HAL
archives-ouvertes.fr

Backward Type Inference for XML Queries

Hyeonseung Im, Pierre Genevès, Nils Gesbert, Nabil Layaïda

To cite this version:

Hyeonseung Im, Pierre Genevès, Nils Gesbert, Nabil Layaïda. Backward Type Inference for XML Queries. 2018. hal-01497857v2

HAL Id: hal-01497857

https://hal.inria.fr/hal-01497857v2

Submitted on 17 Jan 2019

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

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

Backward Type Inference for XML Queries

Hyeonseung Im[a,*], Pierre Genevès[b,e], Nils Gesbert[c,e], Nabil Layaïda[d,e]

[a] *Kangwon National University, 1 Kangwondaehak-gil, Chuncheon-si, 24341 Republic of Korea*
[b] *CNRS, France*
[c] *Grenoble INP – Ensimag, France*
[d] *Inria, France*
[e] *655 avenue de l'Europe, 38330 Montbonnot, France*

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 proposed 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 such forward type inference systems, we propose a novel backward type inference system for XQuery, based on the extended 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.

Keywords: XPath, XQuery, Static type system, Type inference, Regular tree types, Mu-calculus

1. Introduction

XQuery [1] 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 (*i.e.*, regular tree languages) [2] and its static and dynamic semantics are formally defined [3]. One of the key features of XQuery is its use of XPath [4, 5] 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 document. In the end, in the recent recommendations of XPath 3.0 [6] and XQuery 3.0 [7], 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.,* [8]).

*Corresponding author
Email addresses: `hsim@kangwon.ac.kr` (Hyeonseung Im), `pierre.geneves@cnrs.fr` (Pierre Genevès), `nils.gesbert@grenoble-inp.fr` (Nils Gesbert), `nabil.layaida@inria.fr` (Nabil Layaïda)

There are two different approaches to alleviate this limitation. The first approach is to develop a typechecking algorithm based on backward type inference (also known as inverse type inference) [9, 10, 11, 12, 13, 14]. Given an XQuery expression e and an expected output type ρ_o, backward type inference computes the pre-image ρ_i of ρ_o with respect to 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 the pre-image of a regular tree language with respect to a macro tree transducer (MTT) is also regular [15], MTTs and their variants have often been used as a model of XML transformations in the context of backward type inference [11, 12, 14]. Although exact typechecking can be done with backward type inference, its complexity is hyper-exponential (*i.e.,* a stack of exponentials) [10, 12, 16]. To our knowledge, both practical and exact backward type inference for general XML transformations exploiting backward axes such as `parent` and `anc` has not been reported yet.

In contrast, the second approach is to develop an approximate but practical forward type inference system by using a refined type language that can describe not only XML nodes but also their contexts. For example, Castagna *et al.* [17] extend regular tree types with zipper data structures [18] and propose a precise type system for XQuery 3.0. Their type system supports all navigational XQuery expressions including type and value case analysis and higher-order functions. Genevès and Gesbert [19] also develop a precise type system for XQuery by combining regular tree types with modal logic formulas [20]. By encoding context information using modal formulas, their type system also deduces precise types for backward axes as well as forward axes. However, although practical implementation is feasible, forward type inference cannot be exact as it infers an output type given an input type and a general transformation does not preserve regularity.

In this paper, we propose a novel backward type inference system for XQuery as a complementary method to forward type inference systems such as in [17, 19]. In particular, we develop an XQuery source language type system using the refined type language proposed in [19]. While tree transducers can be used as an intermediate language for XQuery, having a source language type system in itself is useful as it is usually easier to understand. Moreover, by building a backward type inference system on the XQuery syntax and the existing type language, it would be possible to combine it with forward type inference, for example, in order to develop a more precise and practical bidirectional typechecking algorithm. Thus, this work can be considered as a stepping stone towards such bidirectional type systems.

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* [20] (Section 2). A focused tree is a variant of zipper data structures [18], 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 [8] with non-downward XPath axes.

As for our type language, we use formula-enriched sequence types [19], which combine the usual regular tree types with tree logic formulas [20] to describe both a tree node and its context (Section 3). Then, using formula-enriched sequence types, we define an exact backward type inference system for XPath axes (Section 4). 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 type inference system for the XQuery core (Section 5). In the presence of an arbitrary `for`-expression with a formula-enriched sequence type as an output type, both practical and exact typing is nontrivial or even may be infeasible, 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 type inference for XPath axes is exact and its complexity is simple exponential.

- We prove soundness of our backward type inference system for the XQuery core, from which we can obtain a typechecking algorithm. We also formally analyze the complexity of our inference system, and show that its complexity is double exponential in terms of the given expression.

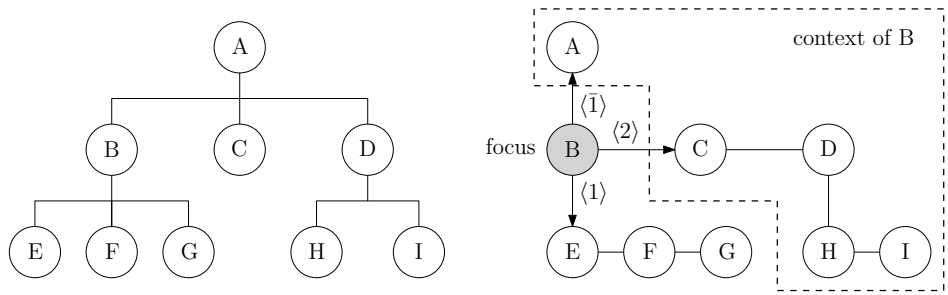


Figure 1: An example XML tree structure and its corresponding binary representation

2. Syntax and Semantics of an XQuery Core

In this section, we introduce an XQuery Core, a minimal XQuery fragment supporting all the navigational XPath axes. Our XQuery core is an extension of miniXQuery proposed in [8] with non-downward axes.

2.1. Focused Trees

We first define XML trees as focused trees, inspired by Huet's zipper data structure [18]. 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.

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

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.

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 if and only if the current node is the leftmost node, *i.e.,* it has no left sibling. Definition 2.1 formally defines the navigation of focused trees.

Definition 2.1 (Navigation of focused trees).

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

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

Expressions	e	$::=$	$\epsilon \mid \texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u \mid e, e \mid \texttt{for}\ \$v\ \texttt{in}\ e\ \texttt{return}\ e$
		$\mid$	$\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e\ \texttt{return}\ e \mid \texttt{if nempty}(e)\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid$	$\$v/axis{::}n \mid \$var$
Variables	$\$var$	$::=$	$\$v \mid \$\overline{v} \mid \$doc$
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 2: Syntax of the XQuery core

Example 2.2. Consider the example XML tree in Figure 1. If the node labeled B is a focus, then the focused tree f_B and its navigations are defined as follows. For simplicity, for each node, we write only its label and omit the child nodes unless necessary.

$$\begin{array}{rcll} f_B & = & (B, (\epsilon; \mathsf{Top}[A]; C :: D :: \epsilon)) & \\ f_B\,\langle 1\rangle & = & (E, (\epsilon; c[B]; F :: G :: \epsilon)) & \text{where } c = (\epsilon; \mathsf{Top}[A]; C :: D :: \epsilon) \\ f_B\,\langle 2\rangle & = & (C, (B :: \epsilon; \mathsf{Top}[A]; D :: \epsilon)) & \\ f_B\,\langle \bar{1}\rangle & = & (A[B :: C :: D :: \epsilon], \mathsf{Top}) & \\ f_B\,\langle \bar{2}\rangle & = & \text{undefined} & \end{array}$$

2.2. XQuery Core

Figure 2 defines the abstract syntax of a simplified navigational fragment of the XQuery core, defined in the XQuery 1.0 and XPath 2.0 Formal Semantics [3]. 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 [21]). 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.

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 3.1). 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 a `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 may thus be considered as a combination of XQuery's untyped element constructor and a `validate` expression, but its typechecking is done statically. 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, $\$doc$ is a special variable for reading the input document, and ϵ denotes an empty sequence, *i.e.,* $e, \epsilon = \epsilon, e = \epsilon$. 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 $\texttt{if nempty}(e)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, 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 $\$v/axis{::}n$ 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 because other axes can easily be encoded.[1] 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, `following-sibling` is denoted by `nsibl`.

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

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

Auxiliary definitions: $\texttt{name}((\sigma[tl], c)) = \sigma$

Figure 3: Semantics of the XQuery core

An XQuery expression e evaluates to a value s, which is defined as a sequence of focused trees. 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.

2.3. Semantics

Figure 3 shows the semantics of the XQuery core, which is defined using the following denotation function:

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

where a substitution η is a mapping from variables to values.

While most of the rules are straightforward and compositional, we took special care for an element constructor ($\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} : u$). First, suppose that the inner expression e evaluates to a sequence $[f_1, \ldots, f_n]$ of focused trees, where $f_i = (t_i, c_i)$. Then we embed them into a new tree structure, 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 $f = (\sigma[t_1 :: \ldots :: t_n :: \epsilon], \mathsf{Top})$.

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. 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.

3. Type Language

Our type language is based on regular tree types [2] and a tree logic, which is a sub-logic of the alternation free modal μ-calculus with converse [20]. In this section, we first briefly introduce regular tree types and the tree logic, together with their semantics in terms of sets of focused trees. Then we introduce our type language, regular tree types enriched with tree logic formulas [19].

3.1. Regular Tree Types

We use a slight variant of XDuce's regular expression type language [22] to type sequences of XML trees (or *elements*), which is expressive enough to capture standard XML types such as DTD and XML Schema [23]. Formally we define our regular tree types as follows.

Definition 3.1 (Regular tree types).

Unit types	u	$::=$	$\texttt{element}\ n\ \{\tau\}$
Name tests	n	$::=$	$\sigma \mid *$
Sequence types	τ	$::=$	$u \mid \texttt{()} \mid \tau,\tau \mid (\tau \mid \tau) \mid \tau^* \mid x$

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 a *type environment* and type variables to support vertical recursion. A type environment E is a finite mapping from type variables x to types τ. For example, we assume that every E that we consider in this paper maps a type variable AnyElt into $\texttt{element}\ *\ \{\mathsf{AnyElt}^*\}$, which is the type of all elements. 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 [24]. We also assume that regular expressions defined by E are composed of mutually exclusive unit types and *1-unambiguous* [25], 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 3.2. Given a type environment E, the semantics of types is defined as the smallest function $[\![_]\!]_E$ that satisfies the following set of equations:

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

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 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.

3.1.1. 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.

Definition 3.3. The *focused-tree interpretation* $[\![\tau]\!]\uparrow$ of a type τ is defined as the set:

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

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

Example 3.4. Consider the example XML tree in Figure 1 again. Let f_B, f_C, and f_D be the focused trees focusing on the nodes labeled B, C, and D, respectively. For leaf nodes, we write only its label.

$$\begin{array}{rcl} f_B & = & (B[E :: F :: G :: \epsilon], (\epsilon; \mathsf{Top}[A]; C :: D[H :: I :: \epsilon] :: \epsilon)) \\ f_C & = & (C, (B[E :: F :: G :: \epsilon] :: \epsilon; \mathsf{Top}[A]; D[H :: I :: \epsilon] :: \epsilon)) \\ f_D & = & (D[H :: I :: \epsilon], (C :: B[E :: F :: G :: \epsilon] :: \epsilon; \mathsf{Top}[A]; \epsilon)) \end{array}$$

Then, f_B, f_C, and f_D match regular tree types τ_B, τ_C, and τ_D, respectively.

$$\begin{array}{rcl} \tau_B & = & \texttt{element}\ B\ \{\texttt{element}\ E\ \{\texttt{()}\}, \texttt{element}\ F\ \{\texttt{()}\}, \texttt{element}\ G\ \{\texttt{()}\}\} \\ \tau_C & = & \texttt{element}\ C\ \{\texttt{()}\} \\ \tau_D & = & \texttt{element}\ D\ \{\texttt{element}\ H\ \{\texttt{()}\}, \texttt{element}\ I\ \{\texttt{()}\}\} \end{array}$$

Of course, they also match a more general type such as AnyElt. Note however that we cannot describe the context information using Definition 3.1.

As shown in the above definition and example, 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` $\$v$`/psibl::`$*$. Let us consider forward type inference; reasoning with backward type inference is similar. Suppose that e is of type τ_D and reduces to f_D. Then, we need to compute f_D`/psibl::`$*$, which reduces to $[f_B, f_C]$. The type of this result, however, should be determined by analyzing τ_D only, without evaluating the given expression. Since τ_D does not contain any useful information about its preceding siblings, we cannot deduce a meaning type for f_D`/psibl::`$*$, and thus for the entire for-loop expression. Therefore, every type system for XQuery built solely on the type language given in Definition 3.1 simply gives to this expression the most general type AnyElt^*.

In the next section, we propose to use a tree logic to solve this problem.

3.2. 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 [20]. 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 [26].

Definition 3.5 (Logic formulas).

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

$a \in \{1,2,\bar{1},\bar{2}\}$ 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 subformula 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:

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

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 $\mathtt{name}(f)$ to denote the label at the current node of f.

Definition 3.6 (Interpretation of formulas).

$$\begin{array}{rcl@{\qquad}rcl} \langle\!\langle\top\rangle\!\rangle_V & \stackrel{\text{def}}{=} & \mathcal{F} & \langle\!\langle\langle a\rangle\,\varphi\rangle\!\rangle_V & \stackrel{\text{def}}{=} & \{f\,\langle\overline{a}\rangle \mid f\in\langle\!\langle\varphi\rangle\!\rangle_V\} \\ \langle\!\langle X\rangle\!\rangle_V & \stackrel{\text{def}}{=} & V(X) & \langle\!\langle\neg\,\langle a\rangle\,\top\rangle\!\rangle_V & \stackrel{\text{def}}{=} & \{f \mid f\,\langle a\rangle \text{ undefined}\} \\ \langle\!\langle\sigma\rangle\!\rangle_V & \stackrel{\text{def}}{=} & \{f \mid \mathtt{name}(f)=\sigma\} & \langle\!\langle\varphi\vee\psi\rangle\!\rangle_V & \stackrel{\text{def}}{=} & \langle\!\langle\varphi\rangle\!\rangle_V \cup \langle\!\langle\psi\rangle\!\rangle_V \\ \langle\!\langle\neg\sigma\rangle\!\rangle_V & \stackrel{\text{def}}{=} & \{f \mid \mathtt{name}(f)\neq\sigma\} & \langle\!\langle\varphi\wedge\psi\rangle\!\rangle_V & \stackrel{\text{def}}{=} & \langle\!\langle\varphi\rangle\!\rangle_V \cap \langle\!\langle\psi\rangle\!\rangle_V \end{array}$$

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

In the rest of the paper, we consider only closed formulas and write $\langle\!\langle\psi\rangle\!\rangle$ for $\langle\!\langle\psi\rangle\!\rangle_\emptyset$. We say that a focused tree f matches a formula ψ if $f\in\langle\!\langle\psi\rangle\!\rangle$.

Example 3.7. Consider the focused trees f_B, f_C, and f_D given in Example 3.4. f_D matches a formula ψ_D where the underlined part describes the subtree rooted at D and the other part describes its context.

$$\psi_D = \underline{D\wedge\langle 1\rangle\,(H\wedge\langle 2\rangle\,I)}\wedge\langle\bar{2}\rangle\,(C\wedge\langle\bar{2}\rangle\,(B\wedge\langle 1\rangle\,(E\wedge\langle 2\rangle\,(F\wedge\langle 2\rangle\,G))\wedge\langle\bar{1}\rangle\,A))$$

From ψ_D, we can now infer formulas ψ_C and ψ_B for f_C and f_B which are the preceding siblings of f_D.

$$\begin{array}{rl} \psi_C = \langle 2\rangle\,\psi_D & = \langle 2\rangle\,(D\wedge\langle 1\rangle\,(H\wedge\langle 2\rangle\,I))\wedge\underline{C}\wedge\langle\bar{2}\rangle\,(B\wedge\langle 1\rangle\,(E\wedge\langle 2\rangle\,(F\wedge\langle 2\rangle\,G))\wedge\langle\bar{1}\rangle\,A) \\ \psi_B = \langle 2\rangle\,\langle 2\rangle\,\psi_D & = \langle 2\rangle\,(\langle 2\rangle\,(D\wedge\langle 1\rangle\,(H\wedge\langle 2\rangle\,I))\wedge C)\wedge\underline{B\wedge\langle 1\rangle\,(E\wedge\langle 2\rangle\,(F\wedge\langle 2\rangle\,G))}\wedge\langle\bar{1}\rangle\,A \end{array}$$

Note however that we cannot deduce a formula for a sequence of focused trees $[f_B, f_C]$.

3.3. Formula-Enriched Sequence Types

In order to type *sequences* of focused trees, which are *values* of our XQuery core, we simply enrich the type language in Definition 3.1 by associating a formula to each unit type. The enriched types, which we call formula types, are thus regular expressions of pairs of unit types and formulas, as defined below.

Definition 3.8 (Formula types).

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

A formula type (φ, u) describes a focused tree (t, c) where u describes only t while φ may describe both t and c. The interpretation of a pair (φ, u) is defined as a 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 semantically defined as the set inclusion relation $[\![\rho_1]\!] \subseteq [\![\rho_2]\!]$.

Example 3.9. Consider a for-loop expression `for` $\$v$ `in` f_D `return` $\$v$`/psibl::`$*$. If the type of f_D is given as (ψ_D, τ_B), then the type of the whole expression may be deduced as $(\mu X.\, \langle 2\rangle\, (\psi_D \vee X), \mathsf{AnyElt})^*$. The type states that the for-loop expression will reduce to a possibly empty sequence of focused trees, each of which has a following sibling of type $(\psi_D, \mathsf{AnyElt})$. As discussed in Section 3.1, we cannot deduce any meaning information in the regular tree type part, and thus simply use AnyElt.

The rationale behind the use of formula types is that it provides more flexibility. From the example, one might think that regular expressions of formulas would be sufficient, which is true for backward type inference for XPath axes. However, sometimes, we may want to ignore context information, for example, to construct a new XML tree node using existing focused trees. In this case, we need to eliminate the context information from the formula matched with each focused tree. Unfortunately, it is nontrivial to eliminate only context information in the presence of recursive formulas. Thus, by combining formulas with regular tree types, we can make use of the usual regular tree type part, ignoring the formula part, if necessary. Moreover, although we do not investigate in this paper, by using the same type language as in [19], it would be easier to integrate our backward type inference with their forward type inference.

In Section 2.2, we assumed that every XML element constructor was annotated not with a formula type (φ, u) but with 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 $(\top, u)$.

4. Inference for XPath Axes

In this section, we present a sound and complete backward type inference system for XPath axes, and based on this we will develop a backward type inference system for the XQuery core in Section 5.

In backward type inference, we are given an expression e and an output type ρ_o for a sequence of focused trees that e may produce. Then we infer an input type ρ_i such that for any tree t of type ρ_i, $e(t)$ always produces a sequence of nodes of type ρ_o. When considering XPath axes, this means that we infer a type describing a set of input 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 a given axis *axis* and an 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.

Invariant 4.1. *In our backward type inference system for XPath axes, if (φ, u) is an 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 $t \in [\![u]\!]$.*

The implication of this invariant is that for type inference for XPath axes, we can safely ignore the regular tree type part because it is always less precise than the formula part.

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,

$$\textsc{Self-Empty}\quad \frac{}{(\neg k(n), \mathsf{AnyElt}) \leftarrow \texttt{self::}n,\ \texttt{()}}$$

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

$$\textsc{Self-Seq1}\quad \frac{\neg \mathit{nullable}(\rho_i) \qquad \mathit{nullable}(\rho_j) \qquad \rho' \leftarrow \texttt{self::}n,\ \rho_i}{\rho' \leftarrow \texttt{self::}n,\ (\rho_1, \rho_2)}\ (i, j = 1, 2, i \neq j)$$

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

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

$$\textsc{Self-Plus}\quad \frac{\rho' \leftarrow \texttt{self::}n,\ \rho}{\rho' \leftarrow \texttt{self::}n,\ \rho^+}$$

$$\textsc{Parent}\quad \frac{\rho' \leftarrow \texttt{self::}n,\ \rho}{\texttt{child-type}(\rho', \mathit{nullable}(\rho)) \leftarrow \texttt{parent::}n,\ \rho}$$

Auxiliary definitions:

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

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

Figure 4: Inference rules for `self` and `parent`

regardless of its context. Technically, this translation can be done using an auxiliary function form(u), which is defined and proved correct in [19]. (For its precise definition, we refer the reader to Figure 10 in [19].) Next, we test the satisfiability of the formula $\varphi \wedge \neg\mathrm{form}(u)$, for example, using the decision procedure presented in [26]; 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`.

4.1. Inference Rules for `self` and `parent`

4.1.1. Self

Figure 4 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, *i.e.,* $k(*) = \top$ and $k(\sigma) = \sigma$.

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 $\mathrm{form}(u)$. Since $\varphi \wedge k(n) \wedge \mathrm{form}(u) <: u$ holds in the rule Self-Formula, Invariant 4.1 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 $\texttt{self::}n$, produces a tree of type $\texttt{element}\ n'\ \{\tau\}$ 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 $\texttt{self::}n$ 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 $\texttt{self::}n$ returns at most one node (rule Self-Plus).

4.1.2. Parent

The intuition behind type inference for `parent` is simple. Given an output type ρ, it is the type of the parent of the input context node. Moreover, if we infer ρ' using the inference rules for `self` with ρ, then the parent node is also of type ρ'. In other words, the input node is a child of the node of type ρ'. Therefore, for the input node, we extract a child type from ρ' using an auxiliary function `child-type`().

To illustrate, assume that the output type ρ is given as $(A, \tau_A)^+$ where $\tau_A = \texttt{element}\ A\ \{\tau_B, \tau_C, \tau_D\}$ and τ_B, τ_C, and τ_D are defined in Example 3.4. Note that ρ is a type for the focused tree f_A rooted at the node labeled A in Figure 1. By applying the inference rules for $\texttt{self::}n$, we obtain $\rho' = (\varphi_A, \tau_A)$ where $\varphi_A = A \wedge k(n) \wedge \mathrm{form}(\tau_A)$. Note that ρ' is also a type for f_A. Suppose that given an input node f, $f/\texttt{parent::}n$ reduces to f_A. This means that f must be one of f_B, f_C, and f_D, which are the child nodes of f_A. To deduce the type of f, first consider the formula part. Any child of f_A matches a formula $\mu Z.\, \langle \bar{1} \rangle\, \varphi_A \vee \langle \bar{2} \rangle\, Z$ which simply states that the context node has a parent matching φ_A. For the regular expression type part, we can deduce from τ_A that any child of f_A matches a regular tree type $\tau_B \mid \tau_C \mid \tau_D$. Finally, we obtain an input type $(\mu Z.\, \langle \bar{1} \rangle\, \varphi_A \vee \langle \bar{2} \rangle\, Z,\ \tau_B \mid \tau_C \mid \tau_D)$.

In general, given an output type ρ, when we infer a parent type $(\varphi, \texttt{element}\ n\ \{\tau\})$ using the inference rules for `self`, τ may be an arbitrary regular expression. Therefore, for the regular expression type part, we compute a child type using an auxiliary function $\mathit{Prime}(\tau)$ [8] which extracts all unit types at the top level of τ and constructs their disjunction. Moreover, if the output type ρ is nullable, then the input node may be a root. In this case, we add a formula φ_{root} to the inferred formula for the input node using a disjunction $\vee$. 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 input node may be a root in the regular tree type part.

4.2. Other Axes

Given an axis other than `self` and `parent` and an output type (φ, u), whereas we can specify the exact shape of the input tree using the formula φ, we cannot using the unit type u because it 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. Still, we do not lose any precision since the formula part of the input type is exact and Invariant 4.1 holds. In other words, for type inference for XPath axes, we can safely ignore the regular tree type part of the inferred input type. Nevertheless, we try to infer a more precise type than AnyElt for the regular tree type part if possible. More precisely, we simply infer AnyElt for `psibl`, `nsibl`, and `desc`, while inferring a more precise type for `child` and `anc`. As studied in [8, 19], in forward type inference systems using only regular tree types as its type language, one can infer precise types only for `self`, `child`, and `desc`. In contrast, in our backward type inference system, we infer precise regular tree types only for `self`, `parent`, and `child` (the formula part is still exact for all XPath axes).

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 axes 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. In order to combine a set of constraints on the input tree, we use an additional judgment of the form

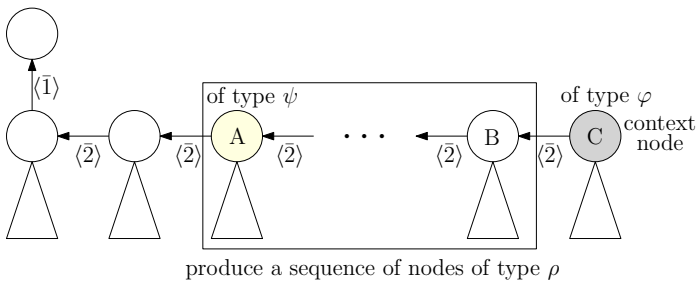


Figure 5: Interpretation of $\varphi \leftarrow$ `psibl::`n, ρ with ψ when the context node is C: A and B are some nodes reached by navigating `psibl` from C.

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

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

Figure 6: Inference rules for `psibl`

$\varphi \leftarrow$ *axis*`::`n, ρ with ψ, 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 $\varphi \leftarrow$ *axis*`::`n, ρ with ψ, consider an example input tree node represented as a binary tree in Figure 5. 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 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, ρ 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:

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

4.2.1. Preceding Sibling

`psibl::`n returns in document order (*i.e.,* pre-order) the 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 6 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.

$$\textsc{Axis-Empty}\quad \frac{}{\psi \leftarrow \mathit{axis}::n,\ \texttt{()}\ \mathsf{with}\ \psi}$$

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

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

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

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

Figure 7: Common inference rules for `psibl`, `anc`, `nsibl`, and `desc`

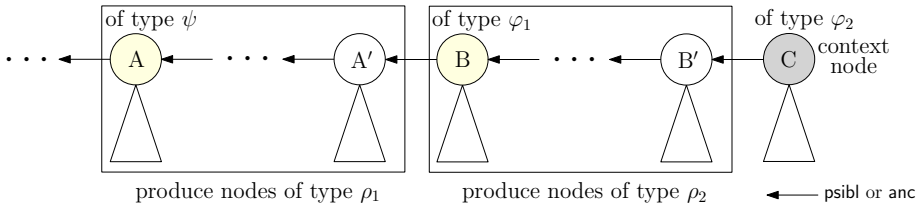


Figure 8: Interpretation of $\varphi \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)\ \mathsf{with}\ \psi$ when *axis* is a backward axis `psibl` or `anc`: we analyze the sequence type from left to right. We first infer φ_1 and then φ_2.

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 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.

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 7. 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). More precisely, as depicted in Figure 8, given a judgment $\varphi \leftarrow \mathit{axis}::n,\ (\rho_1, \rho_2)\ \mathsf{with}\ \psi$, 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

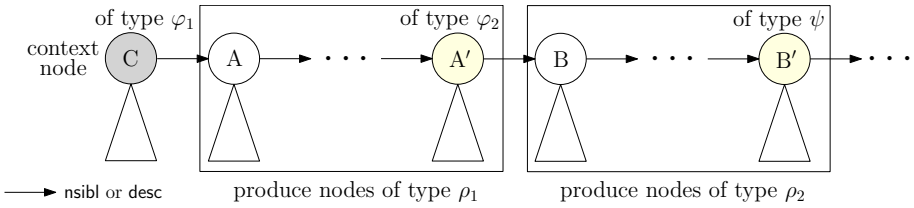


Figure 9: Interpretation of $\varphi \leftarrow axis\texttt{::}n,\ (\rho_1, \rho_2)$ with ψ when *axis* is a forward axis `nsibl` or `desc`: we analyze the sequence type from right to left. We first infer φ_2 and then φ_1.

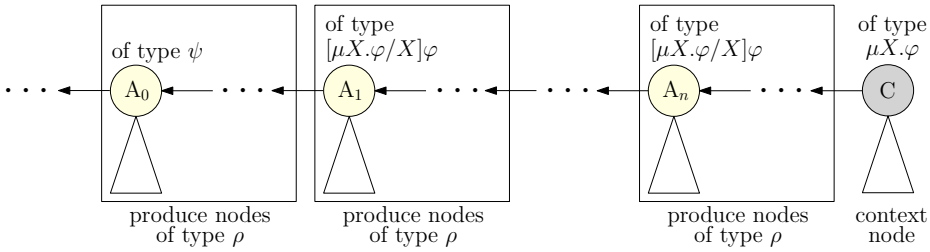


Figure 10: Interpretation of $\varphi \leftarrow axis\texttt{::}n,\ \rho^+$ with ψ when *axis* is a backward axis. A similar illustration can be applied to forward axes.

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 $\varphi_1 \leftarrow axis\texttt{::}n,\ \rho_1$ 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 $\varphi \leftarrow axis\texttt{::}n,\ (\rho_1, \rho_2)$ 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 9, 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 $\varphi \leftarrow axis\texttt{::}n,\ \rho$ with $X \vee \psi$. More precisely, as depicted in Figure 10, 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.

4.2.2. 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 7): 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 explain them in turn.

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

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

$$\begin{array}{rcl}
\texttt{has-anc}(\chi) & = & \mu Z.\, \langle\bar{1}\rangle\, (\chi \vee Z) \vee \langle\bar{2}\rangle\, Z \\
\texttt{desc-type}(()) & = & () \\
\texttt{desc-type}((\varphi, \texttt{element}\ n\ \{\tau\})) & = & \texttt{desc-type}(\tau) \\
\texttt{desc-type}(\rho_1, \rho_2) & = & \texttt{desc-type}(\rho_1) \mid \texttt{desc-type}(\rho_2) \\
\texttt{desc-type}(\rho_1 \mid \rho_2) & = & \texttt{desc-type}(\rho_1) \mid \texttt{desc-type}(\rho_2) \\
\texttt{desc-type}(\rho^+) & = & \texttt{desc-type}(\rho) \\
\texttt{desc-type}(\texttt{element}\ n\ \{\tau\}) & = & \texttt{element}\ n\ \{\tau\} \mid \texttt{desc-type}(\tau) \\
\texttt{desc-type}(\tau_1, \tau_2) & = & \texttt{desc-type}(\tau_1) \mid \texttt{desc-type}(\tau_2) \\
\texttt{desc-type}(\tau_1 \mid \tau_2) & = & \texttt{desc-type}(\tau_1) \mid \texttt{desc-type}(\tau_2) \\
\texttt{desc-type}(\tau^+) & = & \texttt{desc-type}(\tau)
\end{array}$$

Figure 11: Inference rules for `anc`

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

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

Figure 12: Inference rules for `nsibl`

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 5 for `anc` by interpreting the left arrow in the figure as $\langle\bar{1}\rangle$ followed by a possibly empty sequence of $\langle\bar{2}\rangle$.) 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'.

As for the regular tree type part, we use an auxiliary function `desc-type`() which is a recursive version of *Prime*() and computes the type of all possible descendants. Note that for `anc`, the output type is the type of the ancestors of the input context node. In other words, the context node is one of their descendants.

4.2.3. 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

[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.

of rule Axis-Backward-Seq. More precisely, given a judgment $\varphi \leftarrow$ `nsibl::`n, ρ 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 regular tree 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 7 and 12, 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.

4.2.4. Child

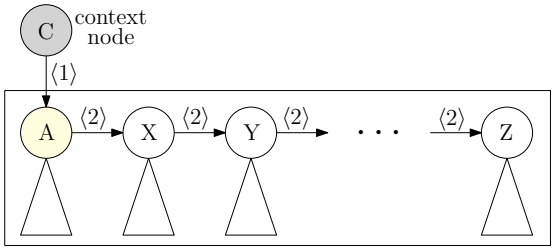
As inference rules for `parent` are defined in terms of those for `self`, rules for `child` can be defined in terms of those for `self-nsibl` (self or next sibling, a variant of `nsibl`, defined in the next subsection). As shown in Figure 13, we first infer a formula φ for `self-nsibl` and then use it 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 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 therefore we use existential modality instead (rule Child-NotNull).

In addition, to infer a unit type for the context node, we use an auxiliary function `parent-type`(ρ), defined in Figure 13, 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 `add-anyelt`(ρ) 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 `parent-type`(ρ) approximates the type of the context node. For example, consider the tree in Figure 13. If only nodes A and Y are returned by C/`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 already encoded in the inferred input formula, 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 5, and thus we do not simply use AnyElt in the regular tree type part of the input type.

4.2.5. 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 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.



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 $\varphi \leftarrow \texttt{self-nsibl::}n,\ \rho$ and φ is true at node A, then $\langle 1\rangle\,\varphi$ is true at node C.

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

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

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

Figure 13: Inference rules for `child`

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

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

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

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

$$\textsc{Axis-Forward-Seq}\quad \frac{\varphi_2 \leftarrow \mathit{axis}\texttt{::}n,\ \rho_2 \text{ with } \psi,\ \mathsf{nullable} \qquad \varphi_1 \leftarrow \mathit{axis}\texttt{::}n,\ \rho_1 \text{ with } \varphi_2,\ \mathsf{nullable} \wedge \mathit{nullable}(\rho_2)}{\varphi_1 \leftarrow \mathit{axis}\texttt{::}n,\ (\rho_1, \rho_2) \text{ with } \psi,\ \mathsf{nullable}}$$

Figure 14: Inference rules for `self-nsibl`

To this end, we introduce a new judgment $\varphi \leftarrow$ `self-nsibl::`n, ρ 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 $\rho \equiv \rho$, `()`.

The nullability parameter is examined only when the output type is a single formula type (φ, u). Consider a judgment $\varphi' \leftarrow$ `self-nsibl::`n, (φ, 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 7 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 14. 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).

4.2.6. Descendant

Like other axes, we use a judgment of the form $\varphi \leftarrow$ `desc::`n, ρ 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 15, 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 follow t_n but precede the leftmost next sibling of the context node must not satisfy name test n. This constraint is encoded in `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 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`$(\varphi \wedge k(n) \wedge \mathrm{form}(u) \wedge \psi,\ k(n))$, which is defined in Figure 15. As for other cases, we simply use the inference rules in Figure 7.

To illustrate, consider an example tree in Figure 16. 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).

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

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

Auxiliary definitions:

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

- `has-desc`(χ): there is a descendant satisfying χ.
- `has-nsdesc`(χ): there is a node satisfying χ which is a next sibling or a descendant of a next sibling.
- `has-prec`(χ): there is a node satisfying χ which precedes the context node in document order and is not an ancestor.
- `has-foll`(χ): there is a node satisfying χ which follows the context node in document order and is not a descendant.
- `noNextUpTo`(χ, α): 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).
- `noWhereElse`(χ): only the context node satisfies χ.
- `fstSelfNsDesc`(χ_1, χ_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.
- `fstFoll`(χ_1, χ_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.
- `fstDescFoll`(χ_1, χ_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 15: Inference rules for `desc`

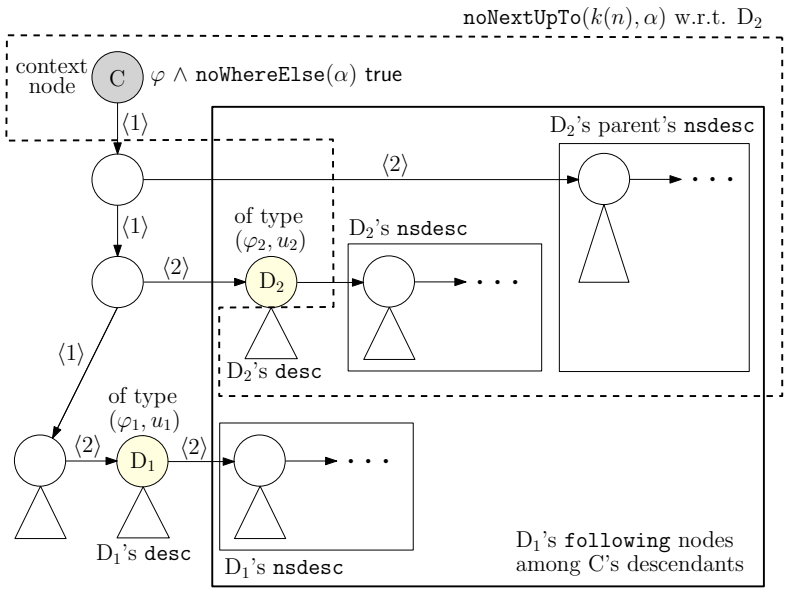


Figure 16: $\varphi \leftarrow$ desc::n, $((\varphi_1, u_1), (\varphi_2, u_2))$ with noNextUpTo$(k(n), \alpha)$ where C is the context node and D_1 and D_2 are the only nodes satisfying name test n, each of which has type (φ_1, u_1) and (φ_2, u_2), respectively. D_1 precedes D_2 in document order.

4.3. Properties of Backward Type Inference for XPath axes

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 4.2 (Exact type inference)**.** *If* $\rho_i \leftarrow$ *axis::*n, ρ_o, *then* $f \in [\![\rho_i]\!]$ *if and only if* $[\![f/\mathit{axis}\text{::n}]\!] \in [\![\rho_o]\!]$.

In Theorem 4.2, 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 4.2, we use the following lemmas for the auxiliary judgment $\varphi \leftarrow$ *axis*::n, ρ with ψ.

Lemma 4.3 (Soundness)**.** *Suppose*

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

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 4.3 is a one-way formalization of the interpretation of the judgment $\varphi \leftarrow$ *axis*::n, ρ with ψ. To illustrate, consider Figure 5 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 4.3.

Proof of Lemma 4.3. 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 4.4 (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 $\varphi \leftarrow \mathit{axis}::n,\ \rho$ with ψ. Here, we only show the case where $\rho = \rho_1, \rho_2$ and *axis* is a backward axis.

(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) $\exists\ i \leq j \leq n$ *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) $[\![f_j / \mathit{axis}::n]\!] = f_1, \ldots, f_{j-1}$
(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)

□

5. Inference for the 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 where free and bound variables are defined in the usual way. Our goal is then to design inference rules that ensure that if we substitute those free variables with any sequences of focused trees satisfying one of the 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 thus 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 5.1. 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:

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

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 seamlessly be added into the external language [27].

5.1. Inference Rules

Figures 17 and 18 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 first be 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 the 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 $\varphi_{root} = \neg\langle\bar{1}\rangle\top \wedge \neg\langle\bar{2}\rangle\top \wedge \neg\langle 2\rangle\top$ 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. Finally, we infer input constraints from the body expression e which reduces to a sequence of child nodes and from the output type form-enriched(τ) for the child nodes. We use form-enriched(τ) to support context-erasing element construction: during the reduction, we remove the context of the result of e (see Figure 3). form-enriched(τ) enriches the given regular tree type τ by simply associating each unit type u that appears in τ with an equivalent downward-only formula 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 single(ρ) 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 $\langle\!\langle \varphi_{root} \wedge \text{form}(u) \wedge \neg\text{single}(\rho)\rangle\!\rangle = \emptyset$ which can be tested in $2^{O(|u|+|\rho|)}$ time by the decision procedure in [26].

As for if-expressions, rules I-IfNonEmpty and I-IfEmpty respectively consider the cases where the condition expression e_1 always reduces to a non-empty sequence and an empty sequence, regardless of the input trees. If e_1 can reduce to both a non-empty sequence and an empty sequence, depending on the input trees, then we use the rule I-IfAny. It simply assumes that e_1 reduces to any sequence and infers a constraint $\mathscr{S}_i$ from each subexpression e_i. If all of $\mathscr{S}_1$, $\mathscr{S}_2$, and $\mathscr{S}_3$ are satisfiable, then the if-expression has the specified output type ρ.

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 satisfiable, *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.

For a sequence concatenation, rule I-Seq divides the output type ρ into two parts using an auxiliary function split(), defined in Figure 18. 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, let us consider for-loop expressions, which are the most challenging with respect to defining precise inference rules. Indeed, they are the main source of the approximation introduced in our backward type inference. To illustrate, consider the following expression:

$$\texttt{for}\ \$v\ \texttt{in}\ \$doc\texttt{/desc::}D\ \texttt{return}\ \$v\texttt{/child::}*$$

where $\$doc$ is bound to an input tree. If $\$doc$`/desc::`$D$ reduces to $[f_1, \ldots, f_n]$ for some n, then the whole expression reduces to f_1`/child::`$*, \ldots, f_n$`/child::`$*$. Suppose that an output type ρ_o is given as follows:

$$\rho_o \quad \equiv \quad \texttt{<}A\texttt{/><}B\texttt{/><}C\texttt{/><}A\texttt{/><}B\texttt{/><}C\texttt{/><}A\texttt{/><}B\texttt{/><}C\texttt{/>}$$

where for simplicity we use `<`A`/>` to denote a formula type ($\top$, `element` A $\{$`()`$\}$) and juxtaposition to denote a sequence concatenation. In order to infer the exact type of $\$doc$, we need to infer the exact type of each f_i. Since the output sequence type ρ_o is finite, it suffices to compute all possible (weak) compositions of

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

$$\textsc{I-FVar}\quad \frac{\rho' \leftarrow \texttt{self::}*,\ \rho}{\{\{(\$v : \rho')\}\} \leftarrow \$v : \rho} \qquad \textsc{I-LVar}\quad \frac{}{\{\{(\$\overline{v} : \rho)\}\} \leftarrow \$\overline{v} : \rho} \qquad \textsc{I-Axis}\quad \frac{\rho' \leftarrow \mathit{axis}\texttt{::}n,\ \rho}{\{\{(\$v : \rho')\}\} \leftarrow \$v/\mathit{axis}{::}n : \rho}$$

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

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

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

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

Auxiliary definitions:

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

Figure 17: Backward type inference rules for the XQuery core

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

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

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

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

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

Auxiliary definitions:

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

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

Figure 18: Backward type inference rules for the XQuery core, continued

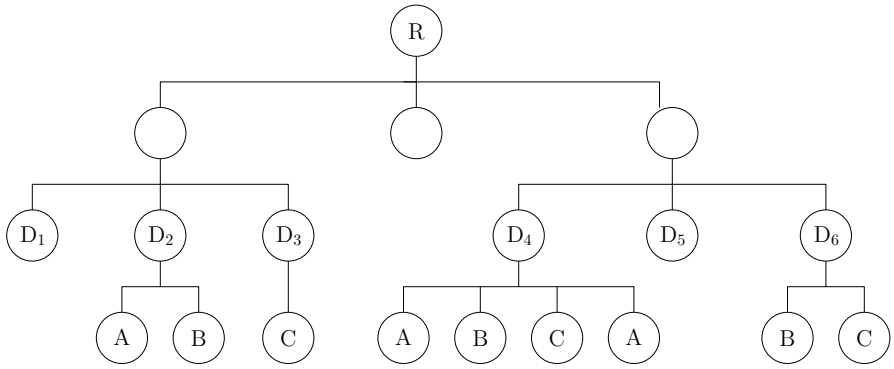


Figure 19: An example XML tree bound to the variable $\$doc$: a for-loop expression `for` $\$v$ `in` $\$doc$/`desc::`$D$ `return` $\$v$/`child::`$*$ reduces to a sequence of focused tree nodes $[A[\epsilon], B[\epsilon], C[\epsilon], A[\epsilon], B[\epsilon], C[\epsilon], A[\epsilon], B[\epsilon], C[\epsilon]]$ where we omit the context of each node for simplicity. In addition, the for-loop expression may have type (<A/><B/><C/>)$^+$, or less precisely (<A/> | <B/> | <C/>)*.

ρ_o, infer an input constraint for each composition, and take a union of all inferred constraints [28]. We say that $(\rho_1; \ldots; \rho_n)$ is a composition of ρ if $(\rho_1, \ldots, \rho_n) <: \rho$. A composition is said to be *weak* if it contains an empty sequence type as an element. For example, we can infer the exact type of the example input tree given in Figure 19 from the following weak composition:

$$(\texttt{()};\ \texttt{<}A\texttt{/><}B\texttt{/>};\ \texttt{<}C\texttt{/>};\ \texttt{<}A\texttt{/><}B\texttt{/><}C\texttt{/><}A\texttt{/>};\ \texttt{()};\ \texttt{<}B\texttt{/><}C\texttt{/>})$$

For this input tree, $\$doc/\texttt{desc::}D$ reduces to $[f_1, \ldots, f_6]$ where each f_i is a focused tree rooted at the node labeled D_i (here, the subscript i is not part of the node label; it is used solely to distinguish the nodes with the same label). Note that the composition above consists of the exact type of each $f_i/\texttt{child::}*$, given the output type ρ_o. Assume that ρ_i is inferred from $f_i/\texttt{child::}*$ and the corresponding type in the composition given above. Then, our backward type inference infers the following exact input type for the whole for-loop expression (with some simplification):

$$\left(\begin{array}{l}\texttt{fstDescFoll}(\rho_1 \wedge \texttt{fstDescFoll}(\rho_2 \wedge \texttt{fstDescFoll}(\rho_3 \wedge \texttt{fstDescFoll}(\rho_4 \wedge \texttt{fstDescFoll}(\rho_5 \wedge \\ \qquad \texttt{fstDescFoll}(\rho_6 \wedge \texttt{noNextUpTo}(D, \alpha), D), D), D), D), D), D) \wedge \texttt{noWhereElse}(\alpha), \mathsf{AnyElt}\end{array}\right)$$

which states that the input node has six descendants with label D and the first descedant in document order is of type ρ_1, the second is of type ρ_2, and so on.

The situation becomes more complex if we consider repetition types. In the presence of repetition operators, the number of possible compositions is infinite in general. To illustrate, consider an output type ρ'_o defined as follows:

$$\rho'_o \quad \equiv \quad (\texttt{<}A\texttt{/><}B\texttt{/><}C\texttt{/>})^+$$

To infer the exact type of the input tree in Figure 19 again, we need to unfold ρ'_o three times to obtain ρ_o, which is defined above, and compute its weak compositions. The problem is that in general we do not statically know how many times we need to unfold the given output repetition type to infer the exact input type for a for-loop expression. One possible solution is to unfold repetition types up to some arbitrary fixed number of times, giving up exact typing. Then the problem is to find such a unfolding number that allows practical and precise type inference.

In this paper, we adopt a simpler but more approximate approach. More precisely, we do not analyze output types of the form ρ^+ or ρ^* across the boundary of ρ, that is, we do not unfold repetition types. In other words, given a for-loop expression, we simply consider only those 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. Consequently, the input tree in Figure 19 is not accepted by our type system if the output type is given as $(\texttt{<}A\texttt{/><}B\texttt{/><}C\texttt{/>})^+$. Our system accepts only those input trees whose descendants labeled D have no child or children of type $(\texttt{<}A\texttt{/><}B\texttt{/><}C\texttt{/>})^+$. Nevertheless, the input tree in Figure 19 is accepted if a more general output type is given such as $(\texttt{<}A\texttt{/>} \mid \texttt{<}B\texttt{/>} \mid \texttt{<}C\texttt{/>})^*$.

Our approximation is similar in spirit to the approximation used in forward type inference systems [8, 17]. To illustrate, assume a regular tree type T defined recursively as $\texttt{<}C\texttt{/>}[\texttt{<}A\texttt{/>}, T, \texttt{<}B\texttt{/>}] \mid \texttt{()}$. When f is of type T, the type of $f/\texttt{desc-or-self::}*$ is deduced as $(\texttt{<}A\texttt{/>} \mid \texttt{<}B\texttt{/>} \mid \texttt{<}C\texttt{/>})^*$ in forward type inference systems. The exact type, however, is the union of $(\texttt{<}C\texttt{/><}A\texttt{/>})^n\texttt{<}B\texttt{/>}^n$ for $n \in \mathbb{N}$, which is not regular. In general, in forward type inference systems, a sequence type $(u_1, \ldots, u_n)^*$ is often approximated into a less precise type $(u_1 \mid \ldots \mid u_n)^*$, losing the information on the order of elements. Similarly, in our backward type inference system, in order to accept more input trees, output types should be given such that the order of elements does not matter.

To infer an input type for for-loop expressions, we use three rules. First, given an expression `for` $\$v$ `in` e_1 `return` e_2, if the output type is `()`, then no matter how many times we evaluate e_2 with different bindings for $\$v$, it must 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 ρ is not `()`. Rule I-ForNonEmpty is similar to the rule I-ForEmpty except that it uses an auxiliary function $Quant()$ 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), then 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 should reduce to a sequence of focused trees of type ρ^+ and their concatenation should also be 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 *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 $[\![C(\$v)]\!] \neq \emptyset$.

5.2. Complexity

5.2.1. 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:

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

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 [2].

Lemma 5.2. *The time complexity of the auxiliary functions directly used in the inference rules, introduced in Section 4, is as follows.*

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

`child-type`(ρ) in Figure 4 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 5.2, 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 5.3. *Given output type ρ, an input type for an XPath axis is inferred in $O(|\rho|)$ 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 $O(|\rho|)$. □

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 inferred input formula $\varphi_1 \vee \varphi_2$: one in φ_1, the other in φ_2.

Lemma 5.4. *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 15, which take a formula χ as argument, return another formula of size $O(|\chi|)$. The proof also relies on that form(u) has the same size as the classical binary representation of the regular tree type u [19]. □

Lemma 5.5. *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`, `child`, and `anc` are easily proved by Lemma 5.4. The case for `self` is proved by structural induction on output type ρ. The cases for `parent`, `child`, and `anc` are proved by the fact that the size of `child-type`(ρ), `parent-type`(ρ), and `desc-type`(ρ) is $O(|\rho|)$ with optimized representations of types. □

Corollary 5.6. *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 [26].*

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.

5.2.2. Complexity for the 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 [8].

Lemma 5.7. *Suppose $\mathscr{S} \leftarrow \mathrm{e} : \rho$. Then the maximum size, denoted by $T(\mathrm{e}, \rho)$, of a 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:

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

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

Lemma 5.8. *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 5.7.

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

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

Lemma 5.9. *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 5.7 and 5.8. We also use a type and its size interchangeably as the second argument to $I(-,-)$.

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

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

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

Theorem 5.10 (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 $\mathscr{S}$ in $2^{O(2^{|e|}|\rho|)}$ time by Lemma 5.9. The size of $\mathscr{S}$ is $O(2^{2^{|e|}|\rho|})$ by Lemma 5.8. 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 5.7, 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 [26]. 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 $2^{O(2^{(|e|+1)}|\rho|)}$. □

5.3. Soundness

Now we state the soundness property for 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 5.11 (Soundness)**.** *Let* e *and ρ be an XQuery expression and its output type, respectively. Suppose $\mathscr{S} \leftarrow \mathrm{e} : \rho$. 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) $s = \Pi_{f_1,\ldots,f_n} [\![e_2]\!]_{\eta,\$v \mapsto f_i}$

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 for-loop expressions. From the soundness and the decidability of the inference system, we deduce a sound typechecking algorithm as a corollary.

Corollary 5.12 (Typechecking)**.** *Let* e *be an XQuery expression with the 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 type ρ_i and output type 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 $C(\$doc)$.

6. Related Work and Discussion

6.1. Typechecking for XML Transformations

The problem of typechecking XML transformations has been extensively studied since the introduction of XML. There are two major approaches, namely forward type inference and backward type inference. Given an expression e that transforms XML documents of type ρ_i into documents of type ρ_o, forward type inference 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 $O \subseteq \rho_i$. In contrast, backward type inference 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 $\rho_i \subseteq I$. When types are modeled as regular tree languages, exact typechecking may be done in the form of backward type inference by using tree transducers as a model of XML transformations [10, 11, 12]. In contrast, in forward type inference, even for simple XML transformations, their image may not be regular, as illustrated in Section 5.1, and therefore typechecking cannot be exact. Still, forward type inference is more intuitive than backward type inference, and thus many practical XML programming languages such as XQuery [1, 3], XDuce [22], and CDuce [29] build on forward type inference and instead introduce some approximation, *i.e.,* some type-safe programs are rejected in these languages. For a more detailed, general survey of typechecking for XML transformations, we refer the reader to [30, 31] and references therein. Below we discuss only closely related work on backward type inference and precise type systems for XPath and XQuery.

6.2. Inverse Type Inference

A problem of *inverse type inference*, which is another name of backward type inference, has been extensively investigated to develop an exact typechecking algorithm for XML transformations [9, 10, 11, 12, 13, 14]. For example, Milo *et al.* [10] propose an exact inverse type inference algorithm for *k-pebble tree transducers*, which are finite state tranducers that can mark nodes of the input tree using up to k different pebbles. Although we can model a broad range of XQuery expressions using k-pebble tree transducers, the complexity of typechecking is hyper-exponential, *i.e.,* when using k pebbles, its complexity is $O(h_{k+2}(n))$ with $h_0(n) = n$ and $h_{m+1}(n) = 2^{h_m(n)}$.

Maneth *et al.* [12] also study the problem of exact inverse type inference for tree transformations using macro tree transducers (MTTs) [15], which can accumulate part of the input and copy it in the output. Their transformation language called TL uses monadic second-order logic (MSO) as a pattern language, which subsumes XPath without arithmetics and data value comparisons. By using MTTs and MSO, TL can be used to describe many real-world XML transformations. 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.

Perst and Seidl [11] extend MTTs with concatenation and propose macro forest transducers (MFTs) as a model of XML transformations. They develop an exact inverse type inference algorithm for MFTs and show that the complexity of typechecking is DEXPTIME-complete. Morever, by combining with a translation from a downward navigational fragment of XQuery into MFTs [32], MTFs can be used as an intermediate language for a subset of XQuery. The translation, however, considers only XPath axes such as `child`, `desc`, and `following-sibling`, and a restricted form of for-loop expressions, *i.e.,* in `for` $\$v$ `in` e_1 `return` e_2, e_1 must be a path expression.

In order to support backward axes, one may have to use tree-walking automata [33] as a pattern language. Indeed, a k-pebble tree transducer can be decomposed into a $(k+1)$-fold composition of tree-walking transducers [16]. Similarly, a TL program using MSO patterns can be compiled into a composition of a MTT and a macro tree-walking transducer, which can then be decomposed into a three-fold composition of (stay) MTTs [12]. Therefore, the complexity of typechecking a TL program is quadruple exponential even if we do not consider a possible blow-up in the translation of MSO patterns to finite automata. In this paper, we also study the problem of backward type inference, but develop a type inference system directly on the XQuery core. We present an exact backward type inference algorithm for XPath axes whose complexity is simple exponential. This result corresponds to the fact that the complexity of inverse type inference for tree-walking transducers is also exponential [16]. As for the XQuery core, instead of trying to develop yet

another hyper-exponential algorithm, we introduce a sound approximation similar to the one used in forward type systems.

6.3. Precise Type Systems for XPath and XQuery

Typing XPath expressions has been a challenging topic. Most previous proposals for the XQuery static type system, including the one standardized by the W3C [3], support only downward navigation in XML trees. As thoroughly discussed in [19], it is mainly due to the discrepancy between the XML data model and the type model, namely regular tree types [2]. Since XPath backward axes are the main source of difficulty, one may want to translate XPath selection queries with backward axes into equivalent queries with only forward axes. Olteanu *et al.* [34, 35] propose such translations which generate a query containing the same number of joins, *i.e.,* identity-based equality, as reverse steps or a query without joins but of exponential size. The translations, however, are defined only for XPath and it is unclear how to extend them to deal with XQuery-like languages. Møller *et al.* [36] propose static typechecking for XSLT [37] programs which builds on a context-sensitive flow analysis. Although they introduce some approximation for abstract evaluation of XPath axes with respect to DTDs, they experimentally validate using a number of benchmarks that their algorithm is highly precise. Benzaken *et al.* develop a precise type inference system for XPath in their work on type-based XML projection [38]. Their system handles backward axes and is also sound and complete for a particular class of regular tree types that are $*$-guarded, non-recursive, and parent-unambiguous. In contrast, our inference system for XPath axes is exact with no such restrictions on types.

Benedikt and Cheney [39] propose a type system for the XQuery Update Facility language [40] assuming the existence of a sound typechecker for XPath axes. In the work on independence analysis of XML queries and updates [41], they use satisfiability solvers [20, 42, 43] to decide disjointness of selection queries, which may contain backward axes. In [42, 43], weak monadic second-order logic of two successors (WS2S) [44] is used, which is one of the most expressive decidable logic when both regular tree types and XML queries are considered. However, the satisfiability problem for WS2S is known to be non-elementary. Our work is based on the same tree logic and its associated satisfiability solver used in [20]. The main difference is that while [20] considers only XPath, we consider a core fragment of XQuery including element construction. Moreover, while in [20] values are defined as sets of nodes, in this work they are defined as sequences of nodes which may come from different trees and also retain their original tree context independently for navigation.

Recently, Castagna *et al.* [17] and Genevès and Gesbert [19] independently propose an extended type language to describe not only a given XML tree node but also its context. In [17], the authors extend the core calculus of CDuce [29] with zipper data structures [18], 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 typecheck such navigational expressions 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 [7, 45], 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 [17], 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 [46], we expect that our type system can be easily extended with (higher-order) functions at least in theory.

This work builds on our previous work [19] which proposes 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 supports all the major navigational features of the XQuery core. The main difference is that while we use forward inference in [19], we use backward inference in this work. 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 for-loop expressions are simpler in [19]), but as a trade-off it provides an exact typechecking algorithm for XPath axes. Another difference is that while we use a small-step operational semantics for the XQuery core in [19], we use a denotational semantics in this work because it is more suitable for proving properties of our backward type inference. Considering all these aspects, it would be quite interesting to combine the two approaches.

7. Conclusion

In this paper, we propose a novel backward type inference system for XQuery as a complementary method to forward type inference. Specifically, the contributions of the paper are 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 backward type inference system for the XQuery core, with a characterized complexity.

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 for-loop expressions using forward type inference, thus obtaining a lower complexity than our backward approach, while typechecking XPath axes using backward type inference, thus obtaining better precision than the forward approaches such as in [17, 19]. In doing so, one possible difficulty would be to find minimal type annotations to enable effective bidirectional typechecking.

Acknowledgement

H. Im was supported in part by 2015 Research Grant from Kangwon National University and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2016R1C1B1015095).

References

[1] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, J. Siméon, XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation (December 2010).
URL `https://www.w3.org/TR/2010/REC-xquery-20101214/`

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

[3] D. Draper, M. Dyck, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, P. Wadler, XQuery 1.0 and XPath 2.0 Formal Semantics (second edition), W3C Recommendation (December 2010).
URL `http://www.w3.org/TR/xquery-semantics/`

[4] J. Clark, S. DeRose, XML Path Language (XPath) Version 1.0, W3C Recommendation (November 1999).
URL `https://www.w3.org/TR/1999/REC-xpath-19991116/`

[5] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon, XML Path Language (XPath) 2.0 (Second Edition), W3C Recommendation (December 2010).
URL `http://www.w3.org/TR/xpath20`

[6] J. Robie, D. Chamberlin, M. Dyck, J. Snelson, XML Path Language (XPath) 3.0, W3C Recommendation (April 2014).
URL `http://www.w3.org/TR/xpath-30/`

[7] J. Robie, D. Chamberlin, M. Dyck, J. Snelson, XQuery 3.0: An XML Query Language, W3C Recommendation (April 2014).
URL `http://www.w3.org/TR/xquery-30/`

[8] D. Colazzo, C. Sartiani, 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, 2011, pp. 89–100. `doi:10.1145/2003476.2003490`.

[9] A. Tozawa, Towards static type checking for XSLT, in: ACM Symposium on Document Engineering, 2001, pp. 18–27.

[10] T. Milo, D. Suciu, V. Vianu, Typechecking for XML transformers, Journal of Computer and System Sciences 66 (1) (2003) 66–97. doi:10.1016/S0022-0000(02)00030-2.

[11] T. Perst, H. Seidl, Macro forest transducers, Inf. Process. Lett. 89 (3) (2004) 141–149. doi:10.1016/j.ipl.2003.05.001.

[12] S. Maneth, A. Berlea, T. Perst, H. Seidl, 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, 2005, pp. 283–294. doi:10.1145/1065167.1065203.

[13] A. Tozawa, XML type checking using high-level tree transducer, in: Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS '06, 2006, pp. 81–96. doi:10.1007/11737414_7.

[14] A. Frisch, H. Hosoya, Towards practical typechecking for macro tree transducers, in: Proceedings of the 11th International Conference on Database Programming Languages, DBPL'07, Springer-Verlag, 2007, pp. 246–260.

[15] J. Engelfriet, H. Vogler, Macro tree transducers, Journal of Computer and System Sciences 31 (1) (1985) 71–146. doi:10.1016/0022-0000(85)90066-2.

[16] J. Engelfriet, The time complexity of typechecking tree-walking tree transducers, Acta Informatica 46 (2) (2009) 139–154. doi:10.1007/s00236-008-0087-y.

[17] G. Castagna, H. Im, K. Nguyen, V. Benzaken, A core calculus for XQuery 3.0: Combining navigational and pattern matching approaches, in: Proceedings of the 24th European Symposium on Programming, 2015, pp. 232–256. doi:10.1007/978-3-662-46669-8_10.

[18] G. Huet, The zipper, J. Funct. Program. 7 (5) (1997) 549–554. doi:10.1017/S0956796897002864.

[19] P. Genevès, N. Gesbert, XQuery and static typing: Tackling the problem of backward axes, in: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, 2015, pp. 88–100. doi:10.1145/2784731.2784746.

[20] P. Genevès, N. Layaïda, A. Schmitt, Efficient static analysis of XML paths and types, in: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, 2007, pp. 342–351. doi:10.1145/1250734.1250773.

[21] N. Alon, T. Milo, F. Neven, D. Suciu, V. Vianu, XML with data values: typechecking revisited, Journal of Computer and System Sciences 66 (4) (2003) 688–727. doi:10.1016/S0022-0000(03)00032-1.

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

[23] M. Murata, D. Lee, M. Mani, K. Kawaguchi, Taxonomy of XML schema languages using formal language theory, ACM Trans. Internet Technol. 5 (4) (2005) 660–704. doi:10.1145/1111627.1111631.

[24] R. M. Amadio, L. Cardelli, Subtyping recursive types, ACM Trans. Program. Lang. Syst. 15 (4) (1993) 575–631. doi:10.1145/155183.155231.

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

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

[27] A. Frisch, G. Castagna, V. Benzaken, Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types, J. ACM 55 (4) (2008) 19:1–19:64. doi:10.1145/1391289.1391293.

[28] S. Heubach, T. Mansour, Combinatorics of Compositions and Words, Chapman and Hall/CRC, 2009.

[29] V. Benzaken, G. Castagna, A. Frisch, CDuce: An XML-centric general-purpose language, in: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP '03, 2003, pp. 51–63. doi:10.1145/944705.944711.

[30] A. Møller, M. I. Schwartzbach, The design space of type checkers for XML transformation languages, in: Proceedings of the 10th International Conference on Database Theory, ICDT '05, Springer-Verlag, 2005, pp. 17–36. doi:10.1007/978-3-540-30570-5_2.

[31] V. Benzaken, G. Castagna, H. Hosoya, B. C. Pierce, S. Vansummeren, XML typechecking, in: Encyclopedia of Database Systems, Springer US, 2009, pp. 3646–3650. doi:10.1007/978-0-387-39940-9_788.

[32] S. Hakuta, S. Maneth, K. Nakano, H. Iwasaki, XQuery streaming by forest transducers, in: IEEE 30th International Conference on Data Engineering, ICDE '14, 2014, pp. 952–963. doi:10.1109/ICDE.2014.6816714.

[33] A. Aho, J. Ullman, Translations on a context free grammar, Information and Control 19 (5) (1971) 439–475. doi:10.1016/S0019-9958(71)90706-6.

[34] D. Olteanu, H. Meuss, T. Furche, F. Bry, XPath: Looking forward, in: XML-Based Data Management and Multimedia Engineering — EDBT 2002 Workshops, 2002, pp. 109–127. doi:10.1007/3-540-36128-6_7.

[35] D. Olteanu, Forward node-selecting queries over trees, ACM Trans. Database Syst. 32 (1). doi:10.1145/1206049.1206052.

[36] A. Møller, M. O. Olesen, M. I. Schwartzbach, Static validation of XSL transformations, ACM Trans. Program. Lang. Syst. 29 (4). doi:10.1145/1255450.1255454.

[37] J. Clark, XSL Transformations (XSLT) Version 1.0, W3C Recommendation (November 1999).
URL https://www.w3.org/TR/1999/REC-xslt-19991116

[38] V. Benzaken, G. Castagna, D. Colazzo, K. Nguyen, Optimizing XML querying using type-based document projection, ACM Trans. Database Syst. 38 (1) (2013) 4.

[39] M. Benedikt, J. Cheney, Semantics, types and effects for XML updates, in: Proceedings of the 12th International Symposium on Database Programming Languages, DBPL '09, Springer-Verlag, 2009, pp. 1–17. doi:10.1007/978-3-642-03793-1_1.

[40] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, J. Siméon, XQuery Update Facility 1.0, W3C recommendation (March 2011).
URL http://www.w3.org/TR/xquery-update-10/

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

[42] N. Klarlund, A. Møller, MONA Version 1.4 User Manual, BRICS (January 2001).

[43] P. Genevès, N. Layaïda, Deciding XPath containment with MSO, Data Knowl. Eng. 63 (1) (2007) 108–136. doi:10.1016/j.datak.2006.11.003.

[44] J. Doner, Tree acceptors and some of their applications, Journal of Computer and System Sciences 4 (5) (1970) 406 – 451. doi:10.1016/S0022-0000(70)80041-1.

[45] M. Benedikt, H. Vu, Higher-order functions and structured datatypes, in: Proceedings of the 15th International Workshop on the Web and Databases, WebDB 2012, 2012, pp. 43–48.

[46] N. Gesbert, P. Genevès, N. Layaïda, Parametric polymorphism and semantic subtyping: The logical connection, in: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11, 2011, pp. 107–116. doi:10.1145/2034773.2034789.