XQTC: A Static Type-Checker for XQuery

Using Backward Type Inference

Pierre Genevès
CNRS*
pierre.geneves@inria.fr

Nabil Layaïda
INRIA
nabil.layaida@inria.fr

Christine Vanoirbeek
EPFL
christine.vanoirbeek@epfl.ch

ABSTRACT

We present a novel technique and a tool for static type-checking of XQuery programs. The tool looks for errors in the program by jointly analyzing the source code of the program, input and output schemas that respectively describe the sets of documents admissible as input and as output of the program. The crux and the novelty of our results reside in the joint use of backward type inference and a two-way logic to represent inferred tree type portions. This allowed us to design and implement a type-checker for XQuery which is more precise and supports a larger XQuery fragment compared to the approaches previously proposed in the literature; in particular compared to the only few actually implemented static type-checkers such as the one in Galax [6]. The whole system uses compilers and a satisfiability solver for deciding containment for two-way regular tree expressions. Our tool takes an XQuery program and two schemas S_{in} and S_{out} as input. If the program is found incorrect, then it automatically generates a counter-example valid w.r.t. S_{in} and such that the program produces an invalid output w.r.t S_{out}. This counter-example can be used by the programmer to fix the program.

1. INTRODUCTION

One of the most essential tasks in web engineering consists in extracting and then transforming data for generating output (for display or for further processing). Since the inception of XML, dedicated programming languages such as XSLT and XQuery were designed specifically to deal with such transformations. Several research papers have studied the properties of these languages in order to better understand the difficulty they introduce and to master their inherent complexity. In particular, the problem of finding errors statically is of primary importance. Errors can be easily introduced because transformations involve complex information extraction via powerful query languages such as XPath combined with imperative style statements (the so-called FLWOR expressions) containing instructions such as conditionals, loops, etc. combined with data producing instructions. Furthermore, data given as input or expected as output may additionally be constrained by some type (schema). This makes errors even harder to detect since types are specified independently and provided externally to the program. This led W3C, the standard body in charge of defining the XQuery language [3], to include a type system in the specification [5]. This type system allows type information to be effectively propagated to corresponding instructions in the program with the aim of detecting errors at compile time. The ultimate benefit of such analysis is to avoid unexpected runtime errors by proving statically that the program is correct for all input documents. This has performance implications since programs proved correct do not need to validate their output dynamically. The process of detecting errors at compile-time, known as static type checking, typically involves two main operations: type inference and type containment test. The containment test consists in checking that the program produces an XML output which is consistent with the expected output type when applied to a document which is valid with respect to the input type. This is done by comparing the inferred type with the expected type. Effective algorithms for testing containment between types have been developed in the literature [11]. They are typically based on tree automata containment, and they proved effective in practice. Type inference is often tricky for expressive languages and constitutes the central problem in type checking transformations.

Forward Type Inference and its Limitations

The vast majority of works in the literature perform forward type inference. Forward type inference consists in calculating the type describing the actual program output, based on the analysis of the program and the input type. This analysis consists in applying typing rules repeatedly to each program statement and combining the results so that the analysis of the entire program yields a single and global inferred output type. This approach is intuitive since the program produces its output in a top-down fashion, and grammars also describe constraints in a top-down forward-only fashion with respect to the parent-child relationship. However, this approach has some fundamental and intrinsic limitations. Notably, one revealing limitation is the lack of support for backward XPath axes. This is a fundamental limitation because the typical forward inference process navigates in the input tree grammar, using child or descendant axes in order

*This work was partly done during a visit at EPFL.

to infer an output type. This navigation in a type is possible (and relatively easy) since tree grammars descriptions are forward-only. However, navigating backward in a tree grammar is very difficult to achieve and to model because there is no guarantee that the target nodes correspond to non-terminals of the (forward-only) grammar. This is why forward type inference is intrinsically tied to forward XPath axes.

Lack of support of backward XPath axes.

Backward XPath axes such as `parent` or `ancestor` are commonly used in XQuery programs found in practice. Even more remarkably, frequent patterns such as

```
child::p[position()=1]
```

(used to pick the first element labeled "`p`" in a sequence) happen to be syntactic sugars for expressions involving backward axes. For example, the above pattern is a shorthand for:

```
child::p[not(preceding-sibling::p)]
```

More generally, this lack of support is unfortunate since one of the main arguments in favor of using languages such as XQuery (or XSLT[1]) concerns their ability to rely on the full power of XPath, which constitutes the essential core around which those languages are built.

Concretely, current type systems abstract over backward axes by inferring the type "any" (no constraint at all) as soon as a backward axis such as `parent` or `preceding-sibling` is found in the program. This gross approximation leads to unsound type-checkers in the context of forward type inference. This is because the inferred type is much less constrained than it would need to be. Therefore, the final containment check between the inferred type and the expected type will more likely fail. An important consequence is that type-checkers based on this approach will report many false errors for correct programs. Reporting false negatives is problematic because of the useless time spent in debugging a correct program.

Backward Type Inference and its Advantages

In this paper we propose another, fundamentally different approach based on backward type inference. Backward type inference consists in inferring the input type modeling the set of admissible input documents so that the program generates a valid output. For this purpose, the analysis uses the program and an expected type as output. Once the inferred input type is obtained, a containment check is performed to test whether this inferred type is included in the input type for documents actually given as input to the program. We claim that this approach offers several decisive advantages that we briefly review below.

Inferring exact constraints for the `for` *loop.*

One of the main advantages concerns the typing of a problematic XQuery statement: the "`for` $v `in` e_1 `return` e_2" loop. In the context of forward type inference, typing precisely the `for` loop amounts to inferring information on how many times the return part e_2 will be executed. The W3C type system, designed to have a polynomial complexity, infers an approximate type.

[1]XSLT can be compiled into XQuery [7].

For example, consider `for` $v `in` $db `desc::`$n `return` $v where the input type and the expected output type are both b, c, b^*. The W3C type inference yields $(b \mid c)^*$. As highlighted by Colazzo and Sartiani [4], this is a gross approximation which is obviously not contained in b, c, b^*. The type-checker will thus raise an error while the program is perfectly correct. Colazzo and Sartiani go further and propose another type-system, designed to be more precise (for an exponential complexity). Basically, they perform a case analysis on e_1 expression in order to infer some more precise information on how many times the return clause e_2 will be executed. In the same manner, we could recursively perform more sub-decomposition in order to generate even more precise information (at the cost of an even greater complexity). At some point, one has to choose a trade-off between complexity and precision (necessarily approximate).

Backward type inference brings a radically different perspective to solve this problem. It allows to completely overcome this difficulty by reading the output type and basically copying selected portions of the output type to the inferred input type. A major advantage of looking at the `for` loop the other way around is that it permits to completely avoid approximation. Using backward type inference, we can type the `for` loop in an exact manner without any additional cost. For the above example, b, c, b^* is directly (and linearly) inferred as input.

Supporting backward navigation.

Another decisive advantage of our approach is that we can infer all structural constraints induced by backward and recursive navigation, and transfer them precisely in the inferred input type. This is a crucial part of our contribution, further explained in Sections 3.3 and 3.2.

Sound type-checkers even with approximation.

In both forward and backward type inferences, obviously at some point, one has to choose a trade-off between precision and complexity. However, this choice has a fundamentally different price in each technique. Backward type inference offers a decisive advantage here: an approximation in the inference process will always preserve containment and therefore soundness, unlike in the forward type inference approach. Approximations in the forward type inference introduce false negatives (as in the example above). The less precise is the inferred output type, the greater is the number of false negatives reported. In contrast, approximation in the backward type inference case lead to less errors detected but all of those reported will be real ones. Therefore, we believe that performance/precision trade-offs are much more relevant in the backward type inference.

Contribution

The contribution of our paper is threefold:

- we propose the joint use of backward type inference and a two-way logic as the best technique for the static type-checking of XQuery;
- we present the first algorithm and tool that implement this approach;
- we obtain more precision in type-checking than in previous approaches and we support a larger fragment of

XQuery (notably we are the first to provide the support of backward XPath axes);

The crux of our results relies on the use of a two-way logic to represent inferred tree type portions during the process of backward type inference. This logic provides a succinct notation for regular tree types containing constraints expressed in both upward and downward directions.

2. PROGRAMMING WITH XQUERY

Consider the following XQuery sample program:

```
1. <Products>
2.  { for $cd in doc('catalog.xml')/descendant::CD
3.    return <CompactDisc>
4.             {$cd/ancestor::Product/ProductNo}
5.             {$cd/child::*}
6.           </CompactDisc> }
7. </Products>
```

This program operates on an instance of a catalog of products whose type is defined by the XML schema given in Figure 1. This type corresponds to a set of products containing `Book` and `CD` elements. The program aims at generating an XML data flow, corresponding to a catalog of products containing only `CompactDisc` information. The program output is expected to be valid against the target XML schema given in Figure 2. At first sight, the program seems correct. First it traverses all `CD` in the input document ranged over `$cd` (line 2). For each of them, it extracts its `ProductNo` information using a backward navigation to the `Product` element (line 4). However, a careful analysis shows that this simple example contains two errors: i) it generates an unexpected element (`Sleeve`) and ii) the order of elements do not conform to the targeted schema. The goal of the present paper is to detect this kind of errors automatically.

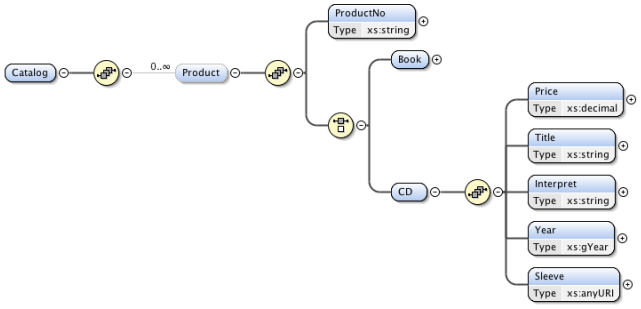

Figure 1: Input Type for "catalog.xml".

2.1 XQuery Programs

The XQuery fragment we consider in this paper is given by the abstract syntax shown in Figure 3, where $axis \in \{\texttt{child}, \texttt{desc}, \texttt{parent}, \texttt{anc}, \texttt{psibl}, \texttt{nsibl}\}$. This fragment is equipped with the main control statements (`for` loops, conditionals, `let` binders) as well as backward, forward and recursive navigation performed by XPath expressions.

We distinguish variables ($\$v$) bound by `for` loops from variables ($\$\overline{v}$) bound by `let` expressions. The former are bound to non-empty trees, whereas the latter are bound to arbitrary (and possibly empty) forests.

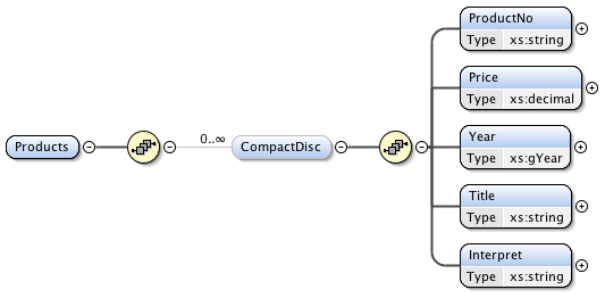

Figure 2: Expected Output Type.

$$
\begin{array}{rcl}
e & ::= & \\
 & & \texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>} \\
 & | & \epsilon \\
 & | & e, e \\
 & | & \texttt{for } \$v \texttt{ in } e \texttt{ return } e \\
 & | & \texttt{let } \$\overline{v} \texttt{ := } e \texttt{ return } e \\
 & | & \texttt{if } cond \texttt{ then } e \texttt{ else } e \\
 & | & \$v\ axis{::}n \\
 & | & \$var \\
 & | & \texttt{root()} \\
cond & ::= & \\
 & & cond \vee cond \mid cond \wedge cond \mid \neg cond \\
 & | & \texttt{empty}(\$var) \mid \$var = \$var \\
\$var & ::= & \\
 & & \$v \\
 & | & \$\overline{v}
\end{array}
$$

Figure 3: Core XQuery Fragment.

2.2 XQuery Types

As a model for XQuery types, we consider the class of regular tree languages that capture most of the DTDs, XML Schemas, and Relax NG schemas found in practice, with the additional benefit that this class is closed under set-theoretic operations. Furthermore, operations such as containment are decidable.

The usual grammar for XQuery types which is typically considered in the literature is given in Figure 4. This grammar actually defines context-free grammars (for which containment is known to be undecidable). Some additional constraints must be applied on this grammar for it to define regular tree grammars (the typical additional restriction is that every occurrence of a recursive variable must either be guarded or must occur in tail). Furthermore, this grammar contains syntactic sugar (i.e. constructions that can be expressed in terms of others, such as $t? = t \mid ()$).

In the present paper, we consider regular tree types from another slightly different (yet equivalent) perspective. We consider the abstract syntax over binary trees, shown in Figure 5. Considering regular tree types through this abstract syntax offers three advantages: first, this grammar already incorporates intrinsically the aforementioned restriction so that it exactly defines regular tree grammars (there is no need for further restriction). Second, this grammar uses fewer constructions which reduces case analyses and in particular the number of inference rules presented in Section 3.3. Third, it makes explicit, through a let binder, recursion which may occur in types. Finally, those advantages come at no cost since unranked trees can be modeled as bi-

nary trees without loss of generality, owing to a bijective mapping between unranked and binary trees [9].

In the rest of the paper, we rely on the widely known "first-child" and "next-sibling" metaphor [9] for navigating unranked trees, and as a mean to represent them in a binary fashion. For example, the type of all forests is represented as follows: $\texttt{let}\ x_{\text{any}}\ = \texttt{element}\ *\ \{x_{\text{any}}, x_{\text{any}}\}\ |\ ()\ \texttt{in}\ x_{\text{any}}$

u	::=		unit type
		$\texttt{element}\ n\ \{t\}$	element
n	::=		name test
		σ	label
	|	$*$	wildcard
t	::=		type
		u	unit type
	|	$()$	empty sequence
	|	t, t	sequence
	|	$t \mid t$	choice
	|	$t?$	optional
	|	$t+$	one or more
	|	$t*$	zero or more
	|	x	type reference

Figure 4: XQuery Types (with syntactic sugar).

n	::=		name test
		σ	label
	|	$*$	wildcard
t	::=		type
		$\texttt{element}\ n\ \{x, x\}$	element with type references
	|	$()$	empty sequence
	|	$t \mid t$	choice
	|	$\texttt{let}\ x\ = t\ \texttt{in}\ t$	recursive type

Figure 5: Types (desugarized and binarized).

3. BACKWARD TYPE INFERENCE

3.1 Main Principles

Formally, our type-checker takes as input:

- a given program e written in the XQuery fragment described in Figure 3;
- a type S_{in} for documents actually given as input to e;
- a type S_{out} for documents expected as output of e.

S_{in} and S_{out} are first compiled into their core representations t_{in} and t_{out} written using the grammar given in Figure 5. The type-checker then applies the inference process to e and t_{out} as described in Section 3.3. This process yields an inferred input type t_{inf}. This type represents the set of all documents admissible as input to e, or in other terms, the set of all documents t such that $e(t)$ is a valid document with respect to t_{out}. We finally check whether:

$$t_{\text{in}} \subseteq t_{\text{inf}} \tag{1}$$

Technically, (1) is checked using the satisfiability-solver introduced in [9]. If (1) holds then no error is reported. If (1) does not hold then we generate a counter-example tree illustrating why the program is found incorrect. The counter-example is a sample input document for which e will output a document which is invalid with respect to t_{out}.

3.2 Logical Representation of Inferred Types

The type which is inferred by our process is represented using logical formulas written with the logic of [9]. One crucial advantage of this representation is that it allows expressing properties over both ancestors and descendants of a given node. This is necessary to support XPath steps other than `child` and `descendant` found in XQuery programs but traditionally ignored in the literature [4]. Other advantages of this representation include the fact that it is expressive enough to support all XQuery types, it is succinct (types are represented as formulas of linear size compared to their regular expression syntax), and the satisfiability problem for a logical formula of size n can be efficiently decided with an optimal $2^{\mathcal{O}(n)}$ worst-case time complexity bound with the solver of [9].

The syntax of the logic in which the inferred type is expressed is defined as follows:

φ, ψ	::=		formula
		$\top$	true
	|	$\bot$	false
	|	$p \quad \mid \quad \neg p$	atomic prop. (negated)
	|	X	variable
	|	$\varphi \vee \psi$	disjunction
	|	$\varphi \wedge \psi$	conjunction
	|	$\langle a \rangle\, \varphi \quad \mid \quad \neg \langle a \rangle\, \top$	existential (negated)
	|	$\mu X.\varphi$	(least) fixpoint

where $a \in \{1, 2, \overline{1}, \overline{2}\}$ are *programs*. Atomic propositions p correspond to labels for tree nodes.

Intuitively, the logic allows one to formulate regular properties on trees: the programs "1" and "2" are respectively used to access the first child node and the next sibling node in an unranked tree. For instance, the formula $a \wedge \langle 1 \rangle\, (b \wedge \langle 2 \rangle\, c)$ is satisfied at the root of the tree denoted by the term $a(b, c)$. The logic also features converse programs $\overline{1}$ and $\overline{2}$ that respectively navigate in the opposite direction compared to 1 and 2.

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

The interpretation of a logical formula is the set of finite trees such that the formula is satisfied at some tree node. From this perspective, a logical formula can be seen as an alternative representation of a finite tree automaton, or, equivalently a type (XML Schema). The semantics of this logic is intuitively explained through examples in [8] and formally defined in [9].

3.3 The Inference Process

The inference process is formalized by inference rules that cover all possible cases of instructions and expected output types. Specifically, we write an inference rule of the form:

$$\frac{\textsc{rule-name} \quad H_1 \qquad H_2 \qquad \ldots \qquad H_i}{t \ \hookleftarrow\ e,\ t_{\text{out}}}$$

This rule means that, for the XQuery instruction e and for the expected output type t_{out}, the input type t is inferred, provided hypotheses $H_1, H_2, \ldots\ H_i$ are satisfied. t represents a constraint that the input of e must satisfy in order for e

to produce an output which satisfies the constraints given by t_{out}. The above rule is named "RULE-NAME".

The inference process is triggered by the application of the rule that applies to the root of the parse tree of the program. The XQuery program is decomposed in terms of subexpressions. The rule that applies at top level recursively calls rules for its sub-expressions. This decomposition of the program expression guides the traversal of the output type. Rule applications produce portions of the inferred type that are assembled in order to compose the final inferred type. Technically, portions are assembled when popping the stack of recursive calls of sub-rules. At the end of the inference process, we obtain a closed logical formula.

3.4 Auxiliary Definitions

We now introduce some auxiliary definitions used in the inference rules.

For two node tests n and n' we define the predicate: $\texttt{match}(n, n') \stackrel{\text{def}}{=} (n = * \vee n' = * \vee n = n')$. For a given nodetest n the function $\texttt{toF}(n)$ computes a corresponding logical formula as follows: $\texttt{toF}(\sigma) \stackrel{\text{def}}{=} \sigma$ and $\texttt{toF}(*) \stackrel{\text{def}}{=} \top$. We also define $\texttt{ntoF}(n, n') \stackrel{\text{def}}{=} \texttt{toF}(n) \wedge \texttt{toF}(n')$.

We need some notations in order to deal with types. First, we introduce x_ϵ as referring to the empty type: $\texttt{let } x_\epsilon \ = \ \texttt{() in } x_\epsilon$. The function $\theta(x)$ denotes the type bound to the variable x. For a given variable x, the function $f(x)$ return the logical translation of $\theta(x)$, as follows:

$$
\begin{aligned}
f(x) &= f'(\theta(x)) \\
f'(\texttt{element } n\ \{x, x\}) &= \texttt{toF}(n) \wedge \texttt{next}(1, x_1) \wedge \texttt{next}(2, x_2) \\
f'(\texttt{()}) &= \bot \\
f'(t_1 \mid t_2) &= f'(t_1) \vee f'(t_2) \\
f'(\texttt{let } x \ = t_1 \texttt{ in } t_2) &= f'_{\text{rec}}(t_2)
\end{aligned}
$$

In the last definition, the function $f'_{\text{rec}}(t_2)$ does the same as $f'(t_2)$ except that it replaces all occurrences of x at top level in t_2 with $\mu X.f'_{\text{rec}}(t_1)$ where X is a fresh variable, and the other (deeper) occurrences of x in t_2 with X.

We define a function particularly useful for describing the frontier of a type:

$$
\begin{aligned}
&\texttt{next}(k, x) = \texttt{next}(k, \theta(x)) \\
&\texttt{next}(k, t) = \begin{cases} \neg \langle k \rangle \top & \text{if } t = \texttt{()} \\ \neg \langle k \rangle \top \vee \langle k \rangle f'(t) & \text{if } t \neq \texttt{()} \wedge \mathit{nullable}(t) \\ \langle k \rangle f'(t) & \text{if } \neg \mathit{nullable}(t) \end{cases}
\end{aligned}
$$

The predicate $\texttt{somewhere}(\varphi)$ is defined as:

$$\mu X.\, (\varphi_{\text{root}} \wedge \mu Y.\varphi \vee \langle 1 \rangle Y \vee \langle 2 \rangle Y) \vee \langle \overline{1} \rangle X \vee \langle \overline{2} \rangle X$$

We introduce notations for performing substitutions in formulas, instructions, and also in types. The notation $\varphi \left[{}^{\psi}/_{\psi'} \right]$ denotes the formula φ in which each occurrence of ψ is replaced by ψ'. $e_1 \left[{}^{\$v}/_{e_2} \right]$ denotes the instruction e_1 in which all occurrences of $\$v$ are replaced by e_2. The notation $t \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right]$ denotes the type expression t in which all (sibling) occurrences of x are replaced by x', as follows:

$$
\begin{aligned}
(\texttt{element } n\ \{x_1, x_2\}) \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right] &= \begin{cases} \texttt{element } n\ \{x_1, x'\} & \text{if } x_2 = x \\ \texttt{element } n\ \{x_1, x_2\} & \text{otherwise} \end{cases} \\
\texttt{()} \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right] &= \texttt{()} \\
(t_1 \mid t_2) \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right] &= t_1 \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right] \mid t_2 \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right] \\
(\texttt{let } x_1 \ = t_1 \texttt{ in } t_2) \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right] &= \texttt{let } x_1 \ = t_1 \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right] \texttt{ in } t_2 \left[{}^{\overrightarrow{x}}/_{\overrightarrow{x'}} \right]
\end{aligned}
$$

We define the widely used predicate $\mathit{nullable}(t)$ that indicates whether the empty tree may belong to the language associated with some type t:

$$
\begin{aligned}
\mathit{nullable}(\texttt{element } n\ \{x, x\}) &= \text{false} \\
\mathit{nullable}(\texttt{()}) &= \text{true} \\
\mathit{nullable}(t_1 \mid t_2) &= \mathit{nullable}(t_1) \vee \mathit{nullable}(t_2) \\
\mathit{nullable}(\texttt{let } x \ = t_1 \texttt{ in } t_2) &= \mathit{nullable}(t_2)
\end{aligned}
$$

The function $\texttt{ctoF}(\mathit{cond})$ returns the logical translation of the condition cond, computed as follows:

$$
\begin{aligned}
\texttt{ctoF}(\mathit{cond}_1 \vee \mathit{cond}_2) &= \texttt{ctoF}(\mathit{cond}_1) \vee \texttt{ctoF}(\mathit{cond}_2) \\
\texttt{ctoF}(\mathit{cond}_1 \wedge \mathit{cond}_2) &= \texttt{ctoF}(\mathit{cond}_1) \wedge \texttt{ctoF}(\mathit{cond}_2) \\
\texttt{ctoF}(\neg \mathit{cond}) &= \neg \texttt{ctoF}(\mathit{cond}) \\
\texttt{ctoF}(\texttt{empty}(\$\mathit{var})) &= \neg(\$\mathit{var}) \\
\texttt{ctoF}(\$v = \$\mathit{var}) &= \top
\end{aligned}
$$

while $\texttt{hasEq}(\mathit{cond})$ simply tests whether the condition cond contains an equality test:

$$
\begin{aligned}
\texttt{hasEq}(\mathit{cond}_1 \vee \mathit{cond}_2) &= \texttt{hasEq}(\mathit{cond}_1) \vee \texttt{hasEq}(\mathit{cond}_2) \\
\texttt{hasEq}(\mathit{cond}_1 \wedge \mathit{cond}_2) &= \texttt{hasEq}(\mathit{cond}_1) \vee \texttt{hasEq}(\mathit{cond}_2) \\
\texttt{hasEq}(\neg \mathit{cond}) &= \texttt{hasEq}(\mathit{cond}) \\
\texttt{hasEq}(\texttt{empty}(\$\mathit{var})) &= \text{false} \\
\texttt{hasEq}(\$\mathit{var} = \$\mathit{var}) &= \text{true}
\end{aligned}
$$

Finally we define the predicate $\mathit{nilable}(e)$ which is true whenever instruction e might not produce anything as output. This predicate is analogous in spirit to $\mathit{nullable}(t)$ but for instructions.

$$
\begin{aligned}
\mathit{nilable}(\texttt{<}\sigma\texttt{>\{}e\texttt{\}</}\sigma\texttt{>}) &= \text{false} \\
\mathit{nilable}(\epsilon) &= \text{true} \\
\mathit{nilable}(e_1, e_2) &= \mathit{nilable}(e_1) \wedge \mathit{nilable}(e_2) \\
\mathit{nilable}(\texttt{for } \$v \texttt{ in } e_1 \texttt{ return } e_2) &= \mathit{nilable}(e_1) \vee \mathit{nilable}(e) \\
\mathit{nilable}(\texttt{let } \$\overline{v} \texttt{ := } e_1 \texttt{ return } e_2) &= \mathit{nilable}(e') \\
\mathit{nilable}(\texttt{if } \mathit{cond} \texttt{ then } e_1 \texttt{ else } e_2) &= \mathit{nilable}(e_1) \vee \mathit{nilable}(e_2) \\
\mathit{nilable}(\$v\ \mathit{axis}\texttt{::}n) &= \text{true} \\
\mathit{nilable}(\$\mathit{var}) &= \$\mathit{var} \\
\mathit{nilable}(\texttt{root()}) &= \text{false}
\end{aligned}
$$

where $e \stackrel{\text{def}}{=} e_2 \left[{}^{\$v}/_{e_1} \right]$ and $e' \stackrel{\text{def}}{=} e_2 \left[{}^{\$\overline{v}}/_{e_1} \right]$.

The predicate $\texttt{isChildOrNsiblStar}(\$\overline{v})$ checks whether the let variable $\$\overline{v}$ is bound to $\$v'$ $\texttt{nsibl::*}$ or $\$v'$ $\texttt{child::*}$ for some for variable $\$v'$.

3.5 Inference Rules

For each case of instruction e, and each case of expected output type t_{out}, we define a rule that infers a logical formula φ_{in} that represents the set of acceptable input to e so that e generates something valid with respect to t_{out}. The purpose of the inferred φ_{in} is to represent all admissible inputs to e yet being the most constrained as possible. The more constrained φ_{in} is, the more precise the type-checking algorithm will be.

One rule (or group of rules) is associated with each possible construct of the output type which can be: choice, recursion, empty sequence or element type (as defined by the grammar shown in Figure 5).

3.5.1 Choice and Recursion

The easiest constructs to deal with are choice types and recursive types, respectively treated by the rules EXPR-UNION and EXPR-REC in Figure 6. EXPR-UNION can be interpreted as follows: for e to generate something valid w.r.t. $(t_1 \mid t_2)$, e can either generate something valid w.r.t t_1 or something valid w.r.t t_2. The constraint inferred for the input simply requires that at least one of those two situations is satisfied, which is naturally formalized as a logical disjunction. The rule EXPR-REC simply propagates the analysis to the returned part of the `let` construct, provided that the type bound to the variable can be accessed using $\theta(x)$.

$$\text{EXPR-UNION}\ \frac{t_l \ \hookleftarrow\ e,\ t_1 \qquad t_r \ \hookleftarrow\ e,\ t_2}{t_l \vee t_r \ \hookleftarrow\ e,\ (t_1 \mid t_2)} \qquad \text{EXPR-REC}\ \frac{t \ \hookleftarrow\ e,\ t_2}{t \ \hookleftarrow\ e,\ \texttt{let}\ x\ = t_1\ \texttt{in}\ t_2}$$

Figure 6: Rules for choice and recursive types.

3.5.2 Empty Sequence

The case of the empty sequence as output type is treated by rules shown in Figure 7. One rule is given for each instruction as the inferred constraint depends on the nature of the instruction. For example, ELEM-EMPTY infers the formula $\bot$ (or, in other terms, the empty type) because the generation of an element cannot produce an empty sequence. On the opposite, EMPTY-EMPTY infers the formula $\top$ (the type of any tree, or in other terms, no particular type constraint), since the empty instruction never generates anything, which matches the empty sequence expected as output. The rule CHILD-EMPTY handles the case where the construct "$\$v$ `child::`n" is supposed to generate an empty sequence: it infers the constraint that $\$v$ must not have any child matching n. Rules for other steps follow the same spirit. The rule FOR-EMPTY infers a disjunction which corresponds to the following situations: (i) e_1 does not produce any output (in which case e_2 does not matter since it is not executed), or (ii) e_1 produces a non-empty output (which implies that e_2 is executed at least once) but e_2 does not produce any output. This situation corresponds to the second part of the disjunction, in which each occurrence of $\$v$ in e_2 is replaced by t which denotes the constraint over the input such that the evaluation of e_1 is non-empty. This constraint is computed by a call $t \xleftarrow{\text{in}} e_1$ which uses the set of inference rules shown on Figure 8. This set of rules is specifically designed for the analysis of `in` clauses found in `for` loops and `let` instructions. These particular rules only take an instruction as parameter (no output type). Consider for instance the following statement:

`for` $\$v$ `in` $\$r$ `child::`a `return` $\$v$ `child::`b

in the presence of the empty sequence `()` expected as output type. For this example, the application of FOR-EMPTY triggers CHILD-EMPTY for $e_1 = \$r$ `child::`a resulting in a formula t_1 expressing the fact that $\$r$ has no children. The rule IN-CHILD is also triggered for e_1 resulting in a formula t that characterizes $\$v$ as a child of $\$r$. In addition, the rule CHILD-EMPTY is triggered for $e_2 = \$v$ `child::`b resulting in a formula t_2 that forbids the existence of "`b`" labeled nodes as children of $\$v$. Finally, the global inferred formula $t_1 \vee t_2 \left[{}^{\$v}/_{t}\right]$ is composed from these known subformulas.

The acute reader may notice here the importance of using a two-way logic: this makes possible the inference of a formula that expresses constraints on both the upward context (ancestor nodes) and the downward context (descendant nodes) of some variable $\$v$ ranging over input tree nodes. In fact, t, as computed above, characterizes the upper context of $\$v$ in the input (or, in other terms, how a node bound to $\$v$ has been reached in the input and what this implies in terms of structural requirements) whereas t_2 characterizes the downward context of $\$v$ in the input (or, in other terms, how the children of $\$v$ in the input, if any, look like based on what we know from the output type). This capability of inferring two-way constraints for the input constitutes a fundamental trait of our approach.

3.5.3 Element Type

For the general case where an element type is expected as output, each instruction is dealt with by a few specifically designed inference rules. The inferred constraint varies a lot depending on the nature of the instruction and on the situation determined by the rule premises. Figure 9 presents rules for all the main control statements, including the `for` loop, the `if` statement, and instructions in charge of output production. We comment on these rules below.

Generation of nodes.

One of the simplest yet essential rule is ELEM-ELEM. This rule checks whether the node generated by the instruction is compatible with the expected node label of the output type. In addition, since the XQuery instruction produces a tree (and not a forest) the output type for siblings of the produced node must be nullable. Otherwise, ELEM-ELEM' infers the formula $\bot$ in order to indicate the problem. EMPTY-ELEM always infers $\bot$ since the empty instruction (modeling no instruction at all) is simply not capable of generating the expected output type that requires the existence of at least one element.

Conditional statements.

The case of the "`if`" statement is especially interesting as it clearly illustrates to which extent the precision of the type-checker can be increased depending on the expressive power of the underlying logic. The rule IF-ELEM applies whenever the condition does not contain any equality test. In this case, the logic is expressive enough to fully model the conditional statement: the rule infers two possible cases: either the input logically follows the restrictions expressed by the condition in which case the additional restrictions computed by the analysis of the "`then`" branch apply; or the input does not logically follow the restrictions computed from the analysis of the condition in which case the further restrictions computed by the analysis of the "`else`" branch apply. This complete logical modeling of the "`if`" statement cannot be achieved if the condition contains an equality test, because we do not know the result of the test statically (as it depends on values obtained at runtime). Rule IF-ELEM' is in charge of this case: an equality test is modeled with the logical formula $\top$ which abstracts over this part of the condition by not requiring any restriction on the input. Because of this unavoidable approximation, the global inferred formula is a disjunction of one of the two possibilities obtained by the further analyses of the "`then`" and "`else`" branches.

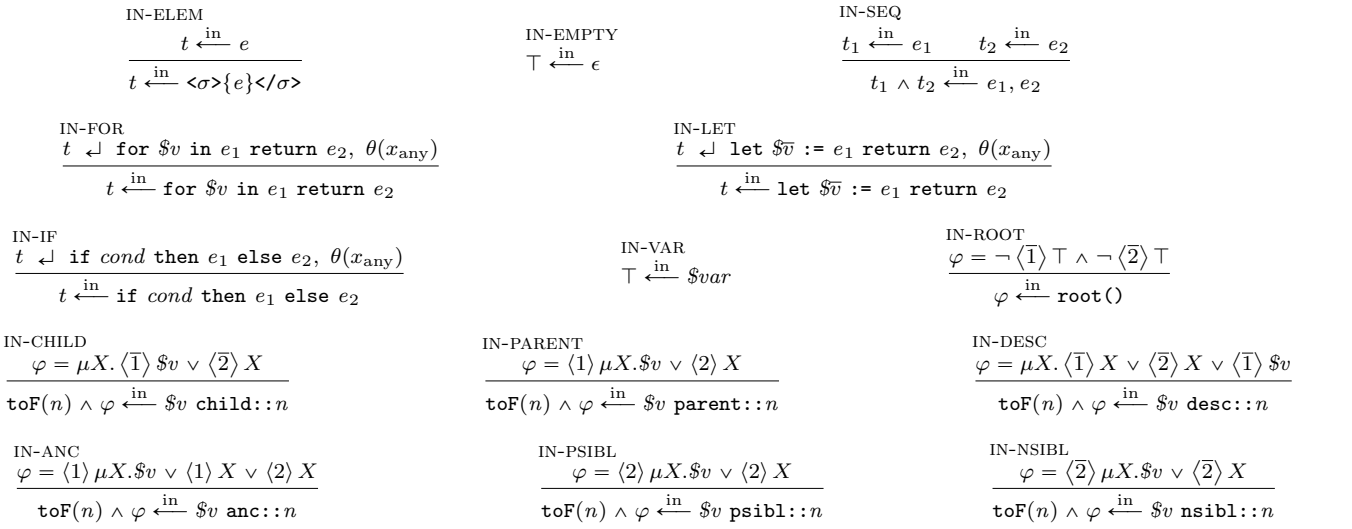

Figure 8: Rules for instructions inside `in` clauses of `for` loops.

Sequence of instructions.

The case of a sequence of instructions e_1, e_2 is especially hard to deal with as we have to determine which part of the output type is generated by e_1 and which part is generated by e_2. This means that we have to perform a joint analysis of $(e_1,\ e_2)$ and the output type in order to determine at which location the output type can be split in two parts respectively generated by e_1 and e_2. For this purpose, we introduce an auxiliary function $\texttt{succ}\,(e, t)$ defined in Figure 10 . For a given instruction e and output type t, the function $\texttt{succ}\,(e, t)$ computes all the possible remaining parts of the output type after removing all the possible parts that can be generated by e. This is represented using a set of type variables. Intuitively, if we consider the case of a sequence e_1, e_2, $\texttt{succ}\,(e_1, t)$ returns the set of variables to be considered as possible entrypoints for e_2. In other terms, $\texttt{succ}\,(e_1, t)$ represents the set of all possible places where t can be split in two parts: the first being generated by e_1 and the second by e_2.

The definition of $\texttt{succ}\,(e, t)$ is presented in Figure 10 where t is used as a shorthand for $\texttt{element}\ n\ \{x_1, x_2\}$ and where $\mathit{axis} \in \{\texttt{child}, \texttt{desc}, \texttt{parent}, \texttt{anc}, \texttt{psibl}, \texttt{nsibl}\}$.

By definition, if $x_\epsilon \in \texttt{succ}\,(e, t)$ then e can produce an output that matches the entire output type t. If $x_\epsilon \notin \texttt{succ}\,(e, t)$ then the output type is expecting something else (a mandatory remaining part) that cannot be generated by e. Finally, $\texttt{succ}\,(e, t) = \varnothing$ means that e generates a pattern which is not acceptable for t.

Using this auxiliary function, we design the backward type inference rules SEQ-ELEM, SEQ-ELEM' and SEQ-ELEM" for the sequence of instructions. SEQ-ELEM eliminates the cases where e_1 generates a pattern which is not compatible with the output type. SEQ-ELEM' eliminates the cases where e_1 does generate a pattern compatible with the output type, but e_2 does not. Finally, SEQ-ELEM" computes the set S of all admissible entrypoints for e_2. The set F filters out entrypoints for which e_2 cannot fully generate the expected remaining part of the type. F contains the admissible entrypoints for e_2 so that e_1, e_2 successfully generates an output which fully satisfies the entire expected output type. The inferred formula φ simply describes all acceptable splits of the output type so that e_1, e_2 generates an output valid with respect to the output type.

Iteration and variable-binding.

To better understand the case of the `for` loop and backward type inference performed by FOR-ELEM, recall that the semantics of the
"`for` $\$v$ `in` e_1 `return` e_2" statement consists in first evaluating instruction e_1, and then in iterating over each member of the sequence obtained by the result of the evaluation of e_1. Iteration is done by successively setting variable $\$v$ to each member of this sequence and executing e_2 each time accordingly. Obviously, e_2 may contain one (or possibly several) occurrences of $\$v$. For this reason, backward inference for the `for` loop is intrinsically tied to backward inference for variables in navigation steps, and the behavior of FOR-ELEM must be understood together with the behavior of rules that deal with step-related variables, and in particular rules of Figure 11. Whenever a variable occurrence is encountered inside e_2, a rule (e.g. CHILD-ELEM) associates the expected output type to this variable by inferring a logical formula that represents the expected output type and that contains the name $\$v$ of the variable which is meant as a placeholder. This placeholder will be replaced at a later stage whenever the inference process will come back to the `for` loop that binds this variable (recall that the inference process is a depth-first tree traversal of the XQuery instruction parse tree). Rule FOR-ELEM" replaces all occurrences of $\$v$ in t_2 with the result t_1 of the analysis of the instruction e_1 bound to $\$v$ by the `for` loop. In this manner, constraints on $\$v$ coming from the output type, propagated by the rules in charge of steps, will properly apply later when the binding of $\$v$ is met.

Three rules handle the case of the `for` loop: FOR-ELEM, FOR-ELEM', FOR-ELEM". The rules FOR-ELEM and FOR-ELEM' handle peculiar cases where the inference simply consists in copying a portion of the output type into the inferred input

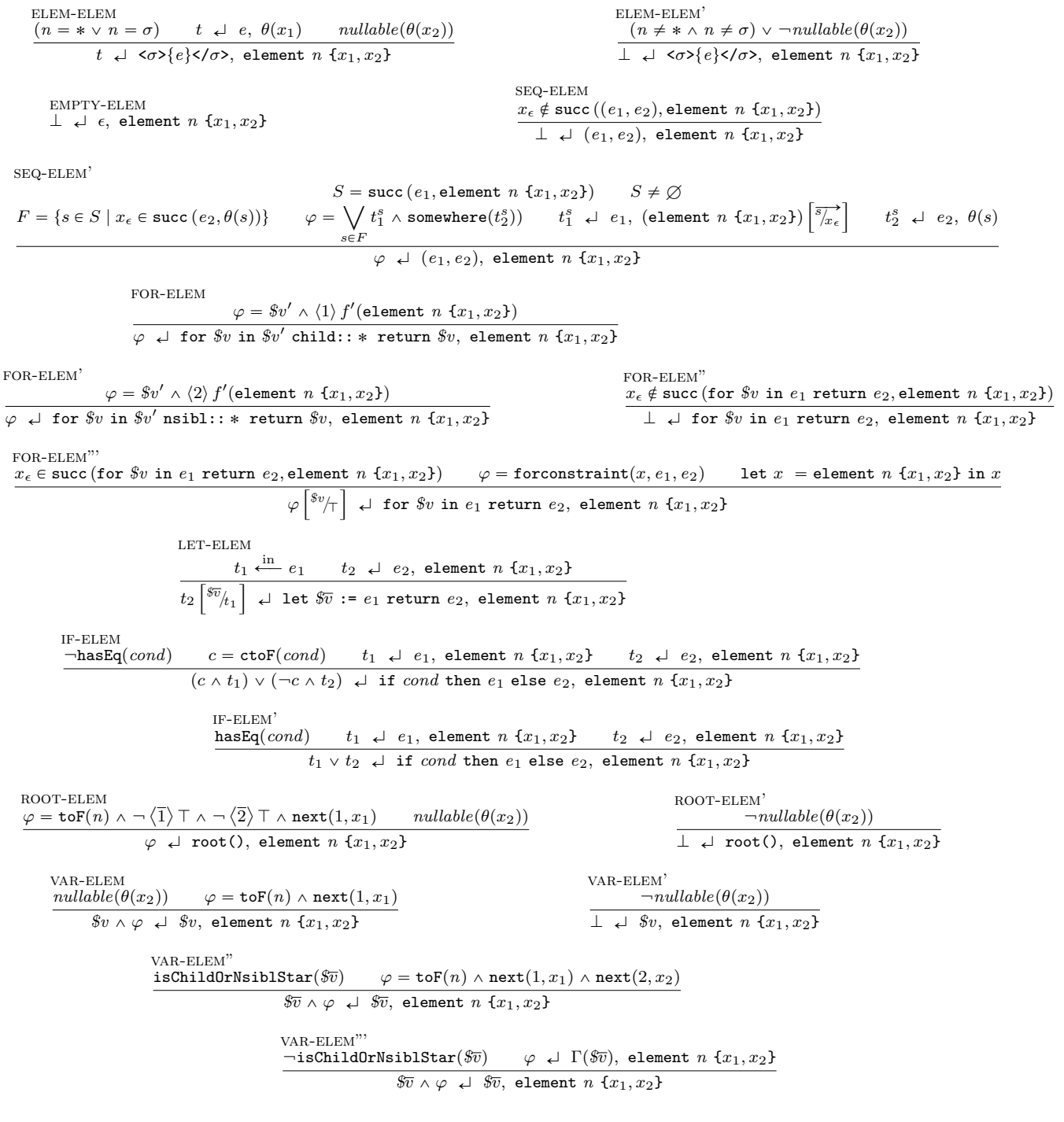

Figure 9: Rules for the main control instructions in the presence of the element type as output, when the expected output type is supposed to be consumed entirely by the considered instruction.

$$
\begin{aligned}
\texttt{succ}\,(e, t_1 \mid t_2) &= \texttt{succ}\,(e, t_1) \cup \texttt{succ}\,(e, t_2)\\
\texttt{succ}\,(e, \texttt{let}\ x\ := t_1\ \texttt{in}\ t_2) &= \texttt{succ}\,(e, t_2)\\
\\
\texttt{succ}\,(\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>}, ()) &= \varnothing\\
\texttt{succ}\,(\epsilon, ()) &= \{x_\epsilon\}\\
\texttt{succ}\,(e_1, e_2, ()) &= \begin{cases} \{x_\epsilon\} & \text{if } \mathit{nilable}(e_1) \wedge \mathit{nilable}(e_2)\\ \varnothing & \text{otherwise}\end{cases}\\
\texttt{succ}\,(\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2, ()) &= \texttt{succ}\,(e_2, ())\\
\texttt{succ}\,(\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2, ()) &= \texttt{succ}\,(e_2, ())\\
\texttt{succ}\,(\texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2, ()) &= \texttt{succ}\,(e_1, ()) \cup \texttt{succ}\,(e_2, ())\\
\texttt{succ}\,(\$v\ \mathit{axis}\texttt{::}n, ()) &= \{x_\epsilon\}\\
\texttt{succ}\,(\$v, ()) &= \varnothing\\
\texttt{succ}\,(\$\overline{v}, ()) &= \texttt{succ}\,(\Gamma(\$\overline{v}), ())\\
\texttt{succ}\,(\texttt{root()}, ()) &= \varnothing
\end{aligned}
$$

$$
\begin{aligned}
\texttt{succ}\,(\texttt{<}\sigma\texttt{>}\{e\}\texttt{</}\sigma\texttt{>}, t) &= \begin{cases} \{x_2\} & \text{if } \texttt{match}(\sigma, n)\\ \varnothing & \text{otherwise}\end{cases}\\
\texttt{succ}\,(\epsilon, t) &= \varnothing\\
\texttt{succ}\,((e_1, e_2), t) &= \begin{cases} \bigcup_{x \in \texttt{succ}(e_1, t)} \texttt{succ}\,(e_2, \theta(x)) & \text{if } \neg \mathit{nilable}(e_1)\\ \bigcup_{x \in \texttt{succ}(e_1, t)} \texttt{succ}\,(e_2, \theta(x)) \cup \texttt{succ}\,(e_2, t) & \text{otherwise}\end{cases}\\
\texttt{succ}\,(\texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2, t) &= \texttt{succ}\,(e_2, t)\\
\texttt{succ}\,(\texttt{if}\ \mathit{cond}\ \texttt{then}\ e_1\ \texttt{else}\ e_2, t) &= \texttt{succ}\,(e_1, t_1) \cup \texttt{succ}\,(e_2, t_2)\\
\texttt{succ}\,(\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2, t) &= \bigcup_{i=1}^{k} \texttt{succ}\,(e^i, t) \quad \text{such that } \bigcup_{i=1}^{k+1} \texttt{succ}\,(e^i, t) = \bigcup_{i=1}^{k} \texttt{succ}\,(e^i, t) \quad \text{where } e \stackrel{\text{def}}{=} e_2\\
\texttt{succ}\,(\texttt{root()}, t) &= \{x_2\}\\
\texttt{succ}\,(\$v\ \mathit{axis}\texttt{::}n', t) &= \begin{cases} \varnothing & \text{if } \neg\texttt{match}(n', n)\\ \{x_2\} \cup \texttt{succ}\,(\$v\ \mathit{axis}\texttt{::}n', \theta(x_2)) & \text{if } \texttt{match}(n', n)\end{cases}\\
\texttt{succ}\,(\$v, t) &= \texttt{succ}\,(\texttt{<}*\texttt{>}\{\epsilon\}\texttt{</}*\texttt{>}, t)\\
\texttt{succ}\,(\$\overline{v}, t) &= \texttt{succ}\,(\Gamma(\$\overline{v}), t)
\end{aligned}
$$

Figure 10: Computation of the set of variables reached in type t after output production by e.

type. The rule FOR-ELEM" is in charge of the more general remaining cases. Here, the principle is very similar to the one followed for the sequence (rule SEQ-ELEM"). This is because a statement `for` $\$v$ `in` e_1 `return` e_2 can be theoretically seen as an unbounded sequence of the form $e_2, e_2, ..., e_2$. This is exactly what is done by the rule FOR-ELEM".

$$\texttt{forconstraint}(x, e_1, e_2) \stackrel{\text{def}}{=} \texttt{sewfirst}(e_1, \texttt{iter}(x, e_1, e_2, \varnothing))$$

We define an auxiliary function $\texttt{iter}(x, e_1, e_2, E)$ that infers an input formula such that the evaluation of e_2 (possibly repeated any number of times) produces what is expected at x in the output type. For some type variable x, some instructions e_1 and e_2, and some set E of type variables, $\texttt{iter}(x, e_1, e_2, E)$ is defined as follows:

$$\texttt{iter}(x, e_1, e_2, E) \stackrel{\text{def}}{=} \mu X^x.\varphi(S, e_1) \vee \bigvee_{s \in S, s \neq x_\epsilon} \texttt{sew}(e_1, t_s, \psi)$$

where $S = \texttt{succ}\,(e_2, \theta(x))$,
$t_s \ \hookleftarrow\ e_2,\ \theta(x)\left[\overrightarrow{s/x_\epsilon}\right]$,
$\varphi(S, e_1) = \begin{cases} \texttt{sewlast}(e_1, t_{x_\epsilon}) & \text{if } \exists s \in S \mid \mathit{nullable}(\theta(s))\\ \bot & \text{otherwise}\end{cases}$,
X^x is a fresh recursion variable associated to x,
and $\psi = \begin{cases} \texttt{iter}(s, e_1, e_2, E \cup \{x\}) & \text{if } s \notin E\\ \psi = X^s & \text{if } s \in E\end{cases}$

E represents the set of already visited type variables. Intuitively, $\texttt{iter}(x, e_1, e_2, E)$ builds the tree of all possible ways to "consume" entirely the output type by productions made by successive evaluations of e_2 that finally arrive to the empty sequence x_ϵ in the output type. Branches that cannot reach x_ϵ are ignored ($\bot$ is returned). The function $\texttt{sew}(\$v\ \texttt{child::}*, \varphi_1, \varphi_2)$ is in charge of properly "sewing" the portions of the inferred input type depending on the way the input tree nodes were accessed (i.e. depending on e_1 in the `for` loop). For example:

$$\texttt{sewfirst}(\$v\ \texttt{child::}*, \varphi) \stackrel{\text{def}}{=} \$v \wedge \langle 1 \rangle\, \varphi$$

$$\texttt{sew}(\$v\ \texttt{child::}*, \varphi_1, \varphi_2) \stackrel{\text{def}}{=} \varphi_1 \wedge \langle 2 \rangle\, \varphi_2$$

$$\texttt{sewlast}(\$v\ \texttt{child::}*, \varphi) \stackrel{\text{def}}{=} \varphi \wedge \neg \langle 2 \rangle\, \top$$

The general idea is that a successful path between an entry point for e_2 and x_ϵ must exist. Basically the expected output type must be "consumed" entirely after a certain number of application of e_2.

Finally, notice that the rule VAR-ELEM' may raise an error whenever some next siblings are required in the output type, because they cannot be generated by $\$v$ which is bound to a single tree, by definition of `for` variables. The rules VAR-ELEM" and VAR-ELEM"' are in charge of `let` variables that can be bound to forests. Inferred free variables will simply be substituted by rule LET-ELEM.

Navigation in the input tree.

Figure 11 presents rules that deal with all the instructions that perform navigation in the input tree. Most of the rules use the hypothesis $H_{\text{sibl}}(x_2, n)$ that checks the compatibility of the expected output type with the nodes returned by the step:

$$H_{\text{sibl}}(x_2, n) \stackrel{\text{def}}{=} \texttt{sat}(\texttt{next}(2, x_2) \wedge \varphi_{\text{restrictSiblings}}(\texttt{toF}(n)))$$

where the predicate $\texttt{sat}(\psi)$ checks whether a given formula ψ is satisfiable, using a direct call to the solver of [9]. In the above definition, $\varphi_{\text{restrictSiblings}}(\psi)$ ensures that all next siblings satisfy ψ, and is defined as follows:

$$\varphi_{\text{restrictSiblings}}(\psi) \stackrel{\text{def}}{=} \neg \langle 2 \rangle \top \vee \langle 2 \rangle \mu X.\psi \wedge (\neg \langle 2 \rangle \top \vee \langle 2 \rangle X)$$

Notice that checking the hypothesis $H_{\text{sibl}}(x_2, n)$ is facultative but increases precision.

3.6 Termination of the Algorithm

PROPOSITION 3.1. *The inference process terminates for any instruction* e *and output type* t.

Proof (sketch): The inference process terminates because each inference rule recursively calls other rules on instructions whose sizes are strictly smaller compared to the size of the initial instruction. Therefore, the bounded size of the XQuery program parse tree guarantees termination of the inference process. The only exception are rules EXPR-UNION and EXPR-REC, which recursively call other rules on same-size instructions, but with output types whose sizes are strictly decreased. This guarantees termination even in the presence of recursion in the output type (due to the guarded recursion restriction). Specifically, in the presence of recursion in the output type, a rule XXX-ELEM will eventually apply in order to reach the recursive occurrence of a type variable. By examining all the XXX-ELEM rules (shown in Figures 9 and 11) we can see that each rule either performs a recursive call on an instruction whose size is decreased, or terminates directly (without any recursive call). □

The next section notably details step by step how the implementation processes an example involving recursion in the output type.

4. PRACTICAL EXPERIMENTS

4.1 Implementation Architecture

The whole system is implemented in Java and is obtained by the implementation and then the assembly of several components. First, we rely on a parser for XQuery expressions (we use JJTree to generate a basic parser from the XQuery normative grammar specification), and, more importantly an API for representing and manipulating Core XQuery expression parse trees. A set of parsers for DTDs, XML Schemas and Relax NG schemas read schema files and produce a unifying unranked regular tree grammar representation from a description in any of those languages. We use the already available "MSV" parser, and we convert its internal representation in terms of our CFT. At this stage, a converter translates the unranked regular tree grammar into the corresponding binary tree grammar. Using an API for manipulating binary tree grammars, we have implemented a compiler that translates the binary tree grammar in terms of a logical formula. We also use an API to generate and manipulate logical formulas. Using these two APIs we have implemented all the auxiliary predicates needed for the inference rules. Inference rules are implemented using the visitor pattern. The input XQuery expression is traversed, and corresponding rules are called on each subexpression, depending on the nature of the construct of the output type. Rules are directly implemented (as a separate Java class) as described in the present paper: each rule generates a portion of a logical formula, that possibly contains a variable that will be substituted by another piece of formula at a later step when the variable binder is met (when the rules for-xxx or let-xxx apply). The global logical formula is thus obtained by the assembly of its sub-formulas when the stack of recursive calls is emptied. Finally, we rely on the satisfiability solver of [9] for checking the unsatisfiability of the negation of the implication between formulas representing the input type and the inferred input type[2].

The implementation of our tool is available from the following URL: http://tyrex.inria.fr/xqtc

4.2 Experimental Results and Discussion

We detail how a simple example is processed by the implementation in order to illustrate how the technical machinery works. Consider the program e defined below:

$$e \stackrel{\text{def}}{=} \texttt{let } \$v \texttt{ := root() return <r>\{}\$v \texttt{ child::*\}</r>}$$

This program generates a document with a `r`-labeled root whose children are the children of the root of the input tree. Consider also an output tree type t_{out} defined as follows:

$$t_{\text{out}} \stackrel{\text{def}}{=} \texttt{let } x_\epsilon \texttt{ = () in let } x \texttt{ = element b \{}x_\epsilon, x\texttt{\} | () in element r \{}x, x_\epsilon\texttt{\}}$$

This type represents the set of documents with an `r`-labeled root which can have an arbitrary number of `b` children leaves (or, intuitively, `r[b*]` for unranked trees). Concretely, the grammar "sample.dtd" describing this type is parsed and converted to a Context Free Type (CFT) matching the syntax given in Figure 4, as shown by the following trace:

```
Resulting CFT is:
$b ->b(())
$1 ->($b | ($b, $1))
$r ->r(($1 | ()))
$a ->a($b)
Start symbol is $r
```

This grammar is then automatically converted into the binary tree type representation (matching the syntax shown in Figure 5):

[2]Checking containment $A \subseteq B$ corresponds to checking the logical implication $A \Rightarrow B$. Since we use a satisfiability checker (and not a validity checker), this property is checked by testing the unsatisfiability of $A \Rightarrow B = \neg A \vee B$, that is, the satisfiability of $\neg(A \Rightarrow B) = A \wedge \neg B$.

$$\text{CHILD-ELEM}\quad \frac{\varphi = \$v \wedge \langle 1\rangle \left(\texttt{toF}(n') \wedge \texttt{next}(1,x_1) \wedge \texttt{next}(2,x_2)\right)}{\varphi \;\hookleftarrow\; \$v\ \texttt{child::*},\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{CHILD-ELEM'}\quad \frac{n \neq * \qquad \texttt{match}(n,n') \qquad H_{\text{sibl}}(x_2,n) \qquad m = \texttt{ntoF}(n,n') \qquad \varphi = m \wedge \texttt{next}(1,x_1)}{\$v \wedge \langle 1\rangle\, \mu X.\varphi \vee (\neg\texttt{toF}(n) \wedge \langle 2\rangle\, X) \;\hookleftarrow\; \$v\ \texttt{child::}n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{WRONG-STEP-ELEM}\quad \frac{n \neq * \qquad \neg\texttt{match}(n,n') \vee \neg H_{\text{sibl}}(x_2,n)}{\bot \;\hookleftarrow\; \$v\ \mathit{axis}::n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{PARENT-ELEM}\quad \frac{\texttt{match}(n,n') \qquad \mathit{nullable}(\theta(x_2)) \qquad \varphi = \texttt{ntoF}(n,n') \wedge \texttt{next}(1,x_1)}{\$v \wedge \mu X.\, \langle\overline{1}\rangle\, \varphi \vee \langle\overline{2}\rangle\, X \;\hookleftarrow\; \$v\ \texttt{parent::}n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{PARENT-ELEM'}\quad \frac{\neg\texttt{match}(n,n') \vee \neg\mathit{nullable}(\theta(x_2))}{\bot \;\hookleftarrow\; \$v\ \texttt{parent::}n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{DESC-ELEM}\quad \frac{\varphi = \$v \wedge \langle 1\rangle\, (\texttt{toF}(n') \wedge \texttt{next}(1,x_1))}{\varphi \;\hookleftarrow\; \$v\ \texttt{desc::*},\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{DESC-ELEM'}\quad \frac{n \neq * \qquad \texttt{match}(n,n') \qquad H_{\text{sibl}}(x_2,n) \qquad m = \texttt{ntoF}(n,n') \qquad \varphi = m \wedge \texttt{next}(1,x_1)}{\$v \wedge \langle 1\rangle\, \mu X.\varphi \vee \neg\texttt{toF}(n) \wedge (\langle 1\rangle\, X \vee \langle 2\rangle\, X) \;\hookleftarrow\; \$v\ \texttt{desc::}n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{ANC-ELEM}\quad \frac{\varphi = \texttt{toF}(n') \wedge \texttt{next}(1,x_1)}{\$v \wedge \mu X.\, \langle\overline{1}\rangle\, \varphi \vee \langle\overline{2}\rangle\, X \;\hookleftarrow\; \$v\ \texttt{anc::*},\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{ANC-ELEM'}\quad \frac{n \neq * \qquad \texttt{match}(n,n') \qquad H_{\text{sibl}}(x_2,n) \qquad \varphi = \texttt{ntoF}(n,n') \wedge \texttt{next}(1,x_1)}{\$v \wedge \mu X.\, \langle\overline{1}\rangle\, (\varphi \vee \neg\texttt{toF}(n) \wedge X) \vee \langle\overline{2}\rangle\, X \;\hookleftarrow\; \$v\ \texttt{anc::}n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{PSIBL-ELEM}\quad \frac{\varphi = \texttt{toF}(n') \wedge \texttt{next}(1,x_1)}{\$v \wedge \langle\overline{2}\rangle\, \varphi \;\hookleftarrow\; \$v\ \texttt{psibl::*},\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{PSIBL-ELEM'}\quad \frac{n \neq * \qquad \texttt{match}(n,n') \qquad H_{\text{sibl}}(x_2,n) \qquad \varphi = \texttt{ntoF}(n,n') \wedge \texttt{next}(1,x_1)}{\$v \wedge \langle\overline{2}\rangle\, \mu X.\varphi \vee \neg\texttt{toF}(n) \wedge \langle\overline{2}\rangle\, X \;\hookleftarrow\; \$v\ \texttt{psibl::}n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{NSIBL-ELEM}\quad \frac{\varphi = \texttt{toF}(n') \wedge \texttt{next}(1,x_1)}{\varphi \wedge \langle 2\rangle\, \varphi \;\hookleftarrow\; \$v\ \texttt{nsibl::*},\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

$$\text{NSIBL-ELEM'}\quad \frac{n \neq * \qquad \texttt{match}(n,n') \qquad H_{\text{sibl}}(x_2,n) \qquad \varphi = \texttt{ntoF}(n,n') \wedge \texttt{next}(1,x_1)}{\$v \wedge \langle 2\rangle\, \mu X.\varphi \vee \neg\texttt{toF}(n) \wedge \langle 2\rangle\, X \;\hookleftarrow\; \$v\ \texttt{nsibl::}n,\ \texttt{element}\ n'\ \{x_1,x_2\}}$$

Figure 11: Rules for steps with the element type as output, and supposed to consume the entire output type.

```
Resulting BTT is:
$Epsilon ->EPSILON
$2 ->b($Epsilon, $Epsilon) | b($Epsilon, $2)
$1 ->EPSILON | b($Epsilon, $Epsilon) | b($Epsilon, $2)
$r ->r($1, $Epsilon)
Start Symbol is $r
```

The XQuery program e is parsed, and the analysis is triggered at the root of the above output type:

```
Analyzing 'sample.xq' vs 'sample.dtd' starting at 'r'...
Considered Input XQuery expression:
let $v := root() return element r {child::* }
```

The system prints the tree of all rules that apply for the present example, as shown in the trace presented in Figure 12. The trace notably shows parameters for each intermediate rule called. The name of the applied rule is printed together with its input parameters (instruction and type) whenever the rule is triggered. Whenever the rule returns its result (potentially after calling other rules), the resulting formula corresponding to the inferred input type portion is printed. Indentation in the trace corresponds to the depth in the tree of rule calls.

Specifically, the first rule triggered is LET-ELEM of Figure 11, which is instantiated as follows:

$$\text{LET-ELEM}\quad \frac{t_1 \;\hookleftarrow\; \texttt{root()},\ \theta(x_{\text{any}}) \qquad t_2 \;\hookleftarrow\; \texttt{<r>\{}\$v\ \texttt{child::*\}</r>},\ t_{\text{out}}}{t_2\left[{}^{\$\overline{v}}/_{t_1}\right] \;\hookleftarrow\; \texttt{let}\ \$\overline{v}\ \texttt{:= root() return <r>\{}\$\overline{v}\ \texttt{child::*\}</r>},\ t_{\text{out}}} \tag{2}$$

The application of this rule recursively calls one rule for building t_1 and one rule for building t_2. It therefore generates two branches.

For the first branch, since $\theta(x_{\text{any}}) = (\texttt{element} * \{x_{\text{any}}, x_{\text{any}}\})$ | `()` is a disjunction at top level, the formula t_1 is obtained by the application of EXPR-UNION, which in turn calls ROOT-ELEM and ROOT-EMPTY. The application of ROOT-EMPTY directly yields the formula $\bot$ and therefore only the result ROOT-ELEM is relevant for the disjunction. This result is obtained as follows:

This rule performs the auxiliary computations: $\texttt{toF}(*) = \top$, $\texttt{next}(1,x_{\text{any}}) = \neg\langle 1\rangle\top \vee \langle 1\rangle f(x_{\text{any}})$ which is equivalent to $\top$ because $f(x_{\text{any}}) = \mu X.(\neg\langle 1\rangle\top \vee \langle 1\rangle X) \wedge (\neg\langle 2\rangle\top \vee \langle 2\rangle X)$ is equivalent to $\top$, and $\mathit{nullable}(\theta(x_{\text{any}})) = \text{true}$. It

then returns the inferred type $t_1 = \neg\langle\overline{1}\rangle\top \wedge \neg\langle\overline{2}\rangle\top$ that simply requires the existence of a root in the input tree.

For the second branch, the rule EXPR-REC is applied twice to go through the let statements in t_{out} until the rule ELEM-ELEM applies for `element r {`x, x_ϵ`}`. This rule application recursively triggers EXPR-UNION for splitting the choice $\theta(x)$. Again, two branches are created from the split: one that applies CHILD-EMPTY, which terminates immediately with $\$v \wedge \neg\langle 1\rangle\top$, and another one which triggers EXPR-UNION and so forth, as illustrated in 12.

If we pop the stack of recursive calls, at (2) we end up with t_1 as computed above and a value for t_2 returned by rule ELEM-ELEM, as shown in 12. The final inferred type is then obtained by the substitution $t_2\left[{}^{\$v}/_{t_1}\right]$ which denotes t_2 where every occurrence of $\$v$ (`X1` in the implementation trace) is replaced with t_1. This yields the final formula t_{inf} shown in Figure 13:

This inferred formula is surrounded by a formula that ensures its satisfaction at the root of the input tree, before being used in the formula to check containment.

The first part of the formula in Figure 13 states that a child of the root is labeled "b" and does not (line 1), or the root has a first child labeled "b" (does not have any children) and an immediate sibling which is labeled "b" and do not have any children, as well as all its potential successors (recursion on X2, line 2), or the root has only one first child child labeled "b" without children and no sibling (line 3).

One can easily check that this is exactly the type that represents the set of all admissible input documents for e to generate a valid t_{out} document.

Notice that the final inferred type contains duplicate formulas (it refers to the root three times). This is because rules do not communicate (different branches do not share their results) in the current state of the implementation. However, this duplication of subformulas is not a real issue, because duplicate subformulas are automatically factorized by the satisfiability solver.

Once t_{inf} is computed, type-checking is performed by checking whether $t_{\text{inf}} \subseteq t_{\text{in}}$. This is formulated as follows:

```
phi =>   ~<-1>T & ~<-2>T & mu X. tinf | <1>X | <2>X
```

where `tinf` is the formula shown in Figure 13 and `phi` is the logical formula corresponding to t_{in}.

For instance, if we instantiate t_{in} as a type allowing one children labeled "a" under the root, then the system produces a counter-example of the form `*[a]`. Obviously, in realistic scenarios, more rules are triggered, the rule application tree is bigger, and generated formulas are much harder to read. Counter-examples, which are also more complex, become of greater value for the XQuery programmer. In hard cases, this counter-example can be given to a XQuery runtime debugger, for a step-by-step analysis to find the precise locations of the erroneous statement(s) in the program. Notice, that the soundness of our type-checker implies that the XQuery program generates an output which is necessarily not valid (with respect to the expected output type) when fed with the counter-example as input. Contrary to other approaches investigated in the literature that produce false negatives, here the effort spent in debugging is always relevant. Notice also that our type-checker is capable of detecting a wide range of errors, from basic ones to much more involved ones. The class of basic errors detected ranges from very basic ones such as inexistent elements, errors in element names, to wrong ordering in expected content models and generation of an element in inappropriate locations. The type-checker also detects much more subtle errors depending on the expressive power used to describe expected output type constraints. In particular, features such as context-sensitive content models allowed in XML schemas or in Relax NGs are fully captured by our BTT and logical modelings. Therefore the type-checker is equipped to detect errors caused by the generation of an output that matches a possible content model but not in the proper context.

The secret for capturing extremely precise context information is that we are capable of tracking precisely all the context manipulation performed by forward, backward and recursive path expressions. We believe that this removes a major obstacle that prevented the development of precise type-checkers and their widespread use. This is precisely the limitation of existing type-checkers, such as the pioneering Galax [6] which is still among the few available implementations today. Galax is typically unable to detect the error induced by the use of the backward axis in the example given in Section 2, because it directly implements the type-system proposed in the XQuery recommendation [5].

5. RELATED WORK

The closest works in terms of objectives are the ones of W3C [5] and [4]. [5] describes a type system based on forward type inference which is polynomial (except for nested let clauses). Colazzo goes one step further by a thorough analysis of precision and complexity of this type-system and by introducing his own, more precise (but exponential) type system. Both works suffer from the same limitations inherent to forward type inference, as described in Section 1. The work in [4] fails to capture backward axes: the proposed type system only supports `child` and `desc-or-self` axes. Furthermore, it involves a complex decomposition of the "in" clause of the `for` loop, which can be avoided using backward type inference, without loss of precision. The type-system proposed by the W3C has been inspired by the seminal work found in [10], which is itself based on finite tree automata containment [11]. In the programming languages community, the XML type-checking problem has also been studied for other particular domain-specific languages such as CDuce [1], XSLT [13] or with specific transformers like transducers [14, 15]. For a recent survey of related works on type-checking for XML, see [2] and references thereof.

The closest work in terms of technical machinery and spirit is [17] which develops backward type inference for a fragment of XSLT called XSLT0. This fragment only supports the parent-to-child relationship, and its expressive power is incomparable to the much more expressive XQuery fragment we consider here. Furthermore, at the time, the logic proposed in [9] on which we develop the present work was not known. This logic is key for an efficient implementation because it avoids the drawbacks of the automata-based approach described in [17]. Specifically, it overcomes the blow-ups due to tree automata complementation.

The concept of backward type inference was first theoretically introduced in [16]. They propose a theoretical framework based on k-pebble tree transducers, which is a output-tree producing version of finite tree automata. They show that the type inferred as input is definable in MSO (monadic-second order logic). In other terms, there exists a monadic-second order logic formula that describes the in-

put type. Unfortunately, their method relies on a conversion from MSO to finite tree automata which requires non-elementary time complexity, that is, the time complexity is not bound by any fixed number of composition of exponentials. As a result, while this is an interesting theoretical result, this makes any direct implementation out of reach. Similar approaches have been proposed in [12, 15].

6. CONCLUSION

We proposed a backward type inference algorithm and a type-checking method for XQuery programs. The XQuery fragment we consider is equipped with the main control statements (`for` loops, conditionals, `let` binders) as well as backward, forward and recursive navigation performed by XPath expressions. We also propose a prototype implementation of these techniques.

This is a step in the design and development of a new generation of type-checkers for XQuery that: (i) are more precise, (ii) are feasible in practice (implementation is tractable and efficient) and (iii) natively support a larger set of features found in XQuery programs, and in particular backward XPath axes. We believe that we identified the main obstacle in type-checking XQuery, namely the combination of precision and practical feasibility, when the main traits of the XQuery language are considered.

One perspective for further work consists in converting the inferred formula to a XML representation (XML Schema). Another perspective for further work consists in implementing the support of attributes and basic types such as string and integer, which can be modeled in the logic using atomic propositions.

7. REFERENCES

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 51–63, New York, NY, USA, 2003. ACM.

[2] V. Benzaken, G. Castagna, H. Hosoya, B. C. Pierce, and S. Vansummeren. XML typechecking. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 3646–3650. Springer US, 2009.

[3] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C WDt, 2005. `http://www.w3.org/TR/xquery/`.

[4] D. Colazzo and C. Sartiani. Precision and complexity of XQuery type inference. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 89–100, 2011.

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

[6] M. F. Fernández and J. Siméon. Building an extensible xquery engine: Experiences with Galax (extended abstract). In *Database and XML Technologies, Second International XML Database Symposium, XSym 2004, Toronto, Canada*, pages 1–4, 2004.

[7] A. Fokoue, K. Rose, J. Siméon, and L. Villard. Compiling XSLT 2.0 into XQuery 1.0. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 682–691, New York, NY, USA, 2005. ACM.

[8] P. Genevès, N. Layaïda, and V. Quint. Impact of XML schema evolution. *ACM Trans. Internet Technol.*, 11:4:1–4:27, July 2011.

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

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

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

[12] K. Inaba, H. Hosoya, and S. Maneth. Multi-return macro tree transducers. In *Implementation and Applications of Automata, 13th International Conference, CIAA*, pages 102–111, 2008.

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

[14] S. Maneth, A. Berlea, T. Perst, and 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, pages 283–294, New York, NY, USA, 2005. ACM.

[15] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In T. Schwentick and D. Suciu, editors, *ICDT*, volume 4353 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2007.

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

[17] A. Tozawa. Towards static type checking for XSLT. In *ACM Symposium on Document Engineering*, pages 18–27. ACM, 2001.

ELEM-EMPTY
$\bot \hookleftarrow$ <σ>{e}</σ>, ()

EMPTY-EMPTY
$\top \hookleftarrow \epsilon$, ()

SEQ-EMPTY
$$\frac{t_1 \hookleftarrow e_1,\ () \qquad t_2 \hookleftarrow e_2,\ ()}{t_1 \wedge \texttt{somewhere}(t_2) \hookleftarrow e_1, e_2,\ ()}$$

FOR-EMPTY
$$\frac{t_1 \hookleftarrow e_1,\ () \qquad t \xleftarrow{\text{in}} e_1 \qquad t_2 \hookleftarrow e_2,\ ()}{t_1 \vee t_2 \left[{}^{\$v}/_{t}\right] \hookleftarrow \texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2,\ ()}$$

LET-EMPTY
$$\frac{t_1 \xleftarrow{\text{in}} e_1 \qquad t_2 \hookleftarrow e_2,\ ()}{t_2 \left[{}^{\$\overline{v}}/_{t_1}\right] \hookleftarrow \texttt{let}\ \$\overline{v}\ \texttt{:=}\ e_1\ \texttt{return}\ e_2,\ ()}$$

IF-EMPTY
$$\frac{c = \texttt{ctoF}(cond) \qquad \begin{matrix}\neg\texttt{hasEq}(cond)\\ t_1 \hookleftarrow e_1,\ ()\end{matrix} \qquad t_2 \hookleftarrow e_2,\ ()}{c \wedge t_1 \vee \neg c \wedge t_2 \hookleftarrow \texttt{if}\ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\ ()}$$

IF-EMPTY'
$$\frac{\texttt{hasEq}(cond) \qquad t_1 \hookleftarrow e_1,\ () \qquad t_2 \hookleftarrow e_2,\ ()}{t_1 \vee t_2 \hookleftarrow \texttt{if}\ cond\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\ ()}$$

CHILD-EMPTY
$$\frac{\varphi = \langle 1\rangle\, \mu X.\texttt{toF}(n) \vee \langle 2\rangle\, X}{\$v \wedge \neg\varphi \hookleftarrow \$v\ \texttt{child::}n,\ ()}$$

DESC-EMPTY
$$\frac{\varphi = \langle 1\rangle\, \mu X.\texttt{toF}(n) \vee \langle 1\rangle\, X \vee \langle 2\rangle\, X}{\$v \wedge \neg\varphi \hookleftarrow \$v\ \texttt{desc::}n,\ ()}$$

PARENT-EMPTY
$$\frac{\varphi = \mu X.\, \langle \overline{1}\rangle\, \texttt{toF}(n) \vee \langle \overline{2}\rangle\, X}{\$v \wedge \neg\varphi \hookleftarrow \$v\ \texttt{parent::}n,\ ()}$$

ANC-EMPTY
$$\frac{\varphi = \mu X.\, \langle \overline{1}\rangle\, \texttt{toF}(n) \vee \langle \overline{1}\rangle\, X \vee \langle \overline{2}\rangle\, X}{\$v \wedge \neg\varphi \hookleftarrow \$v\ \texttt{anc::}n,\ ()}$$

NSIBL-EMPTY
$$\frac{\varphi = \langle 2\rangle\, \mu X.\texttt{toF}(n) \vee \langle 2\rangle\, X}{\$v \wedge \neg\varphi \hookleftarrow \$v\ \texttt{nsibl::}n,\ ()}$$

PSIBL-EMPTY
$$\frac{\varphi = \langle \overline{2}\rangle\, \mu X.\texttt{toF}(n) \vee \langle \overline{2}\rangle\, X}{\$v \wedge \neg\varphi \hookleftarrow \$v\ \texttt{psibl::}n,\ ()}$$

ROOT-EMPTY
$\bot \hookleftarrow$ root(), ()

VAR-EMPTY
$\bot \hookleftarrow \$v,\ ()$

VAR-EMPTY'
$\$\overline{v} \hookleftarrow \$\overline{v},\ ()$

Figure 7: Rules for the empty sequence as output.

```
Inferring Input Type...
let_elem let $v := root() return
element r {child::* } , r($1, $Epsilon)
  expr_union root() , *($Any, $Any) | EPSILON
    root_elem root() , *($Any, $Any)
    root_elem returns: (~(<-2>T) & ~(<-1>T))
    root_empty returns: false
  expr_union returns:(~(<-2>T) & ~(<-1>T))
  elem_elem element r {child::* } , r($1, $Epsilon)
      expr_union child::* , EPSILON | b($Epsilon, $Epsilon) |
       b($Epsilon, $2)
        expr_union child::* , b($Epsilon, $Epsilon) |
         b($Epsilon, $2)
          child_elem:  $v child::* , b($Epsilon, $Epsilon)
          child_elem returns:(~(<2>T) & b & ~(<1>T) &
           (mu X2.(<-1>X1 | <-2>X2)))
          child_elem:  $v child::* , b($Epsilon, $2)
          child_elem returns:((mu X4.(<-2>X4 | <-1>X1)) &
          <2>(mu X3.((b & <2>X3 & ~(<1>T)) | (b & ~(<2>T)
          & ~(<1>T)))) & b & ~(<1>T))
         expr_union returns:((~(<2>T) & b & ~(<1>T) &
          (mu X2.(<-1>X1 | <-2>X2))) | ((mu X4.(<-2>X4 |
          <-1>X1)) & <2>(mu X3.((b & <2>X3 & ~(<1>T)) |
          (b & ~(<2>T) & ~(<1>T)))) & b & ~(<1>T)))
      expr_union returns:((~(<2>T) & b & ~(<1>T) &
       (mu X2.(<-1>X1 | <-2>X2))) | ((mu X4.(<-2>X4 |
       <-1>X1)) & <2>(mu X3.((b & <2>X3 & ~(<1>T)) |
       (b & ~(<2>T) & ~(<1>T)))) & b & ~(<1>T)))
  elem_elem returns: ((~(<2>T) & b & ~(<1>T) &
    (mu X2.(<-1>X1 | <-2>X2))) | ((mu X4.(<-2>X4 |
    <-1>X1)) & <2>(mu X3.((b & <2>X3 & ~(<1>T)) |
    (b & ~(<2>T) & ~(<1>T)))) & b & ~(<1>T)))
let_elem returns: (((mu X5.(<-1>(~(<-2>T) & ~(<-1>T)) |
  <-2>X5)) & ~(<2>T) & b & ~(<1>T)) | ((mu X6.(<-2>X6 |
   <-1>(~(<-2>T) & ~(<-1>T)))) & <2>(mu X3.((b & <2>X3 &
   ~(<1>T)) | (b & ~(<2>T) & ~(<1>T)))) & b & ~(<1>T)))
```

Figure 12: Tree of Applied Rules.

```
1. ((mu X5.<-1>(~<-2>T & ~<-1>T) | <-2>X5) & ~<2>T & b & ~<1>T)
2. | ((mu X6.<-2>X6 | <-1>(~<-2>T & ~<-1>T)) &
3.   <2>(mu X3.((b & <2>X3 & ~<1>T) | (b & ~<2>T & ~<1>T))) & b & ~<1>
```

Figure 13: Inferred Input Type (`tinf`).