# Mechanizing the Metatheory of mini-XQuery

James Cheney[1] and Christian Urban[2]

[1] University of Edinburgh
[2] TU Munich

**Abstract.** We present a Nominal Isabelle formalization of an expressive core fragment of XQuery, a W3C standard functional language for querying XML documents. Our formalization focuses on results presented in the literature concerning XQuery's operational semantics, typechecking, and optimizations. Our core language, called mini-XQuery, omits many complications of XQuery such as ancestor and sibling axes, recursive types and functions, node identity, and unordered processing modes, but does handle distinctive features of XQuery including monadic comprehensions, downward XPath steps and regular expression types. To our knowledge no language with similar features has been mechanically formalized previously. Our formalization is a first step towards a complete formalization of full XQuery and may also be useful as a benchmark for comparing other mechanized metatheory tools.

## 1 Introduction

The long-term vision of research on mechanized metatheory is to develop practical computer-assisted techniques for designing new programming languages, validating implementations and optimization techniques, and improving the reliability and efficiency of existing languages. To realize this vision, it is important to apply mechanized metatheory tools to real programming languages, not just well-studied core calculi [1, 18, 4, 26]. In this paper, we take first steps toward formalizing and verifying properties of the XQuery language [6]. Formalizing XQuery's semantics and verifying optimization techniques will both stretch the capabilities of mechanized metatheory tools and improve confidence in XQuery-based programs.

Over the last two decades the World Wide Web Consortium (W3C) has promulgated many key standards such as Hypertext Markup Language (HTML) used for Web pages, and the more general Extensible Markup Language (XML) that can be used to exchange data and documents. These standard protocols and data formats help ensure cross-compatibility for Web browsers, servers, and other applications, catalyzing the rapid growth of the Web over the past decades. More recently, the W3C has put a great deal of effort into standardizing languages for querying, transforming, or processing XML data — particularly XPath and XQuery.

XQuery is a flagship W3C standard language for querying XML databases that manage efficient access to large amounts of data in XML form. XQuery can be used to write high-level programs; in fact, some Web applications can be written entirely in XQuery. XQuery is considered particularly suitable for integrating loosely-structured data from diverse sources. Moreover, unlike many calculi used for mechanized metatheory tools to date, the commercial value of XQuery is recognized by industry. Relational database vendors such as IBM and Oracle view XML support as required functionality; over 50 commercial products use XQuery. Another vendor, MarkLogic, has over 180 government and publishing industry clients for its native XML database software. There are also several popular open-source XQuery implementations, such as Galax, MonetDB/XQuery, BaseX and Saxonica.

XML databases offer great potential, but also pose new challenges. Efficient XQuery implementations perform sophisticated optimizations based on equational reasoning about programs [14]. Testing these optimizations can detect bugs, but can never guarantee that all bugs have been eliminated. Moreover, most equational reasoning has been validated at the level of a simplified, purely functional form of XQuery. Equational reasoning about these core languages does not necessarily hold for the full language, in part because full XQuery is not *really* pure: node identity allocation is an effect that can be observed by identity tests or duplicate elimination. Consider the following two XQuery expressions:

```
e1:  e union e         e2:  let $x := e in ($x union $x)
```

Here, `e2` performs less work than `e1` by evaluating `e` only once, so it is tempting to rewrite `e1` to `e2`. This is an example of *common subexpression elimination*, an important optimisation technique. However, naive use of common subexpression elimination for XQuery can produce wrong answers, because `e1` and `e2` are not always equivalent. For example, suppose `e` is an expression such as `<a/>` that constructs a new node. In `e1`, two nodes are created because `<a/>` is evaluated twice yielding two element nodes with distinct node identifiers, while in `e2`, only one new node identifier is created. The `union` operation eliminates duplicate nodes, so `count(<a/> union <a/>) = 2` while `count(let $x := <a/> in $x union $x) = 1`. Such corner cases are unusual; common subexpression elimination and its inverse, inlining, are important for optimizing XQuery programs. This provides a strong motivation for understanding exactly when it is safe to substitute equals for equals in XQuery.

A great deal of research about XQuery (e.g. [14, 12, 8]) has focused on simpler core languages that do not exhibit the above pathological behavior. Thus, as a starting point for formalizing full XQuery, our strategy is to first formalize this well-understood core and conduct mechanically-checked proofs of the main results about it. In this paper, we focus on a Turing-incomplete core language called mini-XQuery that exhibits many of the issues needed to handle full XQuery, but whose semantics is much cleaner and easier to deal with because it omits features such as node identity. Mini-XQuery is nevertheless rich enough to study several previously-developed type systems, equivalence laws, and static analyses, including those of Fernandez et al. [14], Colazzo et al. [12] and Cheney [8]. Moreover, our formalization is compact and could serve as a useful benchmark for comparing other mechanized metatheory tools besides Nominal Isabelle. The formalization is available online [9].

The structure of the rest of this paper is as follows: Section 2 presents the mini-XQuery language we will formalize. Section 3 discusses the basic metatheory, including the operational semantics, type soundness, determinacy and totality. Section 4 presents formalizations of operational equivalences about XQuery, including the laws presented by Fernandez et al. [14]. Section 5 presents the formalization of laws and properties of regular expression subtyping. Section 6 reflects on the formalization process itself; Section 7 presents related and future work and Section 8 concludes.

## 2 Background

*Values* We use a simplified model of XML values, in common with previous work that focuses on the element and text node structure of trees and ignores attributes and other leaf node types. While these details matter in implementations, the main challenges lie in formalizing the handling of elements and text nodes.

$$\hat{v} ::= \texttt{text}\{w\} \mid \texttt{elem}\, l\, \{v\} \qquad v ::= \hat{v}, v \mid \texttt{()}$$

Here, $w, l \in \Sigma^*$ are strings and values $v$ are lists of atomic values $\hat{v}$ which can be text nodes $\texttt{text}\{w\}$ or element nodes $\texttt{elem}\ l\ \{v\}$. We define functions such as $[\hat{v}]$ (which makes an atomic value into a singleton list) and $v \mathbin{@} v'$ (which concatenates two sequences).

*Types* In XQuery, the type system is based on regular expression types and the related notion of subtyping is language inclusion [19]. The syntax of types is as follows:

$$\alpha ::= \texttt{text} \mid \texttt{elem}\, l\, \{\tau\} \mid \texttt{item} \qquad \tau ::= \alpha \mid \texttt{()} \mid \tau_1, \tau_2 \mid \tau_1 | \tau_2 \mid \tau^*$$

Note that we distinguish syntactically between atomic types $\alpha$ that match atomic values, versus general types $\tau$ that represent values $v$. We also omit recursive types; although star types provide a limited form of recursion, our types will always have a bounded nesting depth of element constructors. This is not an essential limitation; however, we chose to focus on non-recursive types in this paper to focus attention on the new issues arising for regular expression types.

The meaning of types is defined using the judgment $v : \tau$ indicating when a value matches a type. (Equivalently, we could write this denotationally as $v \in [\![\tau]\!]$). This judgment is defined as follows:

$$\frac{}{\texttt{text}\{w\} : \texttt{text}} \qquad \frac{v : \tau}{\texttt{elem}\, l\, \{v\} : \texttt{elem}\, l\, \{\tau\}} \qquad \frac{}{\hat{v} : \texttt{item}} \qquad \frac{}{\texttt{()} : \texttt{()}} \qquad \frac{\hat{v} : \alpha}{[\hat{v}] : \alpha}$$

$$\frac{v_1 : \tau_1 \quad v_2 : \tau_2}{v_1 \mathbin{@} v_2 : \tau_1, \tau_2} \qquad \frac{v : \tau_1}{v : \tau_1 | \tau_2} \qquad \frac{v : \tau_2}{v : \tau_1 | \tau_2} \qquad \frac{}{\texttt{()} : \tau^*} \qquad \frac{v : \tau}{v : \tau^*} \qquad \frac{v_1 : \tau^* \quad v_2 : \tau^*}{v_1 \mathbin{@} v_2 : \tau^*}$$

*XPath steps and tests.* We handle the downward XPath axis steps and basic node tests:

$$ax ::= \texttt{self} \mid \texttt{child} \mid \texttt{dos} \qquad \phi ::= \texttt{node}() \mid \texttt{text}() \mid l$$

Axis steps include `self`, which corresponds to the identity relation; `child`, which corresponds to the parent-child relation; and the `dos` or *descendant-or-self* axis which corresponds to the transitive, reflexive closure of `child`. We also consider the basic node tests $\texttt{node}()$ which selects any node, $\texttt{text}()$ which selects text nodes only, and $l$ which selects element nodes with label $l$. Note that the (seemingly superfluous) parentheses on $\texttt{node}()$ and $\texttt{text}()$ are part of the syntax of XPath/XQuery. Also note that the $\texttt{node}()$ test is often abbreviated $*$, and $/\texttt{child} :: \phi$ is often abbreviated as just $/\phi$.

*Expressions* The expressions of mini-XQuery are as follows:

$$\begin{array}{rl} e ::= & \texttt{()} \mid e, e' \mid \texttt{elem}\, l\, \{e\} \mid w \mid x \mid x/ax :: \phi \mid \texttt{let}\, x := e\, \texttt{in}\, e' \\ \mid & \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mid \texttt{for}\, x \in e\, \texttt{return}\, e' \end{array}$$

where again $w, l \in \Sigma^*$ are strings. These include expressions for constructing values, such as the empty sequence `()`, sequential composition $e_1, e_2$, element nodes $\texttt{elem}\ l\ \{e\}$ and string literals $w \in \Sigma^*$, as well as standard variables, let-bindings $\texttt{let}\ x\ :=\ e_1\ \texttt{in}\ e_2$, and conditionals $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. Furthermore, the expression $x/ax\ ::\ \phi$ denotes the result of taking an XPath step from a variable and $\texttt{for}\ x \in e\ \texttt{return}\ e'$ denotes iteration over a value viewed as a sequence. Note that in XQuery source programs it is typical to write programs with compound XPath steps and abbreviations

$$\texttt{for}\, x \in y/a//b\, \texttt{return}\, e$$

However, in XQuery this is desugared to

$$\texttt{for}\, z_1 \in y/a\, \texttt{return}\, \texttt{for}\, z_2 \in z_1/*\ \texttt{return}\, \texttt{for}\, x \in z_2/\texttt{dos}::b\, \texttt{return}\, e$$

where the extra $*$-step is due to the fact that $//b$ really abbreviates the descendant step, which is irreflexive.

*Value and typing contexts.* We write $\gamma$ for value contexts mapping variables to values $v$, and $\Gamma$ for typing context mapping variables to types $\tau$. These are represented as lists of pairs of variables with values or types, such that no variable is repeated. As usual, we need to define validity for such contexts and prove a number of routine properties to ensure that e.g. a variable bound in a valid context has a unique value. We also build validity assumptions into the evaluation and typing judgments (for example, by requiring validity in the rules for base cases such as variables) to decrease the number of explicit validity hypotheses we need to state the main results. We omit the details of this part of the formalization. A value context $\gamma$ is considered well-formed with respect to a typing context $\Gamma$ if they bind the same variables and for each $x$ in their common domain, we have $\gamma(x) : \Gamma(x)$. We write $\gamma : \Gamma$ to indicate that this is the case. Again, in the formalization we need to be more pedantic (e.g. we also require that $\gamma$ and $\Gamma$ bind the same variables in the same order) but for the purposes of exposition these details are omitted.

*Evaluation* The XQuery standard gives an operational semantics, while several papers give a denotational semantics for mini-XQuery-like core languages. Although denotational semantics is attractive for a purely functional, terminating core language such as mini-XQuery, we expect that operational techniques will scale better to handling the full language, so we will use a simplified operational semantics for mini-XQuery.

We define the operational semantics judgments via inference rules as shown in Figure 1. The semantics uses two judgments, one for ordinary expression evaluation $\gamma \models e \Rightarrow v$ and one for iterated evaluation $\gamma, x \in v \models^* e \Rightarrow v'$. Intuitively, the iteration judgment does a list comprehension over the input value list $v$, binding $x$ to each atomic $\hat{v}$, evaluating $e$, and then concatenating the resulting value sequences. It is

$$\boxed{\gamma \models e \Rightarrow v}$$

$$\frac{}{\gamma \models \texttt{()} \Rightarrow \texttt{()}} \qquad \frac{\gamma \models e \Rightarrow v}{\gamma \models \texttt{elem}\; l\; \{e\} \Rightarrow \texttt{elem}\; l\; \{v\}} \qquad \frac{\gamma \models e_1 \Rightarrow v_1 \quad \gamma \models e_2 \Rightarrow v_2}{\gamma \models e_1, e_2 \Rightarrow v_1 \mathbin{@} v_2}$$

$$\frac{}{\gamma \models w \Rightarrow w} \qquad \frac{(x, v) \in \gamma}{\gamma \models x \Rightarrow v} \qquad \frac{}{\gamma \models x/ax :: \phi \Rightarrow (\gamma(x)/ax) :: \phi}$$

$$\frac{\gamma \models e_0 \Rightarrow \hat{v}, v \quad \gamma \models e_1 \Rightarrow v_1}{\gamma \models \texttt{if}\; e_0\; \texttt{then}\; e_1\; \texttt{else}\; e_2 \Rightarrow v_1} \qquad \frac{\gamma \models e_0 \Rightarrow \texttt{()} \quad \gamma \models e_2 \Rightarrow v_2}{\gamma \models \texttt{if}\; e_0\; \texttt{then}\; e_1\; \texttt{else}\; e_2 \Rightarrow v_2}$$

$$\frac{\gamma \models e_1 \Rightarrow v_1 \quad \gamma, x \mapsto v_1 \models e_2 \Rightarrow v_2}{\gamma \models \texttt{let}\; x := e_1\; \texttt{in}\; e_2 \Rightarrow v_2} \qquad \frac{\gamma \models e_1 \Rightarrow v_1 \quad \gamma, x \in v_1 \models^* e_2 \Rightarrow v_2}{\gamma \models \texttt{for}\; x \in e_1\; \texttt{return}\; e_2 \Rightarrow v_2}$$

$$\boxed{\gamma, x \in v \models^* e \Rightarrow v'}$$

$$\frac{}{\gamma, x \in \texttt{()} \models^* e \Rightarrow \texttt{()}}\ \textsc{Nil}^* \qquad \frac{\gamma, x \mapsto \hat{v} \models e \Rightarrow v'}{\gamma, x \in [\hat{v}] \models^* e \Rightarrow v'}\ \textsc{Sng}^*$$

$$\frac{\gamma, x \in v_1 \models^* e \Rightarrow v_1' \quad \gamma, x \in v_2 \models^* e \Rightarrow v_2'}{\gamma, x \in v_1 \mathbin{@} v_2 \models^* e \Rightarrow v_1' \mathbin{@} v_2'}\ \textsc{Seq}^*$$

**Fig. 1.** Large-step operational semantics

an important fact about mini-XQuery that these iterations are completely independent, that is, the sub-computations can be evaluated in any order or in parallel (as long as they are reassembled in the correct order). Also note that conditionals test emptiness.

We use auxiliary functions $v/ax$ and $v :: \phi$ to define the behavior of axis steps and node tests on values. Their definitions are:

$$\begin{aligned}
\hat{v}/\texttt{self} &= [\hat{v}] \\
\texttt{elem}\; l\; \{v\}/\texttt{child} &= v \\
\texttt{text}\{w\}/\texttt{child} &= \texttt{()} \\
\texttt{elem}\; l\; \{v\}/\texttt{dos} &= \texttt{elem}\; l\; \{v\}, (v/\texttt{dos}) \\
\texttt{text}\{w\}/\texttt{dos} &= \texttt{text}\{w\} \\
\texttt{()}/ax &= \texttt{()} \\
(\hat{v}, v)/ax &= \hat{v}/ax \mathbin{@} v/ax
\end{aligned}
\qquad
\begin{aligned}
\hat{v} :: \texttt{node}() &= [\hat{v}] \\
\texttt{elem}\; l\; \{v\} :: l &= \texttt{elem}\; l\; \{v\} \\
\texttt{elem}\; l\; \{v\} :: l' &= \texttt{()} \\
\texttt{text}\{w\} :: l &= \texttt{()} \\
\texttt{elem}\; l\; \{v\}{::}\texttt{text}() &= \texttt{()} \\
\texttt{text}\{w\} :: l &= \texttt{text}\{w\} \\
\texttt{()} :: \phi &= \texttt{()} \\
(\hat{v}, v) :: \phi &= \hat{v} :: \phi \mathbin{@} v :: \phi
\end{aligned}$$

Observe that in both cases, the behavior over value sequences is uniform. In the case of the descendant-or-self axis, we always return the value itself followed by the result of evaluating the descendant-or-self axis on the sequence of children of the node (if any). For example:

$$\begin{aligned}
&\texttt{elem}\; a\; \{\texttt{elem}\; b\; \{\texttt{text}\{"x"\}\}, \texttt{elem}\; c\; \{\texttt{elem}\; b\; \{\texttt{text}\{"y"\}\}\}\}/\texttt{dos} :: b/\texttt{text}() \\
= {} &\texttt{text}\{"x"\}, \texttt{text}\{"y"\}
\end{aligned}$$

This example also illustrates another (minor) simplification in mini-XQuery: we do not normalize values to merge adjacent text nodes.

*Substitution* We define a form of substitution adapted to mini-XQuery as follows:

$$
\begin{aligned}
x[e/x] &= e \\
y[e/x] &= y \\
(x/ax :: \phi)[e/x] &= \texttt{let}\, x := e\, \texttt{in}\, x/ax :: \phi \\
(y/ax :: \phi)[e/x] &= y/ax :: \phi \\
w[e/x] &= w \\
\texttt{()}\,[e/x] &= \texttt{()} \\
(e_1, e_2)[e/x] &= (e_1[e/x], e_2[e/x]) \\
\texttt{elem}\, l\, \{e_0\}[e/x] &= \texttt{elem}\, l\, \{e_0[e/x]\} \\
(\texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2)[e/x] &= \texttt{if}\, e_0[e/x]\, \texttt{then}\, e_1[e/x]\, \texttt{else}\, e_2[e/x] \\
(\texttt{let}\, y := e_1\, \texttt{in}\, e_2)[e/x] &= \texttt{let}\, y := e_1[e/x]\, \texttt{in}\, e_2[e/x] \qquad (y \notin FV(x, e, e_1)) \\
(\texttt{for}\, y \in e_1\, \texttt{return}\, e_2)[e/x] &= \texttt{for}\, y \in e_1[e/x]\, \texttt{return}\, e_2[e/x] \qquad (y \notin FV(x, e, e_1))
\end{aligned}
$$

Note that for variable occurrences in axis steps, we cannot always substitute for the variable, so we simply re-bind it locally. This function can be defined as a total function in Nominal Isabelle as described for similar languages in [24, 26]. It is worth pointing out that this substitution function is not the one that comes “for free” with higher-order abstract syntax [18].

*Type system* The typing rules used for mini-XQuery are shown in Figure 2. These rules include the ordinary expression typing judgment $\Gamma \vdash e : \tau$ and iteration typing judgment $\Gamma, x \in \tau \vdash^* e : \tau'$.

Auxiliary rules for axis and node test typechecking are given in Figure 3. In most cases these simply follow the operational behavior or the structure of the regular expression type. Because we omitted recursive types, we can employ a more precise rule for typechecking descendant-or-self steps: specifically, we symbolically evaluate the descendant-or-self step on the regular expression type. This level of precision is not possible in the presence of recursion, because the resulting language is not necessarily regular. Instead, Colazzo et al.’s $\mu$XQ system [12] simply approximates the descendant-or-self step as $(\alpha_1|\cdots|\alpha_n)^*$, where $\alpha_1, \ldots, \alpha_n$. We believe that the nonrecursive case is common enough to warrant special handling for this increased precision.

Our rules are not the same as the original W3C type system either. The W3C system does not use rules similar to the iteration judgment; instead, when an expression of the form $\texttt{for}\ x \in e_1\ \texttt{return}\ e_2$ is typechecked, the type of $e_1$ is split into a *prime* type $\alpha_1|\cdots|\alpha_n$ and a *quantifier* $q \in \{1, +, *, ?\}$. The body of the loop is then checked with $x$ bound to $\alpha_1|\cdots|\alpha_n$ and the return type is adjusted using $q$.

We believe it is more interesting to prove type soundness for the more precise approach; soundness for the W3C type system can then be proved easily by showing that the type inferred by our system is always a subtype of that inferred by the W3C system. We have not formalized the W3C system or this proof, but this appears straightforward (Colazzo and Sartiani [13] present such a result and discuss a number of related issues concerning the expressiveness of the two systems).

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{()}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{elem}\, l\, \{e\} : \texttt{elem}\, l\, \{\tau\}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e, e' : \tau, \tau'}$$

$$\frac{w \in \Sigma^*}{\Gamma \vdash \texttt{text}\{w\} : \texttt{text}} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash c : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{if}\, c\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau_1 | \tau_2} \qquad \frac{x : \tau \in \Gamma \quad \tau/ax \Rightarrow \tau' \quad \tau' :: \phi \Rightarrow \tau''}{\Gamma \vdash x/ax :: \phi : \tau''}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x \in \tau_1 \vdash^* e_2 : \tau_2}{\Gamma \vdash \texttt{for}\, x \in e_1\, \texttt{return}\, e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

$\boxed{\Gamma, x \in \tau \vdash^* e : \tau'}$

$$\frac{}{\Gamma, x \in \texttt{()} \vdash^* e : \texttt{()}} \qquad \frac{\Gamma, x{:}\alpha \vdash e : \tau}{\Gamma, x \in \alpha \vdash^* e : \tau} \qquad \frac{\Gamma, x \in \tau_1 \vdash^* e : \tau_2}{\Gamma, x \in \tau_1^* \vdash^* e : \tau_2^*}$$

$$\frac{\Gamma, x \in \tau_1 \vdash^* e : \tau_1' \quad \Gamma, x \in \tau_2 \vdash^* e : \tau_2'}{\Gamma, x \in \tau_1, \tau_2 \vdash^* e : \tau_1', \tau_2'} \qquad \frac{\Gamma, x \in \tau_1 \vdash^* e : \tau_1' \quad \Gamma, x \in \tau_2 \vdash^* e : \tau_2'}{\Gamma, x \in \tau_1 | \tau_2 \vdash^* e : \tau_1' | \tau_2'}$$

**Fig. 2.** Query well-formedness rules

## 3 Basic Metatheory

In this section we present some of the basic properties of evaluation needed for the main results in the later sections. Values and value typing have a number of properties that are often needed:

**Lemma 1.** *1. For any $v$, we have $v/\texttt{self} = v = v :: \texttt{node}()$.*
*2. For all $\tau$, there exists $v : \tau$.*
*3. For all $v$, we have $v : \texttt{item}^*$.*

**Theorem 1 (Weakening).** *Assume $\gamma \subseteq \gamma'$. Then:*

1. *If $\gamma \models e \Rightarrow v$ holds and $\gamma \subseteq \gamma'$ then $\gamma' \models e \Rightarrow v$ holds.*
2. *If $x \notin \gamma'$ then $\gamma, x \in v \models^* e \Rightarrow v'$ holds then $\gamma', x \in v \models^* e \Rightarrow v'$ holds.*

**Theorem 2 (Strengthening).** *Suppose $x \notin \gamma_1, \gamma_2$ and $x \notin FV(e)$. Then:*

1. *If $\gamma_1, x \mapsto v_0, \gamma_2 \models e \Rightarrow v$ then $\gamma_1, \gamma_2 \models e \Rightarrow v$.*
2. *If $y \notin \gamma_1, x \mapsto v_0, \gamma_2$ and $\gamma_1, x \mapsto v_0, \gamma_2, y \in v_1 \models^* e \Rightarrow v_2$ then $\gamma_1, x \mapsto v_0, \gamma_2, y \in v_1 \models^* e \Rightarrow v_2$.*

**Theorem 3 (Exchange).**

1. *If $\gamma_1, x \mapsto v_1, y \mapsto v_2, \gamma_2 \models e \Rightarrow v$ then $\gamma_1, y \mapsto v_2, x \mapsto v_1, \gamma_2 \models e \Rightarrow v$.*
2. *If $\gamma_1, x \mapsto v_1, y \mapsto v_2, \gamma_2, z \in v \models^* e \Rightarrow v'$ then $\gamma_1, y \mapsto v_2, x \mapsto v_1, \gamma_2, z \in v \models^* e \Rightarrow v'$.*

$$\boxed{\tau :: \phi \Rightarrow \tau'}$$

$$\frac{}{\alpha :: \texttt{node}() \Rightarrow \alpha} \qquad \frac{}{\texttt{text} :: \texttt{text}() \Rightarrow \texttt{text}} \qquad \frac{}{\texttt{elem}\,l\,\{\tau\} :: \texttt{text}() \Rightarrow \texttt{()}}$$

$$\frac{}{\texttt{text} :: l \Rightarrow \texttt{()}} \qquad \frac{}{\texttt{elem}\,l\,\{\tau\} :: l \Rightarrow \texttt{elem}\,l\,\{\tau\}} \qquad \frac{l \neq l'}{\texttt{elem}\,l\,\{\tau\} :: l' \Rightarrow \texttt{()}}$$

$$\frac{}{\texttt{item} :: \phi \Rightarrow \texttt{item}^*} \qquad \frac{}{\texttt{()} :: \phi \Rightarrow \texttt{()}} \qquad \frac{\tau_1 :: \phi \Rightarrow \tau_1' \quad \tau_2 :: \phi \Rightarrow \tau_2'}{\tau_1, \tau_2 :: \phi \Rightarrow \tau_1', \tau_2'}$$

$$\frac{\tau_1 :: \phi \Rightarrow \tau_1' \quad \tau_2 :: \phi \Rightarrow \tau_2'}{\tau_1 | \tau_2 :: \phi \Rightarrow \tau_1' | \tau_2'} \qquad \frac{\tau_1 :: \phi \Rightarrow \tau_2}{\tau_1^* :: \phi \Rightarrow \tau_2^*}$$

$$\boxed{\tau / \phi \Rightarrow \tau'}$$

$$\frac{}{\alpha/\texttt{self} \Rightarrow \alpha} \qquad \frac{}{\texttt{text/child} \Rightarrow \texttt{()}} \qquad \frac{}{\texttt{elem}\,l\,\{\tau\}\texttt{/child} \Rightarrow \tau} \qquad \frac{}{\texttt{text/dos} \Rightarrow \texttt{text}}$$

$$\frac{\tau/\texttt{dos} \Rightarrow \tau'}{\texttt{elem}\,l\,\{\tau\}\texttt{/dos} \Rightarrow \texttt{elem}\,l\,\{\tau\}, \tau'} \qquad \frac{}{\texttt{item}/ax \Rightarrow \texttt{item}^*} \qquad \frac{}{\texttt{()}/ax \Rightarrow \texttt{()}}$$

$$\frac{\tau_1/ax \Rightarrow \tau_1' \quad \tau_2/ax \Rightarrow \tau_2'}{\tau_1, \tau_2/ax \Rightarrow \tau_1', \tau_2'} \qquad \frac{\tau_1/ax \Rightarrow \tau_1' \quad \tau_2/ax \Rightarrow \tau_2'}{\tau_1 | \tau_2/ax \Rightarrow \tau_1' | \tau_2'} \qquad \frac{\tau_1/ax \Rightarrow \tau_2}{\tau_1^*/ax \Rightarrow \tau_2^*}$$

**Fig. 3.** Auxiliary judgments

Another important property is that the iteration rules are invertible, despite their rather nondeterministic flavor. In particular:

**Theorem 4 (Inversion).**

1. *If* $\gamma, x \in \texttt{()} \vDash^* e \Rightarrow v$ *then* $v = \texttt{()}$.
2. *If* $\gamma, x \in [\hat{v}] \vDash^* e \Rightarrow v'$ *then* $\gamma, x \mapsto \hat{v} \vDash e \Rightarrow v'$.
3. *If* $\gamma, x \in v_1 \ @ \ v_2 \vDash^* e \Rightarrow v$ *then there exist* $v_1', v_2'$ *such that* $v = v_1' \ @ \ v_2'$ *and* $\gamma, x \in v_1 \vDash^* e \Rightarrow v_1'$ *and* $\gamma, x \in v_2 \vDash^* e \Rightarrow v_2'$.

*Proof.* Of these, the inversion of SEQ$^*$ is the most complex. We need to reason carefully by induction on the structure of the derivation, using parts (1) and (2) as well as a number of facts about lists and @. □

A final useful property of the value typing rules is that we can "lift" functions on atomic values to functions on values, preserving typing. Let *f* be a function from atomic values to values, and let *lift f* be the natural extension of *f* to a function on values, given by *lift* $f \ \texttt{()} = \texttt{()}$ and *lift* $f \ (\hat{v_1}, v_2) = f(\hat{v_1}) \ @ \ (\mathit{lift}\ f \ v_2)$. Then:

**Lemma 2 (Star lifting).** *Assume that for all* $v' : \tau_1$ *we have lift* $f \ v' : \tau_2$. *Then if* $v : \tau_1^*$ *then* $(\mathit{lift}\ f) \ v : \tau_2^*$.

### 3.1 Type soundness

We now have enough infrastructure to show type soundness. First, we need to establish soundness properties for variables, axis steps, and iterations:

**Lemma 3.** *1. (Variable soundness) If $\gamma : \Gamma$ then $\gamma(x) : \Gamma(x)$.*

2. *(Axis soundness) If $\tau/ax \Rightarrow \tau'$ and $v : \tau$ then $v/ax : \tau'$.*
3. *(Test soundness) If $\tau :: \phi \Rightarrow \tau'$ and $v : \tau$ then $v :: \phi : \tau'$.*
4. *(Star soundness) Assume that for all $v_1, v_2$ such that $v_1 : \tau_1$ and $\gamma, x \in v_1 \models^* e \Rightarrow v_2$ we have $v_2 : \tau_2$. Then if $v_1' : \tau_1^*$ and $\gamma, x \in v_1' \models^* e \Rightarrow v_2'$ then $v_2' : \tau_2^*$.*

*Proof.* Part (1) is immediate. Parts (2) and (3) are by induction on axis or test typing derivations, using Star Lifting for the case involving $\tau^*$. Part (4) is by induction on value typing judgments, using evaluation inversion principles. □

**Theorem 5 (Type soundness).**

1. *If $\Gamma \vdash e : \tau$ and $\gamma : \Gamma$ and $\gamma \models e \Rightarrow v$ then $v : \tau$.*
2. *If $\Gamma, x \in \tau \vdash^* e : \tau'$ and $\gamma, x \in v \models^* e \Rightarrow v'$ and $\gamma : \Gamma$ and $v : \tau$, then $v' : \tau'$.*

*Proof.* By induction on the typing derivations, using the previous lemma and evaluation inversion principles for the iteration cases. □

In fact, we can also show that all well-formed programs evaluate to a value. Although mini-XQuery has iteration, the iteration is always bounded. Naturally, this property does not carry over to the full language, but it is useful here since it means we can eliminate some termination side-conditions on equivalence laws.

**Lemma 4 (Star convergence).** *Suppose that for all $v_1$, if $v_1 : \tau_1$ then there exists $v_2$ such that $\gamma, x \in v_1 \models^* e \Rightarrow v_2$. Then if $v_1' : \tau_1^*$ then there exists $v_2'$ such that $\gamma, x \in v_1' \models^* e \Rightarrow v_2'$.*

**Theorem 6 (Convergence).**

1. *If $\gamma \vdash e : \tau$ and $\gamma : \Gamma$ then there exists $v : \tau$ such that $\gamma \models e \Rightarrow v$.*
2. *If $\gamma, x \in \tau \vdash^* e : \tau'$ and $\gamma : \Gamma$ and $v : \tau$ then there exists $v'$ such that $\gamma, x \in v \models^* e \Rightarrow v'$.*

### 3.2 Determinacy

The evaluation relation is also deterministic. This is not easy to show directly. Instead, we introduce an alternative evaluation relation that is easy to prove deterministic, and show that it is equivalent to the first presentation. This alternative presentation replaces the Sng* and Seq* rules with the following Cons* rule:

$$\frac{\gamma, x \mapsto \hat{v}_1 \models e \Rightarrow v_1' \quad \gamma, x \in v_2 \models^* e \Rightarrow v_2'}{\gamma, x \in \hat{v}_1, v_2 \models^* e \Rightarrow v_1' \mathbin{@} v_2'} \textsc{Cons}^*$$

**Theorem 7 (Equivalence of presentations).** *The* Cons* *rule is derivable from* Nil*, Sng* *and* Seq**; conversely,* Sng* *and* Seq* *are admissible using* Nil* *and* Cons*.

**Theorem 8 (Determinacy).**

1. *If $\gamma \models e \Rightarrow v$ and $\gamma \models e \Rightarrow v'$ then $v = v'$.*

2. *If $\gamma, x \in v_0 \models^* e \Rightarrow v$ and $\gamma, x \in v_0 \models^* e \Rightarrow v'$ then $v = v'$.*

*Proof.* We first show this for the evaluation relation defined using $\textsc{Nil}^*$ and $\textsc{Cons}^*$, which is straightforward. We then use the previous lemma to transfer the result to the $\textsc{Nil}^*$, $\textsc{Sng}^*$, $\textsc{Seq}^*$ presentation of the rules. □

It is probably not strictly necessary to introduce the second presentation based on $\textsc{Nil}^*$ and $\textsc{Cons}^*$, but it is sometimes convenient to use it since there are fewer cases to cover. Proving the systems equivalent was also useful as a sanity check.

## 4 Operational Equivalences

We define operational equivalence as follows:

$$\Gamma \models e \cong e' \iff \forall \gamma : \Gamma . \gamma \models e \Rightarrow v \iff \gamma \models e' \Rightarrow v$$

This definition suffices for our semantics in the absence of side-effects such as node identifier generation. Note that the context $\Gamma$ is necessary to track the variables of $e$ and $e'$ that are needed for evaluation to be sensible. Without this constraint, proving even simple equivalences such as ForEmpty is nontrivial, since we have to explicitly rename the bound name to avoid names already present in the context. Nevertheless, using an explicit typing context for this is just for convenience (we could also have just used a list of variables), since we can always use the top type $\texttt{item}^*$ for all the variables. Using a typing context $\Gamma$ to impose nontrivial type constraints on the free variables of $e$ also means we can take type information into account when reasoning about equivalence. We give an example at the end of the next section.

One important lemma is that we can drop let-bindings if the value of the bound variable is never used:

**Lemma 5 (Let weakening).** *Suppose $x \notin e_2$. Then $\Gamma \models \texttt{let}\ x := e\ \texttt{in}\ e_2 \cong e_2$.*

To prove operational equivalence laws involving commuting `let` and `for`, we need additional properties of evaluation.

**Lemma 6 (Variable iteration).** *Assume $x \notin \gamma$. Then $\gamma, x \in v \models^* x \Rightarrow v'$ holds if and only if $v = v'$.*

**Lemma 7 (Let iteration).** *Assume $\gamma \models e_1 \Rightarrow v_1$. Then $\gamma, x \mapsto v_1, y \in v \models^* e_2 \Rightarrow v_2$ iff $\gamma, y \in v \models^* \texttt{let}\ x := e_1\ \texttt{in}\ e_2 \Rightarrow v_2$.*

**Lemma 8 (For iteration).** *Assume $x \notin e_1, \gamma$ and $y \notin e_2, \gamma$ and $x \neq y$. Then $\gamma, y \in v \models^* \texttt{for}\ x \in e_1\ \texttt{return}\ e_2 \Rightarrow v_2$ holds iff there exists a $v_1$ such that $\gamma, y \in v \models^* e_1 \Rightarrow v_1$ and $\gamma, x \in v_1 \models^* e_2 \Rightarrow v_2$.*

Using these laws, we can verify all of the standard properties of XQuery expressions discussed in for example [14], and a number of others.

**Theorem 9.** *The operational equivalences and laws listed in Figure 4 are valid for mini-XQuery.*

| | |
|---|---|
| $\Gamma \models \texttt{()}, e \cong e$ | SeqUnitL |
| $\Gamma \models e, \texttt{()} \cong e$ | SeqUnitR |
| $\Gamma \models (e_1, e_2), e_3 \cong e_1, (e_2, e_3)$ | SeqAssoc |
| $\Gamma \models \texttt{for}\, x \in \texttt{()}\ \texttt{return}\, e \cong \texttt{()}$ | ForEmpty |
| $\Gamma \models \texttt{for}\, x \in e\, \texttt{return}\, x \cong e$ | ForVar |
| $\Gamma \models \texttt{for}\, x \in (e_1, e_2)\ \texttt{return}\, e \cong (\texttt{for}\, x \in e_1\, \texttt{return}\, e, \texttt{for}\, x \in e_2\, \texttt{return}\, e)$ | ForSeq |
| $\Gamma \models \texttt{for}\, x \in \texttt{text}\{w\}\ \texttt{return}\, e \cong \texttt{let}\, x := w\ \texttt{in}\, e$ | ForString |
| $\Gamma \models \texttt{for}\, x \in \texttt{elem}\, l\, \{e_1\}\ \texttt{return}\, e_2 \cong \texttt{let}\, x := \texttt{elem}\, l\, \{e_1\}\ \texttt{in}\, e_2$ | ForElem |
| $\Gamma \models \texttt{for}\, x \in (\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2)\ \texttt{return}\, e_0$ | |
| $\quad\cong \texttt{if}\, e\, \texttt{then}\, (\texttt{for}\, x \in e_1\, \texttt{return}\, e_0)\ \texttt{else}\ \texttt{for}\, x \in e_2\, \texttt{return}\, e_0$ | ForCond |
| $\Gamma \models \texttt{for}\, x \in (\texttt{let}\, y := e_1\, \texttt{in}\, e_2)\ \texttt{return}\, e \cong \texttt{let}\, y := e_1\, \texttt{in}\, (\texttt{for}\, x \in e_2\, \texttt{return}\, e)$ | ForLet |
| $\Gamma \models \texttt{for}\, x \in (\texttt{for}\, y \in e_1\, \texttt{return}\, e_2)\ \texttt{return}\, e$ | |
| $\quad\cong \texttt{for}\, y \in e_1\, \texttt{return}\, (\texttt{for}\, x \in e_2\, \texttt{return}\, e)$ | ForFor |
| $\Gamma \models \texttt{if}\ \texttt{()}\ \texttt{then}\, e_1\, \texttt{else}\, e_2 \cong e_2$ | CondEmpty |
| $\Gamma \models \texttt{if}\ \texttt{text}\{w\}\ \texttt{then}\, e_1\, \texttt{else}\, e_2 \cong e_1$ | CondString |
| $\Gamma \models \texttt{if}\ \texttt{elem}\, l\, \{e\}\ \texttt{then}\, e_1\, \texttt{else}\, e_2 \cong e_1$ | CondElem |
| $\Gamma \models \texttt{if}\ (e_1, e_2)\ \texttt{then}\, e'_1\, \texttt{else}\, e'_2 \cong \texttt{if}\, e_1\, \texttt{then}\, e'_1\, \texttt{else}\, (\texttt{if}\, e_2\, \texttt{then}\, e'_1\, \texttt{else}\, e'_2)$ | CondCond |
| $\Gamma \models x/\texttt{self} :: \texttt{node}() \cong x$ | SelfId |
| $\Gamma \models \texttt{let}\, x := e\, \texttt{in}\, x \cong e$ | LetVar |
| $\Gamma \models \texttt{let}\, x := e\, \texttt{in}\ \texttt{elem}\, l\, \{e_0\} \cong \texttt{elem}\, l\, \{\texttt{let}\, x := e\, \texttt{in}\, e_0\}$ | LetElem |
| $\Gamma \models \texttt{let}\, x := e\, \texttt{in}\, (e_1, e_2) \cong (\texttt{let}\, x := e\, \texttt{in}\, e_1, \texttt{let}\, x := e\, \texttt{in}\, e_2)$ | LetSeq |
| $\Gamma \models \texttt{let}\, x := e\, \texttt{in}\, (\texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2)$ | |
| $\quad\cong \texttt{if}\, (\texttt{let}\, x := e\, \texttt{in}\, e_0)\ \texttt{then}\, (\texttt{let}\, x := e\, \texttt{in}\, e_1)\ \texttt{else}\, (\texttt{let}\, x := e\, \texttt{in}\, e_2)$ | LetCond |
| $\Gamma \models \texttt{let}\, x := e\, \texttt{in}\, (\texttt{let}\, y := e_1\, \texttt{in}\, e_2)$ | |
| $\quad\cong \texttt{let}\, y := (\texttt{let}\, x := e\, \texttt{in}\, e_1)\ \texttt{in}\, (\texttt{let}\, x := e\, \texttt{in}\, e_2)$ | LetLet |
| $\Gamma \models \texttt{let}\, x := e\, \texttt{in}\, (\texttt{for}\, y \in e_1\, \texttt{return}\, e_2)$ | |
| $\quad\cong \texttt{for}\, y \in (\texttt{let}\, x := e\, \texttt{in}\, e_1)\ \texttt{return}\, (\texttt{let}\, x := e\, \texttt{in}\, e_2)$ | LetFor |

**Fig. 4.** Verified operational equivalence laws for mini-XQuery.

*Proof.* Most of the laws are straightforward using inversion, weakening, and strengthening. The ForVar, ForLet and ForFor laws require variable, let, and for-iteration respectively. □

We can also verify congruence laws for all of the expression forms. These are shown in Figure 5. Note that in the congruence rules for `let` and `for`, we use the $\texttt{item}^*$ type for the type of $x$ in the extended context, meaning that we must verify that $e_2$ is equivalent to $e'_2$ under all possible values for $x$. The congruence rule for `for` also requires a lemma:

**Lemma 9 (For-iteration congruence).** *Assume $\Gamma, x : \texttt{item}^* \models e \cong e'$ and $\gamma : \Gamma$ where $x \notin \gamma$. Then $\gamma, x \in v \models^* e \Rightarrow v'$ holds if and only if $\gamma, x \in v \models^* e' \Rightarrow v'$ holds.*

**Theorem 10 (Evaluation is a congruence).** *The congruence laws of Figure 5 all hold of operational equivalence.*

Finally, using the equivalence laws, congruences, and the definition of substitution we can prove that inlining is sound in mini-XQuery. Of course, inlining is not sound for

$$\frac{}{\Gamma \vDash e \cong e} \qquad \frac{\Gamma \vDash e' \cong e}{\Gamma \vDash e \cong e'} \qquad \frac{\Gamma \vDash e \cong e' \quad \Gamma \vDash e' \cong e''}{\Gamma \vDash e \cong e''} \qquad \frac{\Gamma \vDash e \cong e'}{\Gamma \vDash \mathtt{elem}\, l\, \{e\} \cong \mathtt{elem}\, l\, \{e'\}}$$

$$\frac{\Gamma \vDash e_1 \cong e_1' \quad \Gamma \vDash e_2 \cong e_2'}{\Gamma \vDash (e_1, e_2) \cong (e_1', e_2')} \qquad \frac{\Gamma \vDash e \cong e' \quad \Gamma \vDash e_1 \cong e_1' \quad \Gamma \vDash e_2 \cong e_2'}{\Gamma \vDash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \cong \mathtt{if}\ e'\ \mathtt{then}\ e_1'\ \mathtt{else}\ e_2'}$$

$$\frac{\Gamma \vDash e_1 \cong e_1' \quad \Gamma, x : \mathtt{item}^* \vDash e_2 \cong e_2'}{\Gamma \vDash \mathtt{let}\ x := e_1\ \mathtt{in}\ e_2 \cong \mathtt{let}\ x := e_1'\ \mathtt{in}\ e_2'}$$

$$\frac{\Gamma \vDash e_1 \cong e_1' \quad \Gamma, x : \mathtt{item}^* \vDash e_2 \cong e_2'}{\Gamma \vDash \mathtt{for}\ x \in e_1\ \mathtt{return}\ e_2 \cong \mathtt{for}\ x \in e_1'\ \mathtt{return}\ e_2'}$$

**Fig. 5.** Congruence laws for mini-XQuery equivalence

full XQuery but nevertheless it is an important optimization, and its soundness proof in mini-XQuery sheds some light on what is needed for specific instances of inlining to be sound in XQuery.

**Theorem 11 (Inlining).** $\Gamma \vDash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \cong e_2[e_1/x]$

*Proof.* By induction on the structure of $e$, using many of the rules in Figure 4, congruence rules, and the definition of substitution. □

## 5 Subtyping

Subtyping is based on containment of regular expression types, that is,

$$\tau <: \tau' \iff \forall v. v : \tau \supset v : \tau'$$

Moreover, we often employ type equivalence $\tau \equiv \tau'$ defined as the symmetric closure of $<:$ (or equivalently, as $\forall v. v : \tau \iff v : \tau'$).

We first establish a number of routine properties of regular expression types, including pre-congruence and $*$-induction rules shown in Figure 6, and equivalence or subtyping laws shown in Figure 7.

Since we do not include an empty type or recursive types that could be used to define empty types, we can show that any subtype of $\mathtt{()}$ is equivalent to $\mathtt{()}$. We first need a few auxiliary properties:

**Lemma 10.** *1. Neither* $\mathtt{()} <: \alpha$ *nor* $\alpha <: \mathtt{()}$ *holds for any atomic type* $\alpha$.
*2. If* $\tau_1, \tau_2 <: \mathtt{()}$ *then* $\tau_1 <: \mathtt{()}$ *and* $\tau_2 <: \mathtt{()}$.
*3. If* $\tau_1 | \tau_2 <: \mathtt{()}$ *then* $\tau_1 <: \mathtt{()}$ *and* $\tau_2 <: \mathtt{()}$.
*4. If* $\tau^* <: \mathtt{()}$ *then* $\tau <: \mathtt{()}$.

**Theorem 12.** *If* $\tau <: \mathtt{()}$ *then* $\tau \equiv \mathtt{()}$.

*Proof.* Proof is by induction on the structure of $\tau$, using the previous lemma to rule out the case $\tau = \alpha$ and to bridge the gap between the assumption and induction hypotheses. Consider the case for $\tau^*$. We can assume that $\tau^* <: \mathtt{()}$ holds and that $\tau <: \mathtt{()}$ implies $\tau \equiv \mathtt{()}$. By part (4) of the lemma, we have that $\tau <: \mathtt{()}$, which implies $\tau \equiv \mathtt{()}$, which is equivalent to $\mathtt{()}^*$ as shown in Figure 7. □

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \qquad \frac{\tau_1 <: \tau_2}{\texttt{elem}\, l\, \{\tau_1\} <: \texttt{elem}\, l\, \{\tau_2\}} \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1, \tau_2 <: \tau_1', \tau_2'}$$

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 | \tau_2 <: \tau_1' | \tau_2'} \qquad \frac{\tau_1 <: \tau_2}{\tau_1^* <: \tau_2^*} \qquad \frac{\texttt{()} <: \tau_2 \quad \tau_1 <: \tau_2 \quad \tau_2, \tau_2 <: \tau_2}{\tau_1^* <: \tau_2}$$

**Fig. 6.** Subtyping congruence rules and $*$-induction

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\tau, \texttt{()} & \equiv & \tau & \tau_1|(\tau_2|\tau_3) & \equiv & (\tau_1|\tau_2)|\tau_3 \\
\texttt{()}, \tau & \equiv & \tau & \tau, (\tau_1|\tau_2) & \equiv & (\tau, \tau_1)|(\tau, \tau_2) \\
\tau, (\tau', \tau'') & \equiv & (\tau, \tau'), \tau'' & (\tau_1|\tau_2), \tau & \equiv & (\tau_1, \tau)|(\tau_2, \tau) \\
\tau_1|\tau_2 & \equiv & \tau_2|\tau_1 & \texttt{elem}\, l\, \{\tau_1|\tau_2\} & \equiv & \texttt{elem}\, l\, \{\tau_1\}|\texttt{elem}\, l\, \{\tau_2\} \\
\tau_1 & <: & \tau_1|\tau_2 & \texttt{()} & <: & \tau^* \\
\tau_2 & <: & \tau_2|\tau_2 & \tau & <: & \tau^* \\
\texttt{()}^* & \equiv & \texttt{()} & \tau^*, \tau^* & <: & \tau^*
\end{array}$$

**Fig. 7.** Subtyping and type equivalence laws.

To conclude the section, we show how types can be used to help optimize programs.

**Theorem 13 (Statically dead code elimination).** *If $\Gamma \vdash e : \texttt{()}$ then $\Gamma \models e \cong \texttt{()}$.*

*Proof.* Let $\gamma : \Gamma$ be given; we must show that $\gamma \models e \Rightarrow v$ holds if and only if $\gamma \models \texttt{()} \Rightarrow v$. For the forward direction of the equivalence, we just use soundness and the fact that `()` is the only value of type `()`. For the reverse direction, we use Convergence to show that $e$ must evaluate to some value, then use soundness again to show that it must be `()`, and finally use inversion to show that the only value `()` can evaluate to is also `()`. □

This suggests the following optimization technique: traverse the term, typecheck each subterm and replace each subterm whose type is `()` with `()`.

## 6 Discussion

Table 1 provides some summary information about the formalization (see also [9]), including the number of lines of proof and number of lemmas for each theory (corresponding to the previous three sections of the paper). We have not attempted to be rigorous about blank lines or comments; the merit of raw lines of proof as a metric is unclear to us (especially across different systems), but these figures could at least provide a rough comparison with other possible formalizations. Many of the more subtle proofs turn out to be short, while longer proofs tend to be mostly "brute force" induction steps for which most cases follow the similar reasoning pattern. We have made no attempt to shorten proofs by leveraging Isabelle's automation beyond the basics, because it makes the behavior (and termination) of proof search tactics much harder to control, but it is plausible that more sophisticated use of Isabelle's existing automation (or use of a different technique altogether) could lead to much shorter proofs.

| Theory | Description | Lines | Lemmas |
|---|---|---|---|
| XQuery | Basic definitions, evaluation metatheory, and type soundness. | 1740 | 95 |
| Equivalence | Operational equivalence laws and congruences | 965 | 45 |
| Subtyping | Properties of subtyping; type-based equivalences | 459 | 42 |

**Table 1.** Overview of the formalization

We have begun to formalize some more complex results such as the admissibility of subsumption (a simplified version of the main result of [8]). We have a partial formalization of the syntactic part of the proof from [8], but the semantic aspects of the proof are proving tricky to formalize — the original proof involves reasoning about various regular expression homomorphisms, some of which are partial. We have already found one minor bug in the proof, and we are investigating workarounds.

## 7 Related work

Among W3C standards, XQuery is distinctive in that formalization of its semantics was integrated into the standardization process from an early stage. Fernandez et al. [14] presented a core XML query language that served as one starting point for XQuery, which included several features not present in XQuery such as pattern matching, while excluding other features such as node identity and schema validation. Siméon and Wadler [23] studied and formalized the behavior of validation in XML Schema and XQuery, identifying some formal properties that helped influence the final design.

The only previous work we are aware of on mechanically checking properties of XML query languages is by Genevés and Vion-Dury [16]. They formalize the XML tree model in Coq and define the semantics of XPath axes (including ancestor and sibling axes that we do not handle), and they formalize some equivalence laws for XPath. However, they do not consider XQuery constructs involving name-binding ($\mathtt{for}, \mathtt{let}$) or construction of new XML document values ($\mathtt{elem}\ l\ \{e\}$). We view their formalization as complementary; ultimately a formalization of full XQuery will have to handle all of the features in their work, all those in this paper, and more.

Malecha et al. [20] formalize an implementation of a core SQL-like relational query language in Coq, including a formalization of the B-tree data structure, and leveraging proof automation techniques available in Coq, as also documented by Chlipala et al. [11]. XQuery can be implemented by translating XML trees and queries to algebraic or relational languages (see e.g. Grust et al. [17] or Ré et al. [21]) and it would be interesting to verify such translations. Rose [22] is developing a rewriting-based compiler for full XQuery. While the aim of this system is to simplify compiler development and experimentation with optimization rules, it is also an attractive starting point for verification.

There are also a number of other results from papers on XQuery that we would like to formalize, including the *path-error analysis* of Colazzo et al. [12]. Also, there are refinements to the typechecking algorithm that we have not verified, including some discussed further in more recent work by Colazzo and Sartiani [13]. More ambitious would be to formalize the W3C XQuery Update Facility [7]. Its semantics is defined informally as part of an ongoing standards process, but Benedikt and Cheney [3] give a candidate operational semantics for a core language based on a recent W3C draft.

Naturally it would also be interesting to extend the regular expression type system to handle recursion; it would be even more interesting to formalize the syntax-oriented algorithm for deciding subtyping of Hosoya et al. [19]. Another interesting direction for future work is extending mini-XQuery with XQuery 1.1 features such as higher-order functions, exceptions, and grouping or aggregation constructs in order to understand how they interact with XQuery's distinctive approach to typechecking.

Nominal Isabelle is an implementation of the nominal abstract syntax approach pioneered by Gabbay and Pitts [15]. Our work employs mature aspects of the Nominal Isabelle infrastructure [24], particularly strong induction principles [25] and inversion principles [5] that enable reasoning about name-binding syntax in a way that parallels on-paper reasoning, and which have been used in a number of other case studies, including various lambda-calculi such as LF [26] and the $\pi$-calculus [4]. More recent work on Nominal Isabelle has aimed at supporting additional name-binding constructs [27], such as simultaneous binding in function definitions, and these features should be very useful in scaling our formalization up to XQuery; conversely, the formalization needs of full XQuery may help motivate further investigation of mechanized metatheory techniques, much as the POPLMark challenge has helped spur research on such tools [1].

While some of the properties we proved are essentially syntactic and ought (in principle) to be formalizable in any mechanized metatheory system, others such as operational equivalence and type equivalence involve a mixture of syntactic and semantic methods, which seems to require the expressiveness of first-order logical definitions, which are not available in certain systems. We expect that it would be straightforward (albeit possibly labor-intensive) to formalize mini-XQuery in Coq using standard techniques [2, 10], but it is not clear to us how one could formalize these results (particularly concerning substitution or inversion principles) in an LF-based system such as Twelf [18].

## 8 Conclusions

XQuery is a compelling target for formalization because it is commercially relevant, there is a growing literature on optimization techniques for XQuery, and there is a detailed (albeit not fully mechanized) formal semantics for XQuery already. In this paper, we have taken an important step towards a complete mechanical formalization of XQuery, by formalizing an expressive core language called mini-XQuery. Although Nominal Isabelle seems well-suited for formalizing mini-XQuery, it is possible that other techniques have equal or greater benefits — in particular, techniques based on higher-order abstract syntax or using more advanced tactic programming — could avoid a great deal of "brute force" proof steps. We invite advocates of other approaches to demonstrate the advantages of their systems using mini-XQuery as a benchmark.

## References

1. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The

POPLMark challenge. In *TPHOLs*, pages 50–65, 2005.
2. B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL*, pages 3–15, 2008.
3. M. Benedikt and J. Cheney. Semantics, types and effects for XML updates. In *DBPL*, pages 1–17, 2009.
4. J. Bengtson and J. Parrow. Formalising the pi-calculus using nominal logic. *Logical Methods in Computer Science*, 5(2), 2008.
5. S. Berghofer and C. Urban. Nominal inversion principles. In *TPHOLs*, pages 71–85, 2008.
6. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery`.
7. D. Chamberlin and J. Robie. XQuery update facility 1.0. W3C Candidate Recommendation, August 2008. `http://www.w3.org/TR/xquery-update-10/`.
8. J. Cheney. Regular expression subtyping for XML query and update languages. In *ESOP*, number 4960 in LNCS, pages 32–46, 2008.
9. J. Cheney and C. Urban. Formalization of mini-XQuery in nominal isabelle. `http://homepages.inf.ed.ac.uk/jcheney/XQuery`.
10. A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, pages 143–156, 2008.
11. A. Chlipala, J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, pages 79–90, 2009.
12. D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.
13. D. Colazzo and C. Sartiani. Precision and complexity of XQuery type inference. In *PPDP*, 2011. To appear. Preliminary version in ICTCS 2010.
14. M. F. Fernandez, J. Siméon, and P. Wadler. A semi-monad for semi-structured data. In *ICDT*, pages 263–300, London, UK, 2001. Springer-Verlag.
15. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
16. P. Genevès and J.-Y. Vion-Dury. XPath formal semantics and beyond: A Coq-based approach. In *TPHOLs Emerging Trends*, pages 181–198, Salt Lake City, Utah, United States, August 2004. University Of Utah.
17. T. Grust, J. Rittinger, and J. Teubner. Pathfinder: XQuery off the relational shelf. *IEEE Data Eng. Bull.*, 31(4), 2008.
18. R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Programming*, 17(4–5):613–673, 2007.
19. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
20. J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *POPL*, pages 237–248, 2010.
21. C. Ré, J. Siméon, and M. F. Fernández. A complete and efficient algebraic compiler for XQuery. In *ICDE*, page 14, 2006.
22. K. Rose. CRSX - combinatory reduction systems with extensions. In *RTA*, 2011.
23. J. Siméon and P. Wadler. The essence of XML. *POPL*, pages 1–13, 2003.
24. C. Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.
25. C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *CADE*, pages 35–50, 2007.
26. C. Urban, J. Cheney, and S. Berghofer. Mechanizing the metatheory of LF. *ACM Trans. Comput. Log.*, 12(2):15, 2011.
27. C. Urban and C. Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. In *ESOP*, pages 480–500, 2011.