_2 are nullable, we take the union of the input types inferred from them (rule SELF-SEQ2). When the output type is a union type, the input tree may also have a union type of the two, each of which is inferred from one summand of the output type (rule SELF-OR). Lastly, if the output type is a plus type ρ^+, the input type should be inferred from ρ since `self::`n returns at most one node (rule SELF-PLUS).

3.1.2 Parent

The intuition behind inference rules for `parent` is simple. Given an output type ρ, we first infer an input type ρ' using the inference rules for `self`. Note that ρ' is the type of the parent of the input context node, that is, the context node is a child of the node of type ρ'. Therefore, we infer the most general child type from ρ' using auxiliary function $\texttt{child-type}(\rho')$, which exploits several other functions. First, $Prime(\tau)$ extracts all unit types at top level of τ and constructs their disjunction Colazzo and Sartiani (2011). Second, $\mathsf{distrib}(\chi, \tau)$ distributes the formula χ to all unit types in the regular expression τ Genevès and Gesbert (2015). While we can infer the exact formula (*e.g.,* ψ) that is true at any child node using a formula (*e.g.,* φ) that is true at its parent (using $\texttt{has-parent}(\varphi)$), we can infer only a disjunction of unit types for a child from a regular tree type of the parent. Hence we use the function $\mathsf{distrib}(\_, \_)$ to obtain a type for a single input context node, *i.e.,* a type of the form $(\psi, u_1) \mid \ldots \mid (\psi, u_n)$.

Although we do not give a precise rule in Figure 3, to get the best precision, if the output type is nullable, that is, if the input node may be a root, we add a formula φ_{root} to each formula φ_i using a disjunction, *i.e.,* $\varphi_{root} \vee \varphi_i$, in the inferred input type $(\varphi_1, u_1) \mid \ldots \mid (\varphi_n, u_n)$. Here φ_{root} specifies that a given node is a root and is defined as $\neg \langle \bar{1} \rangle \top \wedge \neg \langle \bar{2} \rangle \top \wedge \neg \langle 2 \rangle \top$. Note that we cannot specify the fact that the context node may be a root in the regular tree type part.

3.2 Other Axes

Given an axis other than `self` and `parent` and output formula type (φ, u), while it is possible to specify the exact shape of the input tree in terms of the formula φ, it is impossible in terms of a regular tree type because u does not contain enough information about the context of the input node. Hence, for other axes, we approximate the unit type part in the inferred input type. However, the formula part is still exact and captures all the information contained in the output type. More precisely, for the unit type part in the input type, we simply infer AnyElt for `psibl`, `nsibl`, `anc`, and `desc`, while inferring a more precise type for `child`. As studied in Colazzo and Sartiani (2011); Genevès and Gesbert (2015), in forward inference systems using regular tree types, one can infer precise types only for `self`, `child`, and `desc`. In contrast, in our backward inference system, we can infer precise regular tree types only for `self`, `parent`, and `child`.

One important difference between `self` and `parent` and other axes is that while the former requires us to inspect only a single node in the input tree, the rest of the axis expressions requires us to inspect a sequence of nodes reached by navigating the axis from the input node and combine the constraints for all those nodes. To combine a set of constraints on the input tree, we use an additional judgment of the form $\varphi \leftarrow \mathit{axis}::n,\ \rho\ \mathsf{with}\ \psi$, which means that when applied to *axis*`::`n, some fragment of the input tree at the direction of *axis* from the context node produces a sequence of nodes of type ρ, and at the same time, ψ is true at the lastly reached node by navigating *axis* within the fragment. Note that using this judgment, we infer only a formula. We infer a unit type for the input node using auxiliary functions.

To illustrate the meaning of the judgment $\varphi \leftarrow \mathit{axis}::n,\ \rho\ \mathsf{with}\ \psi$, consider an example input tree node represented as a binary tree in Figure 4. Suppose that *axis* is `psibl` and C is the context node. Then, some nodes between A and B are chosen if they satisfy name test n, and they have type ρ. Moreover, ψ is true at the node

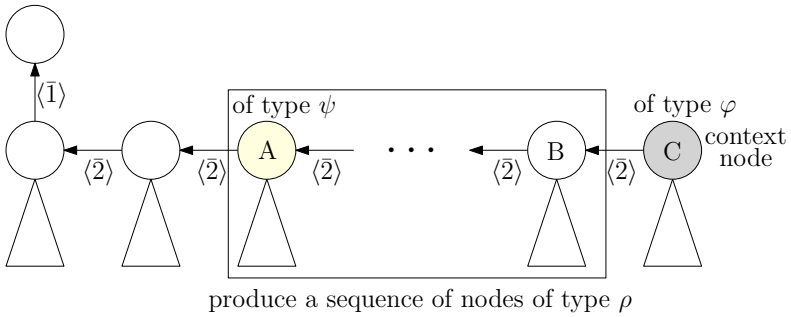


Figure 4: Interpretation of $\varphi \leftarrow$ `psibl::`n, ρ with ψ when the context node is C: A and B are some nodes reached by navigating `psibl` from C.

$$\textsc{Psibl}\ \frac{\varphi \leftarrow \texttt{psibl::}n,\ \rho \text{ with } \mu X.\,[\bar{2}]\,(\neg k(n) \wedge X)}{(\varphi, \mathsf{AnyElt}) \leftarrow \texttt{psibl::}n,\ \rho}$$

$$\textsc{Psibl-Formula}\ \frac{\varphi' = \langle\bar{2}\rangle\,(\mu X.(\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \psi) \vee (\neg k(n) \wedge \langle\bar{2}\rangle\, X))}{\varphi' \leftarrow \texttt{psibl::}n,\ (\varphi, u) \text{ with } \psi}$$

Figure 5: Inference rules for `psibl`

A. In this case, the with parameter ψ specifies the constraints on the nodes reached by further navigating `psibl` from A. Finally, the context node C has the inferred type φ. In the subsection below, we give a more precise interpretation of the judgment when the output type is a sequence type of the form (ρ_1, ρ_2).

With this interpretation, given *axis*`::`n and output type ρ, we first infer a formula φ by choosing the correct initial formula ψ_{init} and using the judgment $\varphi \leftarrow$ *axis*`::`n, τ with ψ_{init}. Then, we compute a unit type u using an appropriate auxiliary function depending on *axis*. When computing u, we ensure a subtype relation $\varphi <: u$. Finally, the input type is determined as a pair (φ, u) as shown below:

$$\frac{\varphi \leftarrow \textit{axis}\texttt{::}n,\ \rho \text{ with } \psi_{init} \qquad u = \mathsf{aux\_func}(\rho)}{(\varphi, u) \leftarrow \textit{axis}\texttt{::}n,\ \rho}$$

3.2.1 Preceding Sibling

`psibl::`*n* returns in document order (*i.e.,* pre-order) preceding siblings of the context node that satisfy name test *n*. Thus, given `psibl::`*n* and an output type, we only infer the constraint (*i.e.,* type) on the preceding siblings. The rest of the nodes can have arbitrary structure.

Figure 5 shows the inference rules for `psibl`. In the rule PSIBL, we initially assume that there is no preceding sibling satisfying name test *n*, that is, $\mu X.\,[\bar{2}]\,(\neg k(n) \wedge X)$. Then we analyze the output type ρ using the judgment of the form $\varphi \leftarrow$ `psibl::`n, ρ with ψ. In this judgment, one important invariant is that ψ is true at the leftmost preceding sibling returned by `psibl::`n when the output type is ρ. When the inferred input formula is φ, the final input type is a pair $(\varphi, \mathsf{AnyElt})$. Since we cannot extract any meaningful information about the context node from the regular tree types of its preceding siblings, we simply use AnyElt.

When the output type is just a pair (φ, u) and the with parameter is ψ, it means that there should be a preceding sibling satisfying name test *n* such that both φ and ψ are also true. Moreover, that sibling node should also have type u. All these constraints are encoded in the inferred formula φ' in the rule PSIBL-FORMULA. As in the rule SELF-FORMULA, we use function $\mathrm{form}(u)$ to translate the unit type u to a corresponding formula. In addition, since the initial with parameter given in the rule PSIBL guarantees that there is no preceding sibling satisfying name test n, the two rules guarantee that if `psibl::`*n* returns a single node, then the context node has only one preceding sibling satisfying *n*.

AXIS-EMPTY

$$\frac{}{\psi \leftarrow \mathit{axis}::n,\ ()\ \mathsf{with}\ \psi}$$

AXIS-OR

$$\frac{\varphi_i \leftarrow \mathit{axis}::n,\ \rho_i\ \mathsf{with}\ \psi}{\varphi_1 \vee \varphi_2 \leftarrow \mathit{axis}::n,\ (\rho_1 \mid \rho_2)\ \mathsf{with}\ \psi}\ (i = 1, 2)$$

AXIS-BACKWARD-SEQ

$$\frac{\varphi_1 \leftarrow \mathit{axis}::n,\ \rho_1\ \mathsf{with}\ \psi \qquad \varphi_2 \leftarrow \mathit{axis}::n,\ \rho_2\ \mathsf{with}\ \varphi_1}{\varphi_2 \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)\ \mathsf{with}\ \psi}\ (\mathit{axis}\ \text{is}\ \texttt{psibl}\ \text{or}\ \texttt{anc})$$

AXIS-FORWARD-SEQ

$$\frac{\varphi_2 \leftarrow \mathit{axis}::n,\ \rho_2\ \mathsf{with}\ \psi \qquad \varphi_1 \leftarrow \mathit{axis}::n,\ \rho_1\ \mathsf{with}\ \varphi_2}{\varphi_1 \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)\ \mathsf{with}\ \psi}\ (\mathit{axis}\ \text{is}\ \texttt{nsibl}\ \text{or}\ \texttt{desc})$$

AXIS-PLUS

$$\frac{\varphi \leftarrow \mathit{axis}::n,\ \rho\ \mathsf{with}\ X \vee \psi}{\mu X.\varphi \leftarrow \mathit{axis}::n,\ \rho^{+}\ \mathsf{with}\ \psi}\ (X\ \text{fresh})$$

Figure 6: Common inference rules for `psibl`, `anc`, `nsibl`, and `desc`

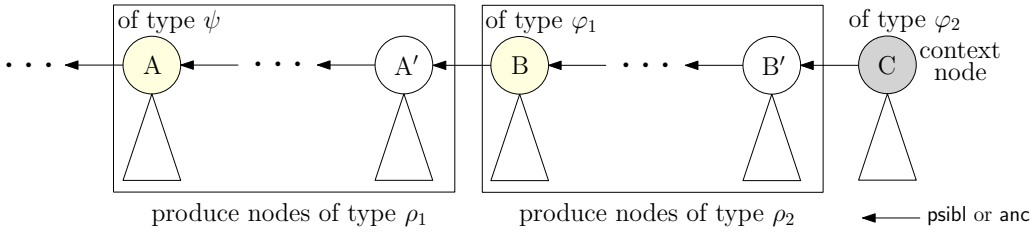


Figure 7: Interpretation of $\varphi \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)$ with ψ when *axis* is a backward axis `psibl` or `anc`: we analyze the sequence type from left to right. We first infer φ_1 and then φ_2.

The rest of the inference rules for empty, sequence, union, and repetition types are generic and are also used for other axes—`anc`, `nsibl`, and `desc`. (When the output type is a sequence type, we distinguish backward axes from forward axes, and thus present two inference rules.) The common rules are given in Figure 6. The first two rules are easy. If the output type is an empty type, the inferred input type is simply the formula ψ given as the with parameter (rule AXIS-EMPTY). Therefore, in combination with the rule PSIBL, the inferred formula in the rule AXIS-EMPTY specifies that no preceding sibling of the input node should satisfy the name test. If the output type is a union type of two, we infer a formula from each and return the union of the two inferred formulas (rule AXIS-OR).

When the output type is a sequence type (ρ_1, ρ_2), our analysis begins with the last node among the nodes reached by navigating the given axis and proceeds towards the context node. Therefore, if the given axis is a backward axis such as `psibl` and `anc`, we analyze the output type from left to right (rule AXIS-BACKWARD-SEQ).

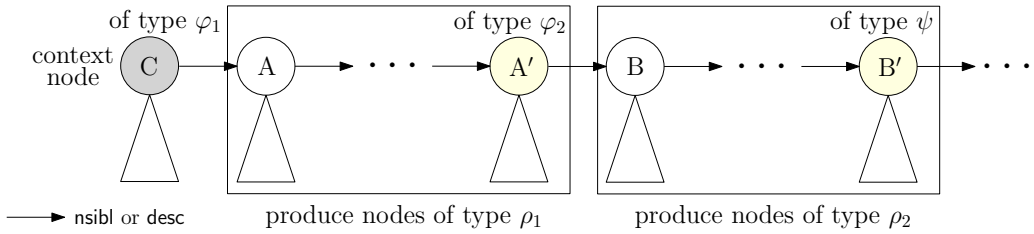


Figure 8: Interpretation of $\varphi \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)$ with ψ when *axis* is a forward axis `nsibl` or `desc`: we analyze the sequence type from right to left. We first infer φ_2 and then φ_1.

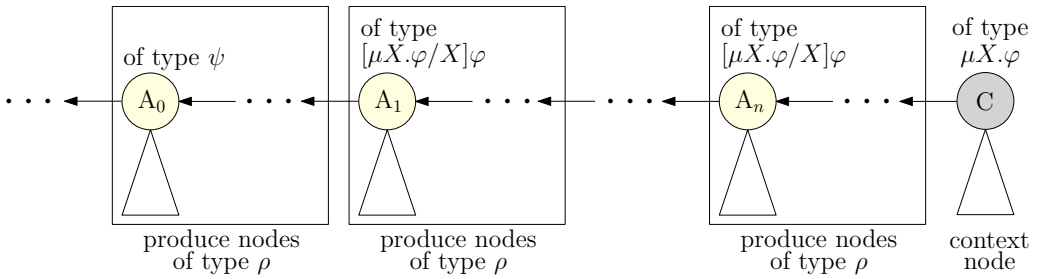


Figure 9: Interpretation of $\varphi \leftarrow \mathit{axis}::n,\ \rho^+$ with ψ when *axis* is a backward axis. A similar illustration can be applied to forward axes.

$$\texttt{has-anc}(\chi) = \mu Z.\, \langle \bar{1} \rangle\, (\chi \vee Z) \vee \langle \bar{2} \rangle\, Z$$

$$\textsc{Anc}\ \frac{\varphi \leftarrow \texttt{anc}::n,\ \rho \text{ with } \neg\texttt{has-anc}(k(n))}{(\varphi, \mathsf{AnyElt}) \leftarrow \texttt{anc}::n,\ \rho}$$

$$\textsc{Anc-Formula}\ \frac{\varphi' = \mu X.\, \langle \bar{1} \rangle\, ((\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \psi) \vee (\neg k(n) \wedge X)) \vee \langle \bar{2} \rangle\, X}{\varphi' \leftarrow \texttt{anc}::n,\ (\varphi, u) \text{ with } \psi}$$

Figure 10: Inference rules for `anc`

More precisely, as depicted in Figure 7, given a judgment $\varphi \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)$ with ψ, we can conceptually divide the nodes reached by navigating *axis* from context node C into two parts: the nodes from A to A$'$ and those from B to B$'$ that produce a sequence of nodes of type ρ_1 and ρ_2, respectively, where the first part precedes the second part in document order. In particular, ψ is true at node A which is the first node in the first part. We first infer a formula φ_1 from ρ_1 and ψ using the judgment $\varphi_1 \leftarrow \mathit{axis}::n,\ \rho_1$ with ψ. Then φ_1 is true at node B which is next to A$'$ in document order and also the first node in the second part. Next, we infer a formula φ_2 from ρ_2 and φ_1 using the judgment $\varphi_2 \leftarrow \texttt{psibl}::n,\ \rho_2$ with φ_1. Finally, φ_2 is true at the context node and is returned as the input type.

The interpretation of the judgment $\varphi \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)$ with ψ is dual if *axis* is a forward axis such as `nsibl` and `desc`. In this case, we analyze the output type from right to left, *i.e.,* ρ_2 first (rule AXIS-FORWARD-SEQ). For example, as depicted in Figure 8, from ρ_2 and ψ, we first infer a constraint on the last node B$'$ at which ψ is true, among the nodes reached by navigating *axis* from context node C, and subsequently infer constraints on the nodes appearing before B$'$ in reverse order, *e.g.,* from B through A$'$ to A, until finally inferring the constraint on the context node.

When the output type is a repetition type ρ^+, we introduce a fresh recursion variable X (rule AXIS-PLUS). Then, we infer a formula φ from output type ρ and parameter $X \vee \psi$ using the judgment $\varphi \leftarrow \mathit{axis}::n,\ \rho$ with $X \vee \psi$. More precisely, as depicted in Figure 9, there exists a block of nodes reached by navigating *axis* from context node C, *e.g.,* the nodes from A_n to the node before C, that produce a sequence of nodes of type ρ, each of which satisfies name test *n*. Moreover, the lastly reached node A_n and C should respectively satisfy $X \vee \psi$ and φ (where φ contains $X \vee \psi$ as a subformula, for example, see the rule PSIBL-FORMULA). If A_n satisfied X, that is, $[\mu X.\varphi/X]\varphi$, there would be more blocks of nodes reached by further navigating *axis* from A_n that would produce nodes of type ρ, where the lastly reached node in each block, *e.g.,* A_1, would also satisfy X. This recursion terminates when some node satisfies ψ rather than X, *e.g.,* A_0 (where the block of nodes containing A_0 should also produce a sequence of nodes of type ρ). Lastly, the closed recursive formula $\mu X.\varphi$ is returned as the input type of the context node.

3.2.2 Ancestor

Inference rules for `anc::`*n* are the same as those for `psibl::`*n* with two exceptions (they are both backward axes and use the same set of rules in Figure 6): first the interpretation of the judgment and the initial value of the with parameter, and second the input type inferred when the output type is a single formula type (φ, u). We briefly

$$\textsc{Nsibl}\quad \frac{\varphi \leftarrow \texttt{nsibl::}n,\ \rho \text{ with } \mu X.\,[2]\,(\neg k(n) \wedge X)}{(\varphi, \mathsf{AnyElt}) \leftarrow \texttt{nsibl::}n,\ \rho}$$

$$\textsc{Nsibl-Formula}\quad \frac{\varphi' = \langle 2\rangle\,(\mu X.(\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \psi) \vee (\neg k(n) \wedge \langle 2\rangle\, X))}{\varphi' \leftarrow \texttt{nsibl::}n,\ (\varphi, u) \text{ with } \psi}$$

Figure 11: Inference rules for `nsibl`

explain them in turn.

First, the interpretation of a judgment $\varphi \leftarrow \texttt{anc::}n,\ \rho$ with ψ is as follows: there is a block of nodes reached by navigating `anc` from the context node such that it produces a sequence of nodes of type ρ, each of which satisfies name test *n*. Moreover, ψ is true at the lastly reached node, or equivalently, the first node in document order, in that block. (We may reuse the example in Figure 4 for `anc` by interpreting the left arrow in the figure as a sequence of $\langle\bar{2}\rangle$ followed by one $\langle\bar{1}\rangle$.) In the rule ANC, we thus set the with parameter to $\neg\,\texttt{has-anc}(k(n))$ to mean that there is no (more) ancestor satisfying name test n. $\texttt{has-anc}(\chi)$ is a formula that describes any tree node such that it has at least one ancestor at which χ is true and $\neg\,\texttt{has-anc}(\chi)$ is its negation.[2] Note that $\langle\bar{2}\rangle$ denotes the left sibling of the context node if any, and $\langle\bar{1}\rangle$ its parent if the context node has no left sibling and is not a root.

When the output type is (φ, u) and the parameter is ψ, it means that the context node has an ancestor t that satisfies name test *n* and is of type (φ, u) (rule ANC-FORMULA). Moreover, ψ should be also true at t. The invariant here is that ψ describes the structure of the ancestors of t. The inferred input formula φ' is thus a recursive formula that denotes a tree node having an ancestor t satisfying all these constraints, *i.e.,* $\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \psi$. Furthermore, the ancestors between t and the context node should not satisfy name test *n* and thus have type $\neg k(n) \wedge X$, which is also encoded in the inferred input formula φ'.

3.2.3 Next Sibling

`nsibl` is the converse of `psibl`. To obtain inference rules for `nsibl`, we just replace $\langle\bar{2}\rangle$ and $[\bar{2}]$ in the rules PSIBL and PSIBL-FORMULA with $\langle 2\rangle$ and $[2]$, respectively, and use rule AXIS-FORWARD-SEQ instead of rule AXIS-BACKWARD-SEQ. More precisely, given a judgment $\varphi \leftarrow \texttt{nsibl::}n,\ \rho$ with ψ, the invariant is that there is a block of nodes reached by navigating `nsibl` from the context node that produces a sequence of nodes of type ρ, each of which satisfies name test *n*. Moreover, the lastly reached node in that block and the context node satisfy ψ and φ, respectively. Since our analysis always starts with the lastly reached node, *i.e.,* the rightmost sibling in the case of `nsibl`, in the rule NSIBL, we set the initial with parameter to $\mu X.\,[2]\,(\neg k(n) \wedge X)$ which means that there is no (more) next sibling satisfying name test *n*. For the unit type part, we simply use AnyElt because of the lack of information about the context in the regular tree types of the next sibling nodes.

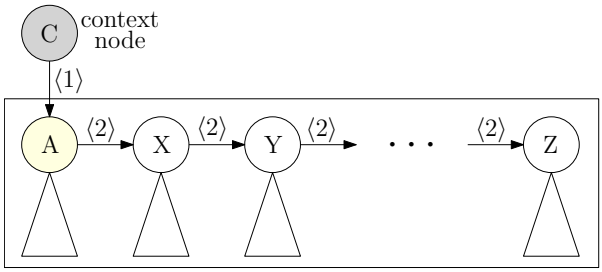
If the output type is a single formula type (φ, u) and the parameter is ψ, it means that the context node has a next sibling t that satisfies name test *n* and is of type (φ, u) (rule NSIBL-FORMULA). Moreover, ψ should be also true at t. The invariant here is that ψ describes the structure of the next siblings of t. In addition, the next siblings between the context node and t, if any, should not satisfy name test *n* and thus have type $\neg k(n) \wedge \langle 2\rangle\, X$. All these constraints are encoded in the inferred input formula φ'.

Lastly, from the rules in Figures 6 and 11, we can conclude that the input formula inferred for `nsibl` always begins with either $[2]$ or $\langle 2\rangle$ and thus describes the structure of the next sibling of the context node.

3.2.4 Child

As inference rules for `parent` are defined in terms of those for `self`, rules for `child` can be defined in terms of those for `self-nsibl` (self or next sibling, a variant of `nsibl`, defined in the next subsection). As shown in Figure 12, we first infer a formula φ for `self-nsibl` and then use it as a constraint for the leftmost child of the context node by adding either $[1]$ or $\langle 1\rangle$ to φ. Specifically, if the output type is nullable, which means that

[2]Technically this encoding allows the presence of hedges satisfying the formula (we do not impose the invariant that there is only a single root), but our semantics ensures that a formula accepts trees only.



The nodes in the box are C's children and node A is its leftmost child. Therefore, $[\![\mathrm{C}/\texttt{child::}n]\!] = [\![\mathrm{A}/\texttt{self-nsibl::}n]\!]$. If $\varphi \leftarrow \texttt{self-nsibl::}n$, ρ and φ is true at node A, then $\langle 1\rangle\,\varphi$ is true at node C.

CHILD-NULLABLE

$$\frac{\varphi \leftarrow \texttt{self-nsibl::}n,\ \rho \qquad \mathit{nullable}(\rho)}{([1]\,\varphi,\ \texttt{parent-type}(\rho)) \leftarrow \texttt{child::}n,\ \rho}$$

CHILD-NOTNULL

$$\frac{\varphi \leftarrow \texttt{self-nsibl::}n,\ \rho \qquad \neg\mathit{nullable}(\rho)}{(\langle 1\rangle\,\varphi,\ \texttt{parent-type}(\rho)) \leftarrow \texttt{child::}n,\ \rho}$$

$$\begin{aligned}
\texttt{parent-type}(\rho) &= \texttt{element} * \{\, \mathsf{AnyElt}^*,\ \texttt{add-anyelt}(\rho),\ \mathsf{AnyElt}^* \,\}\\
\texttt{add-anyelt}(\texttt{()}) &= \texttt{()}\\
\texttt{add-anyelt}((\varphi, u)) &= u\\
\texttt{add-anyelt}(\rho_1 \mid \rho_2) &= \texttt{add-anyelt}(\rho_1) \mid \texttt{add-anyelt}(\rho_2)\\
\texttt{add-anyelt}(\rho_1, \rho_2) &= \texttt{add-anyelt}(\rho_1),\ \mathsf{AnyElt}^*,\ \texttt{add-anyelt}(\rho_2)\\
\texttt{add-anyelt}(\rho^+) &= (\mathsf{AnyElt}^*,\ \texttt{add-anyelt}(\rho))^+
\end{aligned}$$

Figure 12: Inference rules for `child`

the context node may not have a child, then we use universal modality (rule CHILD-NULLABLE). Otherwise, the context node always has a child and hence we use existential modality instead (rule CHILD-NOTNULL).

In addition, to infer a unit type for the context node, we use an auxiliary function $\texttt{parent-type}(\rho)$, defined in Figure 12, which computes the type of any node that has some children of type ρ and possibly more of arbitrary types. To this end, it exploits another auxiliary function $\texttt{add-anyelt}(\rho)$ which extracts all unit types at top level of ρ, while maintaining their order, and adds AnyElt^* between unit types, indicating that there may be more child nodes. Note that $\texttt{parent-type}(\rho)$ approximates the type of the context node. For example, consider the tree in Figure 12. If only nodes A and Y are returned by $\mathrm{C}/\texttt{child::}n$, then other nodes such as X and Z must not satisfy name test *n*. This constraint is described in the inferred formula, as discussed in the next subsection, but not in the inferred unit type. If we add negation of a name test, *i.e.,* $\neg n$, we could infer a more precise unit type. However, since all the constraints are encoded in the inferred formula and thus our inference system for XPath axes is exact, we do not add $\neg n$ in the definition of regular tree types. Still, the more precise we infer a unit type, more precise we can develop an inference system for XQuery in Section 4, and thus we do not simply use AnyElt for the unit type of the context node.

3.2.5 Self or Next Sibling

While inference rules for `self-nsibl` are similar to those for `nsibl`, there is a key difference. Suppose `nsibl::`*n* returns nothing (*i.e.,* the output type is `()`). Then it means that there is no next sibling satisfying name test *n*, and thus the input tree should have type $\mu X.\,[2]\,(\neg k(n) \wedge X)$ (either there is no next sibling or if any, it does not satisfy *n*). In contrast, if `self-nsibl::`*n* returns nothing, it means that the context node does not satisfy *n* and neither do its next siblings, *i.e.,* $\mu X.(\neg k(n) \wedge [2]\,X)$.

This difference leads to two interpretations of the output type depending on whether it is nullable or not. To illustrate, assume that the output type is $(\varphi, u), \rho$. As in the inference rules for `nsibl`, we examine it from right to left. Suppose that a formula ψ is inferred from ρ and that there exists a node t satisfying (φ, u) (t can be either the context node or its next sibling). Then, ψ is a constraint on t's next sibling. More precisely, if ρ is nullable, then the exact constraint on t is $\varphi \wedge [2]\,\psi$ indicating that it may not have a next sibling. Otherwise, the exact

$$\mathsf{nullable} ::= \mathsf{true} \mid \mathsf{false}$$

SELF-NSIBL
$$\frac{\varphi \leftarrow \texttt{self-nsibl::}n,\ \rho \text{ with } \mu X.(\neg k(n) \wedge [2]\, X),\ \mathsf{true}}{\varphi \leftarrow \texttt{self-nsibl::}n,\ \rho}$$

SNSIBL-TRUE
$$\frac{\varphi' = \mu X.(\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge [2]\, \psi) \vee (\neg k(n) \wedge \langle 2 \rangle\, X)}{\varphi' \leftarrow \texttt{self-nsibl::}n,\ (\varphi, u) \text{ with } \psi,\ \mathsf{true}}$$

SNSIBL-FALSE
$$\frac{\varphi' = \mu X.(\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \langle 2 \rangle\, \psi) \vee (\neg k(n) \wedge \langle 2 \rangle\, X)}{\varphi' \leftarrow \texttt{self-nsibl::}n,\ (\varphi, u) \text{ with } \psi,\ \mathsf{false}}$$

AXIS-FORWARD-SEQ
$$\frac{\varphi_2 \leftarrow \textit{axis}\texttt{::}n,\ \rho_2 \text{ with } \psi,\ \mathsf{nullable} \qquad \varphi_1 \leftarrow \textit{axis}\texttt{::}n,\ \rho_1 \text{ with } \varphi_2,\ \mathsf{nullable} \wedge \textit{nullable}(\rho_2)}{\varphi_1 \leftarrow \textit{axis}\texttt{::}n,\ (\rho_1, \rho_2) \text{ with } \psi,\ \mathsf{nullable}}$$

Figure 13: Inference rules for `self-nsibl`

constraint on t is $\varphi \wedge \langle 2 \rangle\, \psi$ indicating that t's next sibling exists and it has type ψ. In other words, given an output type (ρ_1, ρ_2), when examining ρ_1, we need to exploit the nullability of ρ_2.

To this end, we introduce a new judgment $\varphi \leftarrow \texttt{self-nsibl::}n,\ \rho$ with ψ, nullable where nullable denotes either true or false. In this judgment, the meaning of ψ is twofold: it denotes the constraint on either the context node, *i.e.,* `self`, or its next sibling *i.e.,* `nsibl`. The former is when ρ is `()`. Otherwise, ψ is true at the lastly reached next sibling node among the sequence of nodes that is returned by `self-nsibl::`n and that is of type ρ. In the rule SELF-NSIBL, therefore, the initial with parameter is set to $\mu X.(\neg k(n) \wedge [2]\, X)$ meaning that all the next siblings (including the context node if the given output type is `()`) do not satisfy name test n. The nullability is set to true since $\rho \equiv \rho, \texttt{()}$.

The nullability parameter is examined only when the output type is a single formula type (φ, u). Consider a judgment $\varphi' \leftarrow \texttt{self-nsibl::}n,\ (\varphi, u)$ with ψ, nullable. Then, there should be a node t_1 that satisfies name test n and is of type (φ, u) (it can be either the context node or its one of next siblings). Moreover, t_1's next sibling t_2 must have type ψ. If nullable is true, then t_2 may not exist and thus t_1 has type $\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge [2]\, \psi$ (rule SNSIBL-TRUE). Otherwise, t_2 must exist and thus t_1 has type $\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \langle 2 \rangle\, \psi$ (rule SNSIBL-FALSE).

For the rest of the cases, we reuse the inference rules in Figure 6 with minor modifications. For the rules AXIS-EMPTY, AXIS-OR, and AXIS-PLUS, we add one more parameter nullable in each judgment. The nullability is updated when examining the first type of the given sequence type (ρ_1, ρ_2) as shown in the modified rule AXIS-FORWARD-SEQ in Figure 13. Precisely, the last node t among the nodes returned by `self-nsibl::`n with output type ρ_1 may not have a next sibling if ρ_2 is nullable and the given parameter nullable is true. In this case, we use $[2]\, \varphi_2$ as a constraint on t (in combination with the rule SNSIBL-TRUE). Otherwise, t must have a next sibling and we use $\langle 2 \rangle\, \varphi_2$ as a constraint on t (in combination with the rule SNSIBL-FALSE).

3.2.6 Descendant

Like other axes, we use a judgment of the form $\varphi \leftarrow \texttt{desc::}n,\ \rho$ with ψ, but the with parameter ψ now denotes a constraint on the last node in document order among the descendants returned by `desc::`n with output type ρ.

In the rule DESC in Figure 14, the initial with parameter is much more complicated than other axes because we need to specify constraints only on the descendants of the context node, but not on others. More precisely, if the output type is a sequence type of the form $(\varphi_1, u_1), \ldots, (\varphi_n, u_n)$, then `desc::`$n$ returns a sequence $t_1, \ldots, t_n$ of descendants in document order, each of which has type (φ_i, u_i). Now, to infer the exact input type, we need to specify that all the nodes that follows t_n but precedes the leftmost next sibling of the context node must not satisfy name test n. This constraint is encoded in $\texttt{noNextUpTo}(k(n), \alpha)$ which exploits a *nominal*, denoted by a fresh variable α, *i.e.,* an atomic proposition that holds only on the context node on which `desc` is applied. The nominal is then used as a search bound for a descendant during the inference process and its property is ensured

$$\textsc{Desc}\quad \frac{\varphi \leftarrow \texttt{desc::}n,\ \rho \text{ with } \texttt{noNextUpTo}(k(n), \alpha)}{(\varphi \wedge \texttt{noWhereElse}(\alpha),\ \mathsf{AnyElt}) \leftarrow \texttt{desc::}n,\ \rho}\ (\alpha \text{ fresh})$$

$$\textsc{Desc-Formula}\quad \frac{\varphi' = \texttt{fstDescFoll}(\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \psi,\ k(n))}{\varphi' \leftarrow \texttt{desc::}n,\ (\varphi, u) \text{ with } \psi}$$

Auxiliary definitions:

$$\begin{aligned}
\chi\ ?\ \psi_1 : \psi_2 &\equiv (\chi \wedge \psi_1) \vee (\neg\chi \wedge \psi_2)\\
\texttt{has-desc}(\chi) &= \langle 1\rangle\, (\mu Z.\chi \vee \langle 1\rangle\, Z \vee \langle 2\rangle\, Z)\\
\texttt{has-nsdesc}(\chi) &= \langle 2\rangle\, (\mu Z.\chi \vee \langle 1\rangle\, Z \vee \langle 2\rangle\, Z)\\
\texttt{has-prec}(\chi) &= \mu Z.\, \langle \bar{1}\rangle\, Z \vee \langle \bar{2}\rangle\, (\chi \vee \texttt{has-desc}(\chi) \vee Z)\\
\texttt{has-foll}(\chi) &= \mu Z.\, \texttt{has-nsdesc}(\chi) \vee \texttt{has-parent}(Z)\\
\texttt{noNextUpTo}(\chi, \alpha) &= \neg\, \texttt{has-desc}(\chi)\ \wedge\ \mu Z.\, \alpha\ ?\ \top : (\neg\texttt{has-nsdesc}(\chi) \wedge \texttt{has-parent}(Z))\\
\texttt{noWhereElse}(\chi) &= \chi \wedge \neg\, (\texttt{has-anc}(\chi) \vee \texttt{has-}\mathbf{\texttt{prec}}(\chi) \vee \texttt{has-desc}(\chi) \vee \texttt{has-}\mathbf{\texttt{foll}}(\chi))\\
\mathbf{\texttt{fstSelfNsDesc}}(\chi_1, \chi_2) &= \mu Z.\, \chi_1 \vee (\neg\chi_2 \wedge (\texttt{has-desc}(\chi_2)\ ?\ \langle 1\rangle\, Z : \langle 2\rangle\, Z))\\
\mathbf{\texttt{fstFoll}}(\chi_1, \chi_2) &= \mu Z.\, \langle 2\rangle\, \mathbf{\texttt{fstSelfNsDesc}}(\chi_1, \chi_2) \vee (\neg\texttt{has-}\mathbf{\texttt{nsdesc}}(\chi_2) \wedge \texttt{has-parent}(Z))\\
\texttt{fstDescFoll}(\chi_1, \chi_2) &= \langle 1\rangle\, \mathbf{\texttt{fstSelfNsDesc}}(\chi_1, \chi_2) \vee (\neg\texttt{has-desc}(\chi_2) \wedge \mathbf{\texttt{fstFoll}}(\chi_1, \chi_2))
\end{aligned}$$

- $\texttt{has-desc}(\chi)$: there is a descendant satisfying χ.
- $\texttt{has-nsdesc}(\chi)$: there is a node satisfying χ which is a next sibling or a descendant of a next sibling.
- $\texttt{has-}\mathbf{\texttt{prec}}(\chi)$: there is a node satisfying χ which precedes the context node in document order and is not an ancestor.
- $\texttt{has-}\mathbf{\texttt{foll}}(\chi)$: there is a node satisfying χ which follows the context node in document order and is not a descendant.
- $\texttt{noNextUpTo}(\chi, \alpha)$: there is no node satisfying χ which appears strictly after the context node in document order and before a node satisfying α (invariant: α should denote a nominal).
- $\texttt{noWhereElse}(\chi)$: only the context node satisfies χ.
- $\mathbf{\texttt{fstSelfNsDesc}}(\chi_1, \chi_2)$: the first node t in document order of the set {self, all next siblings, and all their descendants} that satisfies χ_1. Any node preceding t in the set does not satisfy χ_2.
- $\mathbf{\texttt{fstFoll}}(\chi_1, \chi_2)$: the first node t satisfying χ_1 among the nodes reachable by navigating `following`. Any node between the context node and t reached by navigating `following` does not satisfy χ_2.
- $\texttt{fstDescFoll}(\chi_1, \chi_2)$: the first node t satisfying χ_1 that appears strictly after the context node in document order. Any node between the context node and t does not satisfy χ_2.

Figure 14: Inference rules for `desc`

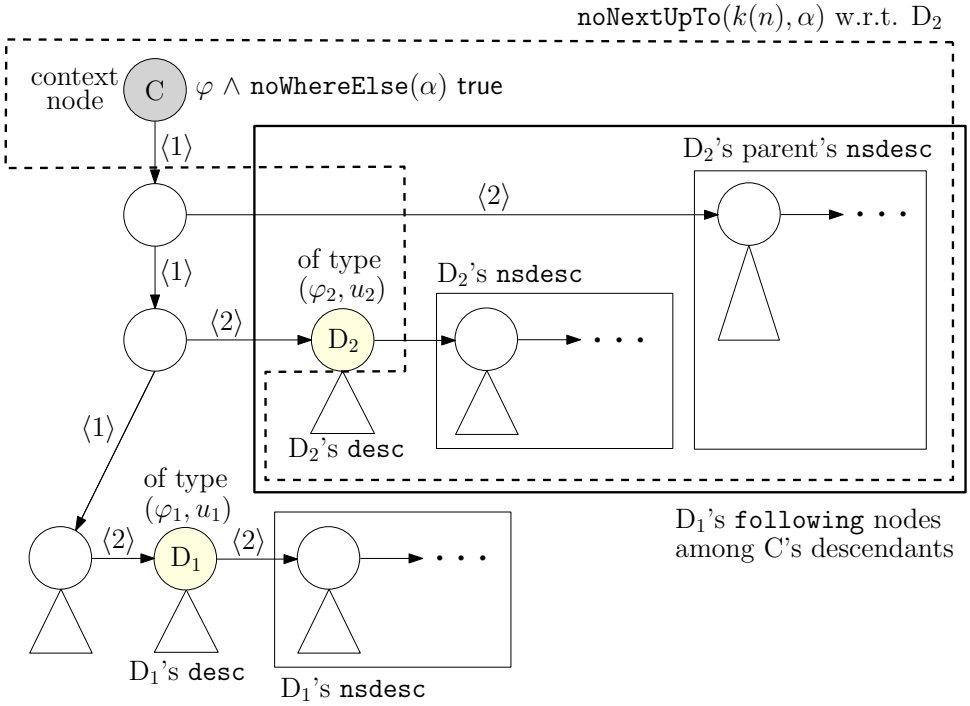


Figure 15: $\varphi \leftarrow$ desc::n, $((\varphi_1, u_1), (\varphi_2, u_2))$ with noNextUpTo$(k(n), \alpha)$ where C is the context node and D_1 and D_2 are the only nodes satisfying name test n, each of which has type (φ_1, u_1) and (φ_2, u_2), respectively. D_1 precedes D_2 in document order.

by noWhereElse(α) in the final input type.

If the output type is a single formula type (φ, u) and the with parameter is ψ, it means that the context node has a descendant t of type (φ, u) that satisfies name test n and at which ψ is true (rule DESC-FORMULA). Moreover, any node between the context node and t in document order should not satisfy name test n. All these constraints are encoded in fstDescFoll$(\varphi \wedge k(n) \wedge \text{form}(u) \wedge \psi,\ k(n))$, which is defined in Figure 14. As for other cases, we simply use the inference rules in Figure 6.

To illustrate, consider an example tree in Figure 15. Suppose that the output type is $(\varphi_1, u_1), (\varphi_2, u_2)$. Then there exist only two descendants satisfying name test n, namely, D_1 and D_2. According to the rule AXIS-FORWARD-SEQ, we first analyze the rightmost output type (φ_2, u_2). In other words, we first infer a constraint on the node D_2. Since D_2 is the last node returned by desc::n, noNextUpTo$(k(n), \alpha)$ should be true at D_2 which means that $k(n)$ is not true at D_2's descendants, its next siblings and their descendants, its parent's next siblings and their descendants, its parent's parent's next siblings and their descendants, and so on until the initial context node C, marked with a nominal α, is reached (rule DESC). Moreover, when locally analyzing D_2 with the output type (φ_2, u_2), the context node is D_1. From D_1's perspective, D_2 is the first node satisfying name test n among D_1's descendants and following nodes. This constraint is expressed by using the function fstDescFoll(χ_1, χ_2) (rule DESC-FORMULA).

3.3 Properties of the Type Inference System

In this section, we briefly discuss the soundness and completeness of our backward type inference system for XPath axes. In other words, our backward inference is exact.

Theorem 3.1 (Exact type inference)**.** *Suppose* $\rho_i \leftarrow$ *axis::*n, ρ_o. *Then* $f \in [\![\rho_i]\!]$ *if and only if* $[\![f/\textit{axis::}\text{n}]\!] \in [\![\rho_o]\!]$.

In Theorem 3.1, the only-if-direction states the soundness and the if-direction states the completeness. More precisely, the soundness states that if some focused tree has the inferred input type ρ_i, then it always produces a sequence of nodes of output type ρ_o. In contrast, the completeness states the opposite, that is, if some focused

tree produces a sequence of nodes of type ρ_o, then it has the inferred type ρ_i. To prove Theorem 3.1, we use the following lemmas for the auxiliary judgment $\varphi \leftarrow \mathit{axis}::n,\ \rho$ with ψ.

Lemma 3.2 (Soundness)**.** *Suppose*

- $\varphi \leftarrow \mathit{axis}::\mathrm{n},\ \rho$ with ψ*,*
- $f \in \langle\!\langle \varphi \rangle\!\rangle$*, and*
- $[\![f/\mathit{axis}::\mathrm{n}]\!] = f_1, \ldots, f_n$.

If axis *is a backward axis:*

- *Let* $f = f_{n+1}$.
- *Then,* $\exists\ 1 \leq i \leq n+1$ *s.t.* $f_i \in \langle\!\langle \psi \rangle\!\rangle$ *and* $f_i, \ldots f_n \in [\![\rho]\!]$.

Otherwise, axis *is a forward axis:*

- *Let* $f = f_0$.
- *Then,* $\exists\ 0 \leq i \leq n$ *s.t.* $f_i \in \langle\!\langle \psi \rangle\!\rangle$ *and* $f_1, \ldots f_i \in [\![\rho]\!]$.

Lemma 3.2 is simply a one-way formalization of the interpretation of the judgment $\varphi \leftarrow \mathit{axis}::n,\ \rho$ with ψ. To illustrate, consider Figure 4 again. In the figure, $f_{n+1} = \mathrm{C}$ and $f_i = \mathrm{A}$ where $\mathrm{C} \in \langle\!\langle \varphi \rangle\!\rangle$ and $\mathrm{A} \in \langle\!\langle \psi \rangle\!\rangle$. Moreover, the sequence $f_i, \ldots, f_n$ of nodes selected by `psibl::`n from node A to node B has type ρ. Below we show some cases of the proof of Lemma 3.2.

of Lemma 3.2. By induction on a derivation of $\varphi \leftarrow \mathit{axis}::n,\ \rho$ with ψ.
If *axis* is a backward axis:
Case 1) $\rho =$ `()`:

1. $\varphi = \psi$ — from the rule AXIS-EMPTY
2. Let $i = n + 1$.
3. Then, $f_i \in \langle\!\langle \psi \rangle\!\rangle$ and $f_i, \ldots, f_n = \epsilon \in [\![$ `()` $]\!]$. — from assumptions

Case 2) $\rho = \rho_1, \rho_2$:

1. $\varphi_1 \leftarrow \mathit{axis}::n,\ \rho_1$ with ψ — from the rule AXIS-BACKWARD-AXIS
2. $\varphi \leftarrow \mathit{axis}::n,\ \rho_2$ with φ_1 — from the rule AXIS-BACKWARD-AXIS
3. $\exists\ 1 \leq j \leq n+1$ *s.t.* $f_j \in \langle\!\langle \varphi_1 \rangle\!\rangle$ and $f_j, \ldots, f_n \in [\![\rho_2]\!]$ — by I.H. on (2)
4. $[\![f_j/\mathit{axis}::n]\!] = f_1, \ldots, f_{j-1}$
5. $\exists\ 1 \leq k \leq j$ *s.t.* $f_k \in \langle\!\langle \psi \rangle\!\rangle$ and $f_k, \ldots, f_{j-1} \in [\![\rho_1]\!]$ — by I.H. on (1) and (4)
6. Let $i = k$.
7. Then, $f_i \in \langle\!\langle \psi \rangle\!\rangle$ and $f_i, \ldots, f_{j-1}, f_j, \ldots, f_n \in [\![(\rho_1, \rho_2)]\!]$.

□

Below we state the completeness lemma for the auxiliary judgment.

Lemma 3.3 (Completeness)**.** *Suppose*

- $\varphi \leftarrow \mathit{axis}::\mathrm{n},\ \rho$ with ψ *and*
- $[\![f/\mathit{axis}::\mathrm{n}]\!] = f_1, \ldots, f_n$.

If axis *is a backward axis:*

- *Let* $f = f_{n+1}$.

- *Suppose* $\exists\, 1 \leq i \leq n+1$ *s.t.* $f_i \in \langle\!\langle \psi \rangle\!\rangle$ *and* $f_i, \ldots f_n \in [\![\rho]\!]$.
- *Then,* $f \in \langle\!\langle \varphi \rangle\!\rangle$.

Otherwise, axis *is a forward axis:*

- *Let* $f = f_0$.
- *Suppose* $\exists\, 0 \leq i \leq n$ *s.t.* $f_i \in \langle\!\langle \psi \rangle\!\rangle$ *and* $f_1, \ldots f_i \in [\![\rho]\!]$.
- *Then,* $f \in \langle\!\langle \varphi \rangle\!\rangle$.

Proof. By induction on a derivation of $\varphi \leftarrow axis\texttt{::}n,\ \rho$ with ψ. Here, we only show the case where $\rho = \rho_1, \rho_2$ and *axis* is a backward axis. Below, the proof first analyze (1) and then (2), which is the opposite to the proof of Lemma 3.2

1. $\varphi_1 \leftarrow axis\texttt{::}n,\ \rho_1$ with ψ — from the rule AXIS-BACKWARD-AXIS
2. $\varphi \leftarrow axis\texttt{::}n,\ \rho_2$ with φ_1 — from the rule AXIS-BACKWARD-AXIS
3. $\exists\, i \leq j \leq n$ *s.t.* $f_i, \ldots, f_{j-1} \in [\![\rho_1]\!]$ and $f_j, \ldots, f_n \in [\![\rho_2]\!]$ — from $f_i, \ldots f_n \in [\![(\rho_1, \rho_2)]\!]$
4. $[\![f_j/axis\texttt{::}n]\!] = f_1, \ldots, f_{j-1}$
5. $1 \leq i \leq j$ and $f_i \in \langle\!\langle \psi \rangle\!\rangle$ — from (3) and assumptions
6. $f_j \in \langle\!\langle \varphi_1 \rangle\!\rangle$ — by I.H. on (1), (3), (4), (5)
7. $f_{n+1} \in \langle\!\langle \varphi \rangle\!\rangle$ — by I.H. on (2), (3), (6)

□

4 Inference for XQuery Core

In this section, we present our backward type inference system for the XQuery core, building on the results of the previous section. We first clarify what we infer from the given expression *e* and output type ρ. Precisely, we use a judgment of the form $\mathscr{S} \leftarrow e : \rho$ which means that given an expression *e* and an output type ρ, it generates a set $\mathscr{S}$ of constraint-sets for free variables in *e*. Our goal is then to design inference rules that ensure that if we substitute those free variables with any set of focused trees satisfying one of constraint-sets in $\mathscr{S}$, *e* evaluates to a value, *i.e.,* a sequence of focused trees, that has the type ρ. By convention, if $\mathscr{S}$ is an empty set, it is unsatisfiable, and we denote it by $\mathbb{0}$. In contrast, a singleton set consisting of an empty set is always satisfiable, and we denote it by $\mathbb{1}$.

Formally, a constraint-set C is a set of bindings of variables with formula-enriched sequence types, where each binding is denoted by $(\$var : \rho)$. Given a constraint-set C, we consider any for-loop and let-bound variables not appearing in C to be implicitly bound to $(\top, \mathsf{AnyElt})$ and $(\top, \mathsf{AnyElt})^*$, respectively. Moreover, a constraint-set C is unsolvable if it contains a constraint specifying that a variable should satisfy $\bot$, *e.g.,* $(\$var : (\bot, u))$. We simply write $\{\bot\}$ to denote such an unsolvable constraint-set. If $\mathscr{S}$ contains $\{\bot\}$, we can safely remove it from $\mathscr{S}$. We often consider a constraint-set C to be a mapping from variables to their types and hence use the usual notations such as:

$$
\begin{array}{rcll}
\mathrm{dom}(C) & \stackrel{\text{def}}{=} & \{\$var \mid (\$var : \rho) \in C\} & \\
C(\$var) & \stackrel{\text{def}}{=} & \rho & \text{if } (\$var : \rho) \in C \\
C(\$v) & \stackrel{\text{def}}{=} & (\top, \mathsf{AnyElt}) & \text{if } \$v \notin \mathrm{dom}(C) \\
C(\$\overline{v}) & \stackrel{\text{def}}{=} & (\top, \mathsf{AnyElt})^* & \text{if } \$\overline{v} \notin \mathrm{dom}(C)
\end{array}
$$

We also introduce the following operations.

Definition 4.1. *Let C_1 and C_2 be constraint-sets, which are not $\{\bot\}$, and $\mathscr{S}$, $\mathscr{S}_1$, and $\mathscr{S}_2$ be sets of constraint-sets. We define:*

$$
\begin{aligned}
C_1 \sqcap C_2 &\stackrel{def}{=} \{(\$var : \rho) \in C_1 \mid \$var \notin \mathrm{dom}(C_2)\} \cup \\
&\quad \{(\$var : \rho) \in C_2 \mid \$var \notin \mathrm{dom}(C_1)\} \cup \\
&\quad \{(\$var : \rho_1 \wedge \rho_2) \mid (\$var : \rho_1) \in C_1 \textit{ and } (\$var : \rho_2) \in C_2\} \\
C\backslash_{\$var_0} &\stackrel{def}{=} \{(\$var : \rho) \in C \mid \$var \neq \$var_0\} \\
\mathscr{S}_1 \sqcap \mathscr{S}_2 &\stackrel{def}{=} \{C_1 \sqcap C_2 \mid C_1 \in \mathscr{S}_1,\ C_2 \in \mathscr{S}_2\} \\
\mathscr{S}_1 \sqcup \mathscr{S}_2 &\stackrel{def}{=} \mathscr{S}_1 \cup \mathscr{S}_2 \\
\mathscr{S}\backslash_{\$var} &\stackrel{def}{=} \{C\backslash\$var \mid C \in \mathscr{S}\}
\end{aligned}
$$

For any constraint-set C, $C \sqcap \{\bot\} = \{\bot\} \sqcap C = \{\bot\}$.

In the definition above, we use $\rho_1 \wedge \rho_2$ to denote the intersection of ρ_1 and ρ_2 whose semantics $[\![\rho_1 \wedge \rho_2]\!]$ is inductively defined as $[\![\rho_1]\!] \cap [\![\rho_2]\!]$. In other words, for any focused tree f, $f \in [\![\rho_1 \wedge \rho_2]\!]$ if and only if $f \in [\![\rho_1]\!]$ and $f \in [\![\rho_2]\!]$. Although we use intersection types only internally during type inference, they can be seamlessly added into the external language Frisch et al. (2008).

4.1 Inference Rules

Figures 16 and 17 show our backward type inference rules for XQuery core. We first describe the case where the output type is a union type $\rho_1 \mid \rho_2$ (rule I-OR). In this case, the input constraint is a union of $\mathscr{S}_1$ and $\mathscr{S}_2$, which are inferred from ρ_1 and ρ_2, respectively. If one of $\mathscr{S}_i$ is unsatisfiable, *i.e.,* $\mathbb{0}$, it is simply ignored since $\mathscr{S}_j \sqcup \mathbb{0} = \mathscr{S}_j$ where $i \neq j$. If both $\mathscr{S}_1$ and $\mathscr{S}_2$ are unsatisfiable, the input constraint is $\mathbb{0}$ which means that expression *e* can never have the output type $\rho_1 \mid \rho_2$ in the first place. Similarly, if the output type is an intersection type $\rho_1 \wedge \rho_2$, the input constraint is an intersection of $\mathscr{S}_1$ and $\mathscr{S}_2$, each of which is inferred from ρ_i (rule I-AND). In this case, if one of $\mathscr{S}_i$ is unsatisfiable, then the input type is also unsatisfiable. During the inference, either rule I-OR or rule I-AND should be first tried.

Rules I-EMP, I-FVAR, I-LVAR, and I-AXIS are relatively easy. First, in the rule I-EMP, if the output type ρ is nullable, then the input constraint is $\mathbb{1}$ which means that ϵ is of type ρ without further constraints. In the rule I-FVAR, we use the inference rules for the `self` axis since a for-loop variable is bound only to an XML element, not a sequence. In contrast, rule I-LVAR just binds a let-bound variable to the given sequence type since it can be bound to an arbitrary sequence. Rule I-AXIS uses inference rules for the axis expression, and binds the for-loop variable to the inferred type.

In the rule I-ELEMENT, we consider only a type-annotated element constructor of the form (<σ>{*e*}</σ> : u). The annotated type u should be a subtype of the output type ρ since we are using a backward type inference. Specifically, since an element constructor always reduces to a root element, we check the subtype relation $(\varphi_{root}, u) <: \rho$ where formula $\varphi_{root} = \neg\langle\bar{1}\rangle\top \wedge \neg\langle\bar{2}\rangle\top \wedge \neg\langle 2\rangle\top$ specifies that the given node is a root (the subtype relation is explained shortly). Let u be `element` *n* $\{\tau\}$. Then, node label σ should match name test *n*, denoted by $\sigma \leq n$. Finally, we infer input constraints from the body expression *e* that reduces to a sequence of child nodes and from the output type $\mathrm{form\text{-}enriched}(\tau)$ for the child nodes, where we use an auxiliary function $\mathrm{form\text{-}enriched}(\tau)$ to support context-erasing element construction: during the reduction, we remove the context of the result of *e* (see Figure 2). $\mathrm{form\text{-}enriched}(\tau)$ enriches the given regular tree type τ by simply associating each unit type u that appears in τ with an equivalent downward-only formula $\mathrm{form}(u)$, *i.e.,* without context information.

To check the subtype relation $(\varphi_{root}, u) <: \rho$, we first compute the type ρ' for the set of all single focused tree nodes that are contained in $[\![\rho]\!]$. Then $(\varphi_{root}, u) <: \rho$ if $(\varphi_{root}, u) <: \rho'$ because (φ_{root}, u) denotes a set of focused tree nodes. Next, we translate u and ρ' into equivalent formulas φ and ψ, respectively, and then test the satisfiability of $\varphi_{root} \wedge \varphi \wedge \neg\psi$. To this end, we use an auxiliary function $\mathrm{single}(\rho)$ which computes a formula whose denotation includes only singleton sequences of focused tree nodes contained in $[\![\rho]\!]$. That is, $(\varphi_{root}, u) <: \rho$ if and only if $\langle\!\langle \varphi_{root} \wedge \mathrm{form}(u) \wedge \neg\mathrm{single}(\rho) \rangle\!\rangle = \emptyset$ which can be tested in $2^{O(|u|+|\rho|)}$ time by the decision procedure in Genevès and Gesbert (2015).

For if-expressions, we consider three cases to infer an input constraint that is as precise as possible. In order to gain more precision, we should first try either rule I-IFNONEMPTY or rule I-IFEMPTY. Rule I-IFANY is the most general rule and infers a constraint $\mathscr{S}_i$ from each subexpression e_i, which minimally assumes that the condition

$$\text{I-OR}\quad \frac{\mathscr{S}_i \leftarrow e : \rho_i}{\mathscr{S}_1 \sqcup \mathscr{S}_2 \leftarrow e : \rho_1 \mid \rho_2}\ (i = 1,2) \qquad \text{I-AND}\quad \frac{\mathscr{S}_i \leftarrow e : \rho_i}{\mathscr{S}_1 \sqcap \mathscr{S}_2 \leftarrow e : \rho_1 \wedge \rho_2}\ (i = 1,2) \qquad \text{I-EMP}\quad \frac{\mathit{nullable}(\rho)}{\mathbb{1} \leftarrow \epsilon : \rho}$$

$$\text{I-FVAR}\quad \frac{\rho' \leftarrow \texttt{self::}*,\ \rho}{\{\{(\$v : \rho')\}\} \leftarrow \$v : \rho} \qquad \text{I-LVAR}\quad \frac{}{\{\{(\$\overline{v} : \rho)\}\} \leftarrow \$\overline{v} : \rho} \qquad \text{I-AXIS}\quad \frac{\rho' \leftarrow \mathit{axis}\texttt{::}n,\ \rho}{\{\{(\$v : \rho')\}\} \leftarrow \$v/\mathit{axis}::n : \rho}$$

$$\text{I-ELEMENT}\quad \frac{(\varphi_{\mathit{root}}, u) <: \rho \qquad u = \texttt{element}\ n\ \{\tau\} \qquad \sigma \leq n \qquad \mathscr{S} \leftarrow e : \text{form-enriched}(\tau)}{\mathscr{S} \leftarrow (\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u) : \rho}$$

$$\text{I-IFNONEMPTY}\quad \frac{\mathscr{S}_1 \leftarrow e_1 : (\top, \mathsf{AnyElt})^+ \quad \mathscr{S}_2 \leftarrow e_2 : \rho}{\mathscr{S}_1 \sqcap \mathscr{S}_2 \leftarrow \texttt{if nempty}(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \rho} \qquad \text{I-IFEMPTY}\quad \frac{\mathscr{S}_1 \leftarrow e_1 : \texttt{()} \quad \mathscr{S}_3 \leftarrow e_3 : \rho}{\mathscr{S}_1 \sqcap \mathscr{S}_3 \leftarrow \texttt{if nempty}(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \rho}$$

$$\text{I-IFANY}\quad \frac{\mathscr{S}_1 \leftarrow e_1 : (\top, \mathsf{AnyElt})^* \quad \mathscr{S}_2 \leftarrow e_2 : \rho \quad \mathscr{S}_3 \leftarrow e_3 : \rho}{\mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \mathscr{S}_3 \leftarrow \texttt{if nempty}(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \rho}$$

$$\text{I-LET}\quad \frac{\mathscr{S}_2 \leftarrow e_2 : \rho \quad S = \{\mathscr{S}_1 \sqcap \{C\backslash_{\$\overline{v}}\} \mid \mathscr{S}_1 \leftarrow e_1 : C(\$\overline{v}),\ C \in \mathscr{S}_2\}}{\bigsqcup_{\mathscr{S} \in S} \mathscr{S} \leftarrow \texttt{let}\ \$\overline{v} := e_1\ \texttt{return}\ e_2 : \rho}$$

Auxiliary definitions:

$$\begin{aligned}
\text{form-enriched}(\texttt{()}) &= \texttt{()} \\
\text{form-enriched}(u) &= (\text{form}(u), u) \\
\text{form-enriched}(\tau_1, \tau_2) &= \text{form-enriched}(\tau_1),\ \text{form-enriched}(\tau_2) \\
\text{form-enriched}(\tau_1 \mid \tau_2) &= \text{form-enriched}(\tau_1) \mid \text{form-enriched}(\tau_2) \\
\text{form-enriched}(\tau^+) &= \text{form-enriched}(\tau)^+ \\
\\
\text{single}(\texttt{()}) &= \bot \\
\text{single}((\varphi, u)) &= \varphi \wedge \text{form}(u) \\
\text{single}(\rho_1, \rho_2) &= \begin{cases} \bot & \text{if } \neg\mathit{nullable}(\rho_1) \text{ and } \neg\mathit{nullable}(\rho_2) \\ \text{single}(\rho_1) & \text{if } \neg\mathit{nullable}(\rho_1) \text{ and } \mathit{nullable}(\rho_2) \\ \text{single}(\rho_2) & \text{if } \mathit{nullable}(\rho_1) \text{ and } \neg\mathit{nullable}(\rho_2) \\ \text{single}(\rho_1) \vee \text{single}(\rho_2) & \text{if } \mathit{nullable}(\rho_i) \text{ and } \mathit{nullable}(\rho_j) \end{cases} \\
\text{single}(\rho_1 \mid \rho_2) &= \text{single}(\rho_1) \vee \text{single}(\rho_2) \\
\text{single}(\rho_1 \wedge \rho_2) &= \text{single}(\rho_1) \wedge \text{single}(\rho_2) \\
\text{single}(\rho^+) &= \text{single}(\rho)
\end{aligned}$$

Figure 16: Backward type inference rules for the XQuery core

$$\text{I-SEQ}\quad \frac{S = \{\mathscr{S}_1 \sqcap \mathscr{S}_2 \mid \mathscr{S}_i \leftarrow e_i : \rho_i,\ (\rho_1, \rho_2) \in \text{split}(\rho)\}}{\bigsqcup_{\mathscr{S} \in S} \mathscr{S} \leftarrow (e_1, e_2) : \rho}$$

$$\text{I-FORNULL}\quad \frac{\mathscr{S} \leftarrow e_1 : () \qquad \mathit{nullable}(\rho)}{\mathscr{S} \leftarrow \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2 : \rho}$$

$$\text{I-FOREMPTY}\quad \frac{\mathscr{S}_2 \leftarrow e_2 : () \qquad S = \{\mathscr{S}_1 \sqcap \{C\backslash_{\$v}\} \mid \mathscr{S}_1 \leftarrow e_1 : C(\$v)^*,\ C \in \mathscr{S}_2\}}{\bigsqcup_{\mathscr{S} \in S} \mathscr{S} \leftarrow \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2 : ()}$$

$$\text{I-FORNONEMPTY}\quad \frac{\mathscr{S}_2 \leftarrow e_2 : \rho \qquad \mathbb{0} \leftarrow e_2 : () \qquad S = \{\mathscr{S}_1 \sqcap \{C\backslash_{\$v}\} \mid \mathscr{S}_1 \leftarrow e_1 : C(\$v).\mathit{Quant}(\rho),\ C \in \mathscr{S}_2\}}{\bigsqcup_{\mathscr{S} \in S} \mathscr{S} \leftarrow \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2 : \rho}\ (\rho \neq ())$$

$$\text{I-FOR}\quad \frac{\begin{array}{c}\mathscr{S} \leftarrow e_2 : \rho \\ \mathscr{S}' \leftarrow e_2 : () \qquad S = \left\{ \mathscr{S}'' \sqcap \{C\backslash_{\$v} \sqcap C'\backslash_{\$v}\} \;\middle|\; \begin{array}{l}(C, C') \in \mathscr{S} \times \mathscr{S}', \\ \mathscr{S}'' \leftarrow e_1 : (C'(\$v)^*, C(\$v), C'(\$v)^*).\mathit{Quant}(\rho)\end{array} \right\}\end{array}}{\bigsqcup_{\mathscr{S} \in S} \mathscr{S} \leftarrow \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2 : \rho}\ (\rho \neq ())$$

$$\text{I-ERR}\quad \frac{\text{(if no other rule applies)}}{\mathbb{0} \leftarrow e : \rho}$$

Auxiliary definitions:

$$\begin{array}{rcl}
\text{split}(()) & = & \{((), ())\} \\
\text{split}((\varphi, u)) & = & \{((), (\varphi, u)), ((\varphi, u), ())\} \\
\text{split}(\rho_1 \mid \rho_2) & = & \text{split}(\rho_1) \cup \text{split}(\rho_2) \\
\text{split}(\rho^+) & = & \{((), \rho^+), (\rho^+, ()), (\rho^+, \rho^+)\} \\
\text{split}(\rho_1, \rho_2) & = & \{(\rho_1, \rho_2)\} \cup \{(\rho_{11}, (\rho_{12}, \rho_2)) \mid (\rho_{11}, \rho_{12}) \in \text{split}(\rho_1)\} \cup \\
 & & \{((\rho_1, \rho_{21}), \rho_{22}) \mid (\rho_{21}, \rho_{22}) \in \text{split}(\rho_2)\}
\end{array}$$

$$\begin{array}{rcll}
\mathit{Quant}(\rho) & = & + & \text{if } \rho \text{ is of the form } \rho'^+ \\
\mathit{Quant}(\rho) & = & 1 & \text{otherwise}
\end{array}
\qquad
\begin{array}{rcl}
\rho.+ & = & \rho^+ \\
\rho.1 & = & \rho
\end{array}$$

Figure 17: Backward type inference rules for XQuery core, continued

expression e_1 reduces to a sequence. If all of $\mathscr{S}_1$, $\mathscr{S}_2$, and $\mathscr{S}_3$ are satisfied, then the if-expression has the specified output type ρ. Rules I-IfNonEmpty and I-IfEmpty are straightforward improvements which consider the cases where e reduces to a non-empty sequence and an empty sequence, respectively.

For a let-binding `let` $\$\overline{v}$ `:=` e_1 `return` e_2, rule I-Let first infers a constraint $\mathscr{S}_2$ from e_2 and output type ρ. Then, for each constraint-set $C \in \mathscr{S}_2$ such that $[\![C(\$\overline{v})]\!] \neq \emptyset$, we infer a constraint $\mathscr{S}_1$ from e_1 and $C(\$\overline{v})$. Note that if $[\![C(\$\overline{v})]\!] = \emptyset$, then C is unsatisfiable. In order for the whole let-expression to have type ρ, both $\mathscr{S}_1$ and $C \setminus \$\overline{v}$ should be satisfied, i.e., $\mathscr{S}_1 \sqcap \{C \setminus \$\overline{v}\}$, where $C \setminus \$\overline{v}$ removes the constraint for $\$\overline{v}$ from C because it is bound only in e_2.

In contrast to forward inference, in our backward inference, sequence concatenation and for-loop expressions are the most challenging to define precise type inference. To illustrate, consider an expression `<a/><b/><a/><b/>` and an output type $(\texttt{<a/><b/>})^+$. Here, for simplicity, we write `<a/>` to denote both an XML element $\texttt{<}a\texttt{>}\{\epsilon\}\texttt{</}a\texttt{>}$ and a formula type $(\top, \texttt{element}\ a\ \{\texttt{()}\})$. The meaning of `<a/>` will be clear from the context. For this example, to infer the exact input constraint, we need to unfold output type $(\texttt{<a/><b/>})^+$ two times and use `<a/><b/><a/><b/>` to type the expression of the same form. In general, however, we cannot unfold and split the output type in an exact way to type a sequence expression: given output type $(\rho_1, \rho_2)^+$, we can always come up with a nontrivial sequence expression that requires us to unfold $(\rho_1, \rho_2)^+$ n-times to type it for arbitrary $n \in \mathbb{N}$. The same is true for for-loop expressions as they reduce to a sequence concatenation of the individual results of the `return` expression.

A naive approach to type sequence and for-loop expressions is to unfold the output type k-times (starting with $k = 1$), split the unfolded type, and try to infer an input constraint. If it fails, that is, if the inferred constraint is $\mathbb{0}$, we increase k by 1, and repeat the inference procedure until k reaches some fixed number n. Instead of relying on this semi-decision procedure, however, we simply do not analyze the output type of the form ρ^+ across the boundary of ρ inside its repetition, *i.e.,* we do not unfold ρ^+. In other words, we only type the sequence expressions whose subexpressions can be typed with the whole output type ρ^+. This over-approximation is a trade-off between the practicality and the preciseness of our backward type inference system.

For a sequence concatenation, rule I-Seq divides the output type ρ into two parts using an auxiliary function split(_), defined in Figure 17, which does not analyze the formula type of the form ρ^+ across the boundary of ρ. Then, we infer an input constraint for each case in split(ρ), and returns a union of all inferred constraints as a final result. Note that for any ρ, if $(\rho_1, \rho_2) \in \text{split}(\rho)$ then $\rho_1, \rho_2 = \rho$.

Finally, for-loop expressions are the most challenging to define precise inference rules. Like sequence expressions, when the output type is described as a repetition of a sequence type, *e.g.,* $(\rho, \rho')^+$, there are infinitely many ways to divide it into a set of formula types $\rho_1, \ldots, \rho_n$, where $(\rho_1, \ldots, \rho_n) <: (\rho, \rho')^+$ and each ρ_i can be thought of as an output type of the result of each execution of the `return` expression. Instead of arbitrarily partitioning the output type, we simply consider only the cases where each execution of the `return` expression evaluates to a sequence of focused trees whose type is a subtype of the given output type. The practical implication of this choice is that the output type for for-loop expressions should be of the form $(\rho_1 \mid \ldots \mid \rho_n)^*$ where each ρ_i may be used as a type of the result of each execution of the `return` expression.

Specifically, we use four rules to type for-loop expressions. First of all, we try rules I-ForNull and I-ForEmpty, and if they fail, try rules I-ForNonEmpty and I-For. Given an expression `for` $\$v$ `in` e_1 `return` e_2, rule I-ForNull first tests if e_1 reduces to an empty sequence. If so, *i.e.,* $\mathscr{S} \leftarrow e_1 : \texttt{()}$, the whole for-loop expression reduces to an empty sequence, too, and thus the output type ρ should be nullable. Otherwise, we analyze the `return` expression e_2. If the output type is `()`, no matter how many times we evaluate e_2 with different bindings for $\$v$, it has to reduce to ϵ. Therefore, in the rule I-ForEmpty, we infer a constraint $\mathscr{S}_1$ from e_1 with output type $C(\$v)^*$ where C is a constraint-set inferred by analyzing e_2 with `()` and $[\![C(\$v)]\!] \neq \emptyset$. Note that $C(\$v)^*$ is zero or more repetitions of formula type $C(\$v)$ that makes e_2 reduce to an empty sequence. The other two rules I-ForNonEmpty and I-For cover the cases where output type $\rho \neq \texttt{()}$ and e_1 reduces to a non-empty sequence. Rule I-ForNonEmpty is similar to the rule I-ForEmpty except that it uses an auxiliary function $\mathit{Quant}(\rho)$ and e_2 never reduces to an empty sequence. In this case, the output type for e_1 should be exactly the same as for $\$v$—for example, if the output type is (φ, u), e_1 must reduce to a single focused tree since e_2 never reduces to an empty sequence, regardless of the value of $\$v$. An exception is when the output type is ρ^+. Then, no matter how many times we evaluate e_2, it reduces to a sequence of type ρ^+ and their concatenation is also of type ρ^+ (*i.e.,* $\rho^+, \ldots, \rho^+ = \rho^+$)—in this case, the type for e_1 can be $C(\$v)^+$ where C is a constraint-set inferred from e_2 and $[\![C(\$v)]\!] \neq \emptyset$. We use $\mathit{Quant}(\_)$ to capture this difference. Finally, rule I-For considers the case where e_2 can reduce to both an empty and a non-empty sequence depending on the value of $\$v$. In this case, we infer an input constraint from e_1 with output sequence type $(C'(\$v)^*, C(\$v), C'(\$v)^*)$ where $C'(\$v)$ and $C(\$v)$ are the types that make e_2 reduce to an empty and a non-empty sequence, respectively. In particular, we consider

only those constraint-sets C' and C such that $[\![C'(\$v)]\!] \neq \emptyset$ and $[\![C(\$v)]\!] \neq \emptyset$.

4.2 Complexity

4.2.1 Complexity for XPath axes

We first analyze the complexity of our backward type inference system for XPath axes. To this end, we first define the length $\mathsf{len}(\rho)$ and the size $|\rho|$ of a formula-enriched sequence type ρ:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{len}((\varphi, u)) & = & 1 & |(\varphi, u)| & = & |\varphi| + |u| \\ \mathsf{len}(\,()\,) & = & 1 & |\,()\,| & = & 1 \\ \mathsf{len}(\rho_1, \rho_2) & = & \mathsf{len}(\rho_1) + \mathsf{len}(\rho_2) + 1 & |\rho_1, \rho_2| & = & |\rho_1| + |\rho_2| + 1 \\ \mathsf{len}(\rho_1 \mid \rho_2) & = & \mathsf{len}(\rho_1) + \mathsf{len}(\rho_2) + 1 & |\rho_1 \mid \rho_2| & = & |\rho_1| + |\rho_2| + 1 \\ \mathsf{len}(\rho^+) & = & \mathsf{len}(\rho) + 1 & |\rho^+| & = & |\rho| + 1 \end{array}$$

The size $|\varphi|$ of a formula φ and the length $\mathsf{len}(\tau)$ and the size $|\tau|$ of a regular tree type τ are also defined as usual. In particular, in the analysis below, we mean by $|\tau|$ the size of the classical binary representation of τ Hosoya et al. (2005).

Lemma 4.2. *The time complexity of the auxiliary functions directly used in the inference rules, introduced in Section 3, is as follows.*

- *nullable*(ρ) *is* $O(\mathsf{len}(\rho))$.
- `child-type`(ρ) *is* $O(\mathsf{len}(\rho))$.
- `parent-type`(ρ) *is* $O(\mathsf{len}(\rho))$.

`child-type`(ρ) introduced in Figure 3 is defined only when the argument ρ is of the form $(\varphi_1, u_1) \mid \ldots \mid (\varphi_n, u_n)$ where $u_i = \texttt{element}\ n_i\ \{\tau_i\}$, and its precise complexity is indeed $O(\mathsf{len}(\rho) \times \max \mathsf{len}(\tau_i))$. We consider $\max \mathsf{len}(\tau_i)$ as a constant and omit it in the above analysis.

Among the functions listed in Lemma 4.2, only *nullable*() may be called many times during the inference. More precisely, when the output type is $(\varphi_1, u_1), \ldots, (\varphi_n, u_n)$, the naive cumulative cost of calling *nullable*() is in total $O(n^2)$. With additional space, however, if we memoize the result of *nullable*() on each subterm of the output type ρ when it is called for the first time, the cumulative cost is still $O(\mathsf{len}(\rho))$.

Lemma 4.3. *Given output type ρ, the input type for an XPath axis is inferred in $O(\mathsf{len}(\rho))$ time.*

Proof. Easy from the fact that we analyze the structure of the output type, with an empty type `()` and a pair type (φ, u) as base cases, and the cumulative cost of using auxiliary functions during the inference is $O(\mathsf{len}(\rho))$. □

To analyze the size of the inferred input type, below we assume that we use an optimization technique such as hash-consing to represent types and formulas, *i.e.,* to share the same subterms. Otherwise, in the input type, some formula may be duplicated an exponential number of times in terms of the length of the output type, *e.g.,* when the output type is of the form $(\rho_1 \mid \rho_2), \ldots, (\rho_{n-1} \mid \rho_n)$. Note that in the rule AXIS-OR, with a naive representation of formulas, the with parameter ψ may be duplicated in the input type $\varphi_1 \vee \varphi_2$: one in φ_1, the other in φ_2.

Lemma 4.4. *Assume $\varphi \leftarrow$ axis::n, ρ with ψ. Then the size of φ is $O(|\rho| + |\psi|)$.*

Proof. By induction on a derivation of $\varphi \leftarrow$ *axis*::*n*, ρ with ψ. In the proof, we use the fact that all the auxiliary definitions used in Figure 14, which take a formula χ as argument, return another formula of size $O(|\chi|)$. The proof also relies on that $\mathrm{form}(u)$ has the same size as the classical binary representation of the regular tree type u Genevès and Gesbert (2015). □

Lemma 4.5. *Given output type ρ, the size of the inferred input type for an XPath axis is $O(|\rho|)$.*

Proof. The cases for the axes except `self`, `parent`, and `child` are easily proved by Lemma 4.4. The case for `self` is proved by structural induction on output type ρ. The cases for `parent` and `child` are proved by the fact that the size of `child-type`(ρ) and `parent-type`(ρ) is $O(|\rho|)$. □

Corollary 4.5.1. *Given output type ρ and an XPath axis, we can check in $2^{O(|\rho|)}$ time if there exists some tree that when applied to the axis, returns a sequence of nodes of type ρ, by testing the satisfiability of the inferred input type using the decision procedure in Genevès et al. (2015).*

Precisely, if $\rho' \leftarrow$ *axis*::*n*, ρ, then ρ' is of the form $(\varphi_1, u_1) \mid \ldots \mid (\varphi_n, u_n)$ where $\varphi_i <: u_i$, and thus it suffices to check the satisfiability of each φ_i in the inferred input type.

4.2.2 Complexity for XQuery core

Now we analyze the complexity of our backward type inference system for the XQuery core. We define the size $|C|$ of C and the size $|\mathscr{S}|$ of $\mathscr{S}$ as the number of bindings in C and the number of constraint-sets in $\mathscr{S}$, respectively. Then, $|C_1 \sqcap C_2| \leq |C_1| + |C_2|$, $|\mathscr{S}_1 \sqcap \mathscr{S}_2| \leq |\mathscr{S}_1| \times |\mathscr{S}_2|$, and $|\mathscr{S}_1 \sqcup \mathscr{S}_2| \leq |\mathscr{S}_1| + |\mathscr{S}_2|$. The size $|e|$ of an XQuery expression e is inductively defined as usual, *e.g.,* see Definition 8.1 in Colazzo and Sartiani (2011).

Lemma 4.6. *Suppose $\mathscr{S} \leftarrow \mathrm{e} : \rho$. Then the maximum size, denoted by $T(\mathrm{e}, \rho)$, of the largest type appearing in $\mathscr{S}$ is $O(2^{|\mathrm{e}|}|\rho|)$.*

Proof. By solving the following set of recursive equations, which are derived from the inference rules:

$$
\begin{array}{rcl}
T(e, \rho_1 \mid \rho_2) & = & \max_i T(e, \rho_i) \\
T(e, \rho_1 \wedge \rho_2) & = & T(e, \rho_1) + T(e, \rho_2) + 1 \\
T((e_1, e_2), \rho) & = & \max_{(\rho_1, \rho_2) \in \mathrm{split}(\rho)} (T(e_1, \rho_1) + T(e_2, \rho_2) + 1) \\
T((\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : \texttt{element}\ n\ \{\tau\}), \rho) & = & T(e, \text{form-enriched}(\tau)) \\
T(\texttt{if nempty}(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \rho) & = & T(e_1, (\top, \mathsf{AnyElt})^*) + T(e_2, \rho) + T(e_3, \rho) + 2 \\
T(\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2, \rho) & = & T(e_2, \rho) + T(e_1, T(e_2, \rho)) + 1 \\
T(\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2, \rho) & = & T(e_2, \rho) + T(e_2, \texttt{()}) + \\
 & & \quad T(e_1, (T(e_2, \rho) + 2T(e_2, \texttt{()}) + 5)) + 2 \\
T(e, \rho) & = & O(|\rho|) \qquad \text{(otherwise)}
\end{array}
$$

where we use a type and its size interchangeably as the second argument to $T(\_, \_)$. □

Lemma 4.7. *Suppose $\mathscr{S} \leftarrow \mathrm{e} : \rho$. Then the maximum size, denoted by $N(\mathrm{e}, \rho)$, of $\mathscr{S}$ is $O(2^{2^{|\mathrm{e}|}|\rho|})$.*

Proof. By solving the following set of recursive equations, which are derived from the inference rules. We use the result from Lemma 4.6.

$$
\begin{array}{rcl}
N(e, \rho_1 \mid \rho_2) & = & N(e, \rho_1) + N(e, \rho_2) \\
N(e, \rho_1 \wedge \rho_2) & = & N(e, \rho_1) \times N(e, \rho_2) \\
N((e_1, e_2), \rho) & = & |\mathrm{split}(\rho)| \times \max_{(\rho_1, \rho_2) \in \mathrm{split}(\rho)} (N(e_1, \rho_1) \times N(e_2, \rho_2)) \\
N((\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : \texttt{element}\ n\ \{\tau\}), \rho) & = & N(e, \text{form-enriched}(\tau)) \\
N(\texttt{if nempty}(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \rho) & = & N(e_1, (\top, \mathsf{AnyElt})^*) \times N(e_2, \rho) \times N(e_3, \rho) \\
N(\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2, \rho) & = & N(e_2, \rho) \times N(e_1, T(e_2, \rho)) \\
N(\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2, \rho) & = & N(e_2, \rho) \times N(e_2, \texttt{()}) \times \\
 & & \quad N(e_1, (T(e_2, \rho) + 2T(e_2, \texttt{()}) + 5)) \\
N(e, \rho) & = & O(1) \qquad \text{(otherwise)}
\end{array}
$$

In the above equations, we use a type and its size interchangeably as the second argument to $N(\_, \_)$. □

Lemma 4.8. *Suppose $\mathscr{S} \leftarrow \mathrm{e} : \rho$. Then $\mathscr{S}$ is computed in $2^{O(2^{|\mathrm{e}|}|\rho|)}$ time in the worst case.*

Proof. Let $I(e, \rho)$ denote the complexity of deducing a set of constraint-sets from e and ρ using our inference system. We obtain the complexity by solving the following set of recursive equations, which are derived from the inference rules. We use the result from Lemmas 4.6 and 4.7. We also use a type and its size interchangeably as the second argument to $I(\_, \_)$.

$$
\begin{array}{rcl}
I(\epsilon, \rho) = I(\$\overline{v}, \rho) & = & 1 \\
I(e, \rho_1 \mid \rho_2) = I(e, \rho_1 \wedge \rho_2) & = & 1 + I(e, \rho_1) + I(e, \rho_2) \\
I(\$v, \rho) = I(\$v/\mathit{axis}\texttt{::}n, \rho) & = & O(\mathsf{len}(\rho)) \\
I((e_1, e_2), \rho) & = & 1 + |\mathrm{split}(\rho)| \times \max_{(\rho_1, \rho_2) \in \mathrm{split}(\rho)} (I(e_1, \rho_1) + I(e_2, \rho_2)) \\
I((\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : \texttt{element}\ n\ \{\tau\}), \rho) & = & 2 + I(e, \text{form-enriched}(\tau)) + 2^{O(|u| + |\rho|)} \\
I(\texttt{if nempty}(e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \rho) & = & 1 + I(e_1, (\top, \mathsf{AnyElt})^*) + I(e_2, \rho) + I(e_3, \rho) \\
I(\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2, \rho) & = & 1 + I(e_2, \rho) + N(e_2, \rho) \times I(e_1, T(e_2, \rho)) + \\
 & & \quad N(e_2, \rho) \times 2^{O(T(e_2, \rho))} \\
I(\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2, \rho) & = & 1 + I(e_2, \rho) + I(e_2, \texttt{()}) + N(e_2, \rho) \times N(e_2, \texttt{()}) \times \\
 & & \quad I(e_1, (T(e_2, \rho) + 2T(e_2, \texttt{()}) + 5)) + \\
 & & \quad N(e_2, \rho) \times 2^{O(T(e_2, \rho))} + N(e_2, \texttt{()}) \times 2^{O(T(e_2, \texttt{()}))}
\end{array}
$$

In the above equation, the case of the element construction includes the complexity for the subtype check $u <: \rho$. The cases of let- and for-expressions include the complexity of satisfiability checks for the inferred type for the bound variable, e.g., $C(\$var)$. □

Finally, we state the worst-case time complexity of our backward type inference system for the XQuery core.

Theorem 4.9 (Complexity)**.** *Assume we are given an XQuery expression* e *and its output type ρ. Then the set of* solvable *constraint-sets is computed in $2^{O(2^{(|\mathrm{e}|+1)}|\rho|)}$ time by our inference system. That is, the overall cost is double exponential in terms of the given expression* e.

Proof. Suppose $\mathscr{S} \leftarrow e : \rho$. We obtain $\mathscr{S}$ inl $2^{O(2^{|e|}|\rho|)}$ time by Lemma 4.8. The size of $\mathscr{S}$ is $O(2^{2^{|e|}|\rho|})$ by Lemma 4.7. The size of any constraint-set C in $\mathscr{S}$ is the number of free variables in *e*, which is a constant. Since the size of the largest type in $\mathscr{S}$ is $O(2^{|e|}|\rho|)$ by Lemma 4.6, for each constraint-set C in $\mathscr{S}$, its satisfiability can be tested in $2^{O(2^{|e|}|\rho|)}$ time by the decision procedure in Genevès et al. (2015). Overall, the complexity of our inference system is $2^{O(2^{|e|}|\rho|)} + O(2^{2^{|e|}|\rho|}) \times 2^{O(2^{|e|}|\rho|)}$ which is simply $2^{O(2^{(|e|+1)}|\rho|)}$. □

4.3 Soundness

Now we state the soundness property of our inference system. Below we use $\vdash \eta : C$ to mean that if $\$var \mapsto s \in \eta$, then $(\$var : \rho) \in C$ and $s \in [\![\rho]\!]$.

Theorem 4.10 (Soundness)**.** *Let* e *be an XQuery expression and ρ be a type of its output. Suppose $\mathscr{S} \leftarrow$ e : ρ. Then for any $C \in \mathscr{S}$ such that $C \neq \{\bot\}$, if $\vdash \eta : C$ and $[\![\mathrm{e}]\!]_\eta = s$, then $s \in [\![\rho]\!]$.*

Proof. By induction on a derivation of $\mathscr{S} \leftarrow e : \rho$. Here, we only show the case for the rule I-FOR. Other cases are similarly proved. We have the following assumptions:

1. $\bigsqcup_{\mathscr{S} \in S} \mathscr{S} \leftarrow \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2 : \rho$
2. $C_0 \in \bigsqcup_{\mathscr{S} \in S} \mathscr{S}$ and $\vdash \eta : C_0$
3. $[\![\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2]\!]_\eta = s$

Then, we need to prove $s \in [\![\rho]\!]$.

4. Let $\bigsqcup_{\mathscr{S} \in S} \mathscr{S}$ be $\mathscr{S}_1 \sqcup \ldots \sqcup \mathscr{S}_m$.
5. Without loss of generality, let $C_0 \in \mathscr{S}_i$.

From the premises of the rule I-FOR, we have

6. $\mathscr{S} \leftarrow e_2 : \rho$
7. $\mathscr{S}' \leftarrow e_2 : ()$
8. $C \in \mathscr{S}$ and $C' \in \mathscr{S}'$
9. $\mathscr{S}'' \leftarrow e_1 : (C'(\$v)^*, C(\$v), C'(\$v)^*).Quant(\rho)$
10. $\mathscr{S}_i = \mathscr{S}'' \sqcap \{C\backslash_{\$v} \sqcap C'\backslash_{\$v}\}$

From (3), we have

11. $[\![e_1]\!]_\eta = f_1, \ldots, f_n$
12. $s = \Pi_{f_1,\ldots,f_n} [\![e_2]\!]_{\eta, \$v \mapsto f_i}$

The case where $[\![e_1]\!]_\eta = \epsilon$ is covered by the rule I-FORNULL. From (2), (5) and (10),

13. $\exists C'_0 \in \mathscr{S}''$ such that $C_0 = C'_0 \sqcap C\backslash_{\$v} \sqcap C'\backslash_{\$v}$ and $\vdash \eta : C'_0$.

By induction hypothesis on (9) with (11) and (13), we have

14. $f_1, \ldots, f_n \in [\![(C'(\$v)^*, C(\$v), C'(\$v)^*).Quant(\rho)]\!]$.

Assume $Quant(\rho) = 1$. The case where $Quant(\rho) = +$ is similarly proved using the following property: $\rho^+, \ldots, \rho^+ = \rho^+$. Then, there exists j such that

15. $f_1, \ldots, f_{j-1} \in [\![C'(\$v)^*]\!]$ and thus $f_k \in [\![C'(\$v)]\!]$ where $k = 1, \ldots, j-1$
16. $f_j \in [\![C(\$v)]\!]$
17. $f_{j+1}, \ldots, f_n \in [\![C'(\$v)^*]\!]$ and thus $f_k \in [\![C'(\$v)]\!]$ where $k = j+1, \ldots, n$

From (2), (5) and (10), we have $\vdash \eta : C\backslash_{\$v}$ and $\vdash \eta : C'\backslash_{\$v}$. Together with (15)–(17), we have

18. $\vdash \eta, \$v \mapsto f_j : C$
19. $\vdash \eta, \$v \mapsto f_k : C'$ where $k = 1, \ldots, j-1, j+1, \ldots, n$

By induction hypothesis on (6) and (7) with (18) and (19), respectively, we have

20. $[\![e_2]\!]_{\eta, \$v \mapsto f_j} \in [\![\rho]\!]$
21. $[\![e_2]\!]_{\eta, \$v \mapsto f_k} \in [\![\, () \,]\!]$ where $k = 1, \ldots, j-1, j+1, \ldots, n$

From (20) and (21), we have $s = \Pi_{f_1, \ldots, f_n} [\![e_2]\!]_{\eta, \$v \mapsto f_i} \in [\![\rho]\!]$ as desired. □

Unlike the type inference for XPath axes, the type inference for the XQuery core is only sound and not complete, mainly because of the approximation introduced for sequence concatenation and for-loop expressions. From the soundness and the decidability of the inference system, we deduce a sound typechecking algorithm as a corollary.

Corollary 4.10.1 (Typechecking)**.** *Let* e *be an XQuery expression with an only free variable* $\$doc$*, which denotes an input document. Let* ρ_i *be an input type (the type for* $\$doc$*) and* ρ_o *an output type. Then there exists an algorithm that says yes if* $\mathscr{S} \leftarrow \mathrm{e} : \rho_o$ *and* $\exists C \in \mathscr{S}$ *such that* $C \neq \{\bot\}$ *and* $\rho_i <: C(\$doc)$*.*

In the corollary above, the use of variable $\$doc$ has no particular implication; it suffices to have a name of some element that is considered as the root in the input type, *e.g., e* should be of the form `let` $\$doc$ `:=` `/self::*` `return` e'. To typecheck a given expression e with input and output types ρ_i and ρ_o, we first infer a constraint-set C from e and ρ_o using our backward type inference, and then simply check the inclusion relation between ρ_i and the inferred type $C(\$doc)$.

5 Related Work and Discussion

The problem of typechecking XML transformations has been extensively studied since the introduction of XML. For a detailed survey of XML typechecking, we refer the reader to Møller and Schwartzbach (2005); Benzaken et al. (2009) and references therein. In this section, we only discuss the work on backward type inference and recent proposals for precise type systems for XQuery.

5.1 Backward Type Inference

Given an expression e that transforms XML documents of type ρ_i into documents of type ρ_o, there are two major approaches to typechecking e: forward inference and backward inference. The former first computes the image O of the input type ρ_i under the transformation e, *i.e.,* $O := \{e(t) \mid t \in \rho_i\}$, and then checks if $O \subseteq \rho_i$; the latter instead computes the pre-image I of the output type ρ_o under e, *i.e.,* $I := \{t \mid e(t) \in \rho_o\}$, and then checks if $\rho_i \in I$. When types are modeled as regular tree languages, exact typechecking may be done in the form of backward type inference, whereas not in forward inference because even for simple XML transformations, their image may not be regular Milo et al. (2003). Still, forward inference is more intuitive than backward inference, and thus many practical XML programming languages such as XQuery Boag et al. (2010); Draper et al. (2010), XDuce Hosoya and Pierce (2003), and CDuce Benzaken et al. (2003) build on forward inference and instead introduce some approximation: thus some type-safe programs are rejected in these languages.

In contrast, Milo et al. (2003) propose a problem of *inverse type inference*, which is another name of backward type inference, to provide exact typechecking for XML transformations. Specifically, they use *k-pebble tree transducers* for XML transformations, which are finite state tranducers that can mark nodes of the input tree

using up to k different pebbles. Although their approach is exact, it has little practical applicability because its complexity is non-elementary, *i.e.,* when using k pebble, its complexity is $O(h_{k+2}(n))$ with $h_0(n) = n$ and $h_{m+1}(n) = 2^{h_m(n)}$. Similarly, Maneth et al. (2005) also studied the problem of exact typechecking for tree transformations using macro tree transducers Engelfriet and Vogler (1985), which can accumulate part of the input and copy it in the output, in the form of backward type inference. They use Monadic Second Order Logic (MSO) as pattern language, which subsumes XPath without arithmetics and data value comparisons. Their formalism, however, is based on finite automata and thus requires for implementation purposes a translation from MSO to a finite automaton which may introduce a non-elementary blow up. In this paper, we also study the problem of backward type inference, but develop a type inference system directly on the XQuery core. Although our system is exact only for XPath axes, it is more practical than Milo et al. (2003); Maneth et al. (2005) with a much lower complexity.

5.2 Precise Type Systems for XQuery

Typing XPath expressions has been a challenging topic and until recently there was no satisfactory solution. Most previous proposals for the XQuery static type system, including the one standardized by the W3C Draper et al. (2010), support only downward navigation in XML trees. As thoroughly discussed in Genevès and Gesbert (2015), it is mainly due to the discrepancy between the XML data model and the type model, namely regular tree types Hosoya et al. (2005). Recently, Castagna et al. (2015) and Genevès and Gesbert (2015) independently proposed an extended type language to describe not only a given XML tree node but also its context. Below, we discuss each in turn.

In Castagna et al. (2015), the authors extend the core calculus of CDuce Benzaken et al. (2003) with zipper data structures Huet (1997), which denote the position in the surrounding tree of the value they annotate as well as its current path from the root. By annotating not only values but also types with zippers, they allow tree navigation in any direction and such navigational expressions can be typed precisely (in their work, zipped values and zipped types play a similar role as focused trees and formula-enriched sequence types, respectively). Then, they propose a translation from XQuery 3.0 Core Robie et al. (2014b); Benedikt and Vu (2012), which newly added value and type case analysis and higher-order functions, into the extended CDuce and provide a type system for XQuery 3.0 via the translation. In contrast to Castagna et al. (2015), currently we do not support function declarations and applications, and thus higher-order functions as well. However, because regular tree types extended with arrow types can be translated into tree logic formulas and their subtype relation can be decided through the logic's decision procedure Gesbert et al. (2011), we expect that our type system can be easily extended with (higher-order) functions at least in theory.

Our work was inspired by the work of Genevès and Gesbert (2015), who first proposed the idea of using focused trees to denote XML values and of combining regular tree types with tree logic formulas to describe both tree nodes and their contexts simultaneously, and thus which supports all the major navigational features of the XQuery core. The main difference is that while they use forward inference, we use backward inference. Our backward type inference is arguably more complex because we need to analyze the structure of the output type as well as the given expression (in particular, inference rules for sequence concatenation and for-loop expressions are simpler in Genevès and Gesbert (2015)), but as a trade-off it provides an exact typechecking algorithm for XPath axes. Another difference is that while Genevès and Gesbert (2015) uses a small-step operational semantics for the XQuery core, we use a denotational semantics because it is more suitable for proving properties of our backward type inference. Considering all these aspects, it will be quite interesting to combine their approach with ours.

6 Conclusion

In this paper, we propose a novel backward type inference system for XQuery as a complementary method to forward type inference. In particular, the contributions of the paper is summarized as follows. First, we define a focused-tree-based denotational semantics for a navigational fragment of XQuery, including all major XPath axes. Second, we propose a novel tree-logic-based backward type inference system for XPath axes and prove its soundness and completeness. In contrast to ours, forward type inference is only sound. Finally, based on this result, we propose a sound and practical backward type inference system for the XQuery core.

An interesting direction for future work would be to develop a bidirectional typechecking algorithm by combining both backward and forward type inference methods. The basic idea is to typecheck sequence concatenation and for-loop expressions using forward type inference, thus obtaining a lower complexity and better precision than

our backward approach, while typechecking XPath axes using backward type inference, thus obtaining better precision than the forward approaches Castagna et al. (2015); Genevès and Gesbert (2015). In doing so, one possible difficulty would be to find minimal type annotations to enable effective bidirectional typechecking.

References

Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. 2003. XML with data values: typechecking revisited. *J. Comput. System Sci.* 66, 4 (2003), 688–727. DOI:http://dx.doi.org/10.1016/S0022-0000(03)00032-1

Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.* 15, 4 (Sept. 1993), 575–631. DOI:http://dx.doi.org/10.1145/155183.155231

Michael Benedikt and Huy Vu. 2012. Higher-Order Functions and Structured Datatypes. In *Proceedings of the 15th International Workshop on the Web and Databases (WebDB 2012)*. 43–48. http://db.disi.unitn.eu/pages/WebDB2012/papers/p13.pdf

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-centric General-purpose Language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, New York, NY, USA, 51–63. DOI:http://dx.doi.org/10.1145/944705.944711

Véronique Benzaken, Giuseppe Castagna, Haruo Hosoya, Benjamin C. Pierce, and Stijn Vansummeren. 2009. XML Typechecking. In *Encyclopedia of Database Systems*. Springer US, 3646–3650. DOI:http://dx.doi.org/10.1007/978-0-387-39940-9_788

Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. 2010. XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation. (December 2010). Retrieved February 28, 2017 from http://www.w3.org/TR/xpath20

Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. 2010. XQuery 1.0: An XML Query Language (Second Edition). W3C Recommendation. (December 2010). Retrieved February 28, 2017 from http://www.w3.org/TR/xquery/

Anne Brüggemann-Klein and Derick Wood. 1998. One-Unambiguous Regular Languages. *Information and Computation* 140, 2 (1998), 229–253. DOI:http://dx.doi.org/10.1006/inco.1997.2688

Giuseppe Castagna, Hyeonseung Im, Kim Nguyen, and Véronique Benzaken. 2015. A Core Calculus for XQuery 3.0: Combining Navigational and Pattern Matching Approaches. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, Proceedings*. 232–256. DOI:http://dx.doi.org/10.1007/978-3-662-46669-8_10

James Clark and Steve DeRose. 1999. XML Path Language (XPath) Version 1.0. W3C Recommendation. (November 1999). Retrieved February 28, 2017 from http://www.w3.org/TR/xpath

Dario Colazzo and Carlo Sartiani. 2011. Precision and Complexity of XQuery Type Inference. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP '11)*. ACM, New York, NY, USA, 89–100. DOI:http://dx.doi.org/10.1145/2003476.2003490

Denise Draper, Michael Dyck, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. 2010. XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition). W3C Recommendation. (December 2010). Retrieved February 28, 2017 from http://www.w3.org/TR/xquery-semantics/

Joost Engelfriet and Heiko Vogler. 1985. Macro tree transducers. *J. Comput. System Sci.* 31, 1 (1985), 71–146. DOI:http://dx.doi.org/10.1016/0022-0000(85)90066-2

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *J. ACM* 55, 4, Article 19 (Sept. 2008), 64 pages. DOI: http://dx.doi.org/10.1145/1391289.1391293

Pierre Genevès and Nils Gesbert. 2015. XQuery and Static Typing: Tackling the Problem of Backward Axes. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 88–100. DOI:http://dx.doi.org/10.1145/2784731.2784746

Pierre Genevès, Nabil Layaïda, and Alan Schmitt. 2007. Efficient Static Analysis of XML Paths and Types. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 342–351. DOI:http://dx.doi.org/10.1145/1250734.1250773

Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. 2015. Efficiently Deciding μ-Calculus with Converse over Finite Trees. *ACM Trans. Comput. Logic* 16, 2, Article 16 (April 2015), 41 pages. DOI: http://dx.doi.org/10.1145/2724712

Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2011. Parametric Polymorphism and Semantic Subtyping: The Logical Connection. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 107–116. DOI:http://dx.doi.org/10.1145/2034773.2034789

Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.* 3, 2 (May 2003), 117–148. DOI:http://dx.doi.org/10.1145/767193.767195

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.* 27, 1 (Jan. 2005), 46–90. DOI:http://dx.doi.org/10.1145/1053468.1053470

Gérard Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (Sept. 1997), 549–554. DOI:http://dx.doi.org/10.1017/S0956796897002864

S. Maneth, A. Berlea, T. Perst, and H. Seidl. 2005. 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)*. ACM, New York, NY, USA, 283–294. DOI:http://dx.doi.org/10.1145/1065167.1065203

Tova Milo, Dan Suciu, and Victor Vianu. 2003. Typechecking for XML transformers. *J. Comput. System Sci.* 66, 1 (2003), 66–97. DOI:http://dx.doi.org/10.1016/S0022-0000(02)00030-2

Anders Møller and Michael I. Schwartzbach. 2005. The Design Space of Type Checkers for XML Transformation Languages. In *Proceedings of the 10th International Conference on Database Theory (ICDT '05)*. Springer-Verlag, 17–36. DOI:http://dx.doi.org/10.1007/978-3-540-30570-5_2

Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. 2005. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Internet Technol.* 5, 4 (Nov. 2005), 660–704. DOI: http://dx.doi.org/10.1145/1111627.1111631

Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. 2014a. XML Path Language (XPath) 3.0. W3C Recommendation. (April 2014). Retrieved February 28, 2017 from http://www.w3.org/TR/xpath-30/

Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. 2014b. XQuery 3.0: An XML Query Language. W3C Recommendation. (April 2014). Retrieved February 28, 2017 from http://www.w3.org/TR/xquery-30/