Regular Expression Subtyping for XML Query and Update Languages

James Cheney

University of Edinburgh

Abstract. XML database query languages such as XQuery employ regular expression types with structural subtyping. Subtyping systems typically have two presentations, which should be equivalent: a declarative version in which the subsumption rule may be used anywhere, and an algorithmic version in which the use of subsumption is limited in order to make typechecking syntax-directed and decidable. However, the XQuery standard type system circumvents this issue by using imprecise typing rules for iteration constructs and defining only algorithmic typechecking, and another extant proposal provides more precise types for iteration constructs but ignores subtyping. In this paper, we consider a core XQuery-like language with a subsumption rule and prove the completeness of algorithmic typechecking; this is straightforward for XQuery proper but requires some care in the presence of more precise iteration typing disciplines. We extend this result to an XML update language we have introduced in earlier work.

1 Introduction

The Extensible Markup Language (XML) is a World Wide Web Consortium (W3C) standard for tree-structured data. Regular expression types for XML [14] have been studied extensively in XML processing languages such as XDuce [13] and CDuce [1], as well as projects to extend general-purpose programming languages with XML features such as Xtatic [10] and OCamlDuce [9]. Moreover, subtyping (based on regular tree language inclusion) plays an important role in all of these systems.

XQuery is a W3C standard XML database query language [6]. Static typechecking is important in XML database applications because type information is useful for optimizing queries and avoiding expensive run-time checks and revalidation. XQuery provides for static typing using regular expression types and subtyping. However, XQuery's type system is imprecise in some situations involving iteration (`for`-loops). In particular, if the variable $\$x$ has type[1] $a[b[]^*, c[]^?]$, then the query

```
for $y in $x/* return $y
```

is assigned type $(b[]|c[])^*$ in XQuery, but in fact the result will always match the regular expression type $b[]^*, c[]^?$. The reason for this inaccuracy is that XQuery's type system typechecks a `for` loop by converting the type of the body of the expression (here, $\$x/a$ with type $b[]^*, c[]^?$) to the "factored" form $(\alpha_1|\ldots|\alpha_n)^q$, where q is a quantifier such

[1] We use the compact notation for regular expression types introduced by Hosoya, Vouillon and Pierce [14] in preference to the more verbose XQuery or XML Schema syntaxes.

as ?, +, or $*$ and each α_i is an atomic type (i.e. a data type such as `string` or single element type $a[\tau]$).

More precise type systems have been contemplated for XQuery-like languages, including a precursor to XQuery designed by Fernandez, Siméon, and Wadler [8]. Most recently, Colazzo et al. [5] have introduced a core XQuery language called μXQ, with a regular expression-based type system that performs "path correctness" analysis and provides more precise types for iterations using techniques similar to those in [8], but does not support subtyping. In μXQ, the above expression is assigned the more accurate type $b[]^*, c[]^?$; however, the example cannot be assigned the less precise type $(b[]|c[])^*$ since subtyping was not incorporated into the original formulation of μXQ.

Combining subtyping with accurate typing for iteration constructs is especially important for XML updates. We are developing a statically-typed update language called FLUX [2] in which ideas from μXQ are essential for typechecking updates involving iteration. Using XQuery-style factoring for iteration in FLUX would make it impossible to typecheck updates that modify data without modifying the overall schema of the database—a very common case. For example, using XQuery-style factoring for iteration in FLUX, we would not be able to verify statically that given a database of type $a[b[\texttt{string}]^*, c[]^?]$, the result of an update that modifies some b elements and deletes some c elements still has type $a[b[\texttt{string}]^*, c[]^?]$, rather than $a[(b[\texttt{string}]|c[])^*]$.

One question left unresolved in previous work on both μXQ and FLUX is the relationship between declarative and algorithmic presentations of the type system (in the terminology of [16, Ch. 15–16]). Declarative derivations permit arbitrary uses of the *subsumption rule*:

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

whereas algorithmic derivations limit the use of this rule in order to ensure that typechecking is syntax-directed and decidable. The declarative and algorithmic presentations of a system should agree. If they agree, then declarative typechecking is decidable algorithmically; if they disagree, then the algorithmic system is *incomplete*, that is, it rejects programs that should typecheck according to the declarative rules.

The XQuery standard avoided this issue by defining typechecking algorithmically, that is, building subsumption into several rules and omitting a general subsumption rule. Subtyping was omitted from μXQ, because it interferes with μXQ's "path correctness" component [5, Sec. 4.4] . Subtyping was considered in our initial work on FLUX [3], but we were initially unable to establish that typechecking was decidable.

In this paper we develop the foundations of subtyping for XML query and update languages. Our main contributions relative to previous work [5, 3] are definitions and proofs of completeness of algorithmic typechecking (and hence decidability of declarative typechecking) for μXQ and FLUX, extended with subtyping and type, query, and update recursion. We follow the standard technique of proving that declarative derivations can always be normalized to algorithmic derivations [16, Ch. 16]. However, for μXQ's more precise iteration type discipline, completeness of algorithmic typechecking does not follow directly by the obvious structural induction. Instead, we must establish a stronger property based on the semantics of regular expression types.

[2] "FunctionaL Updates for XML"; earlier called LUX ("Lightweight Updates for XML") in [3]

The structure of the rest of the paper is as follows. Section 2 reviews regular expression types and subtyping. Section 3 introduces the core language μXQ, discusses examples highlighting the difficulties involving subtyping in μXQ, and proves decidability of declarative typechecking. We also review the FLUX core update language in Section 4, discuss examples, and extend the proof of decidability of declarative typechecking to FLUX. Sections 5–6 sketch related and future work and conclude. Space limitations preclude a satisfying self-contained exposition of the μXQ and FLUX languages; the reader is encouraged to consult the earlier papers for further details [5, 3].

2 Regular Expression Types and Subtyping

For the purposes of this paper, *XML values* are trees built up out of booleans $b \in \mathit{Bool} = \{\texttt{true}, \texttt{false}\}$, strings $w \in \Sigma^*$ over some alphabet Σ, and labels $l, m, n \in \mathit{Lab}$, according to the following syntax:

$$\bar{v} ::= b \mid w \mid n[v] \qquad v ::= \bar{v}, v \mid \texttt{()}$$

Values include *tree values* $\bar{v} \in \mathit{Tree}$ and *forest values* $v \in \mathit{Val}$. We write v, v' for the result of appending two forest values. This operation is associative with unit $\texttt{()}$.

We consider a regular expression type system with structural subtyping, similar to those considered in several transformation and query languages for XML [14, 5, 8]. The syntax of types and type environments is as follows.

$$\begin{array}{ll}
\text{Atomic types} & \alpha ::= \texttt{bool} \mid \texttt{string} \mid n[\tau] \\
\text{Sequence types} & \tau ::= \alpha \mid \texttt{()} \mid \tau|\tau' \mid \tau, \tau' \mid \tau^* \mid X \\
\text{Type definitions} & \tau_0 ::= \alpha \mid \texttt{()} \mid \tau_0|\tau_0' \mid \tau_0, \tau_0' \mid \tau_0^* \\
\text{Type signatures} & E ::= \cdot \mid E, \texttt{type}\ X = \tau_0
\end{array}$$

We call types of the form $\alpha \in \mathit{Atom}$ *atomic* types (or sometimes tree or singular types), and types $\tau \in \mathit{Type}$ of all other forms *sequence types* (or sometimes forest or plural types). It should be obvious that a value of singular type must always be a sequence of length one (that is, a tree); plural types may have values of any length. There exist plural types with only values of length one, but which are not syntactically singular (for example $\texttt{int}|\texttt{bool}$). As usual, the $+$ and $?$ quantifiers can be defined as follows: $\tau^+ = \tau, \tau^*$ and $\tau^? = \tau|\texttt{()}$. We abbreviate $n[\texttt{()}]$ as $n[]$.

Our type language differs slightly from the standard approaches to regular expression types [14, 5]. In [14], it was shown that Kleene star can be translated away by introducing type variables and definitions, modulo a syntactic restriction on top-level occurrences of type variables in type definitions. We include Kleene star as a primitive, and permit (mutually) recursive type declarations, but forbid any top-level occurrences type variables in definitions τ_0. Therefore Kleene star is *not* definable in terms of the other operations here; this is why we include it as a primitive. For example, $\texttt{type}\ X = \mathit{nil}[]|\mathit{cons}[a, X]$ and $\texttt{type}\ Y = \mathit{leaf}[]|\mathit{node}[Y, Y]$ are allowed but $\texttt{type}\ X' = \texttt{()}|a[], X'$ and $\texttt{type}\ Y' = b[]|Y', Y'$ are not. The equation for X' defines the regular tree language $a[]^*$, and would be permitted in XDuce, while that for Y' defines a context-free tree language that is not regular and is forbidden in XDuce.

An environment E is well-formed if all type variables appearing in definitions are themselves declared in E. Given a well-formed environment E, we write $E(X)$ for the definition of X. A type τ denotes the set of values $[\![\tau]\!]_E$, defined as follows.

$$[\![\texttt{string}]\!]_E = \Sigma^* \qquad [\![\texttt{bool}]\!]_E = \mathit{Bool} \qquad [\![\,\texttt{()}\,]\!]_E = \{\,\texttt{()}\,\}$$
$$[\![n[\tau]]\!]_E = \{n[v] \mid v \in [\![\tau]\!]_E\} \qquad [\![X]\!]_E = [\![E(X)]\!] \quad [\![\tau|\tau']\!]_E = [\![\tau]\!]_E \cup [\![\tau']\!]_E$$
$$[\![\tau, \tau']\!]_E = \{v, v' \mid v \in [\![\tau]\!]_E, v' \in [\![\tau']\!]_E\}$$
$$[\![\tau^*]\!]_E = \{\,\texttt{()}\,\} \cup \{v_1, \ldots, v_n \mid v_1 \in [\![\tau]\!]_E, \ldots, v_n \in [\![\tau]\!]_E\}$$

Formally, $[\![\tau]\!]_E$ is defined by a least fixed point construction which we gloss over. Henceforth, we treat E as fixed and define $[\![\tau]\!] \triangleq [\![\tau]\!]_E$. This semantics validates standard identities such as associativity of ',' ($[\![(\tau_1, \tau_2), \tau_3]\!] = [\![\tau_1, (\tau_2, \tau_3)]\!]$), unit laws ($[\![\tau,\, \texttt{()}\,]\!] = [\![\tau]\!] = [\![\,\texttt{()}\,, \tau]\!]$), and idempotence of '*' ($[\![(\tau^*)^*]\!] = [\![\tau^*]\!]$).

In addition, we define a binary *subtyping* relation on types. A type τ_1 is a subtype of τ_2 ($\tau_1 <: \tau_2$), by definition, if $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. Our types can be translated to XDuce types, so subtyping reduces to XDuce subtyping; although this problem is EXPTIME-complete in general, the algorithm of [14] is well-behaved in practice. Therefore, we shall not give explicit inference rules for checking or deciding subtyping, but treat it as a "black box".

3 Query language

We review an XQuery-like core language based on μXQ [5]. In μXQ, we distinguish between *tree variables* $\bar{x} \in \mathit{TVar}$, introduced by `for`, and *forest variables*, $x \in \mathit{Var}$, introduced by `let`. We write $\hat{x} \in \mathit{Var} \cup \mathit{TVar}$ for an arbitrary variable. The other syntactic classes of our variant of μXQ include booleans, strings, and labels introduced above, function names $F \in \mathit{FSym}$, expressions $e \in \mathit{Expr}$, and programs $p \in \mathit{Prog}$; the abstract syntax of expressions and programs is defined as follows:

$$\begin{aligned} e ::= {} & \texttt{()} \mid e, e' \mid n[e] \mid w \mid x \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid F(e_1, \ldots, e_n) \\ & \mid\ b \mid \texttt{if}\ c\ \texttt{then}\ e\ \texttt{else}\ e' \mid \bar{x} \mid \bar{x}/\texttt{child} \mid e :: n \mid \texttt{for}\ \bar{x} \in e\ \texttt{return}\ e' \\ p ::= {} & \texttt{query}\ e : \tau \mid \texttt{declare function}\ F(x_1{:}\tau_1, \ldots, x_n{:}\tau_n) : \tau\ \{e\}\texttt{;}\ p \end{aligned}$$

The distinguished variables x in $\texttt{let}\ x = e\ \texttt{in}\ e'(x)$ and $\bar{x}$ in $\texttt{for}\ \bar{x} \in e\ \texttt{return}\ e'(\bar{x})$ are bound in $e'(x)$ and $e'(\bar{x})$ respectively. Here and elsewhere, we employ common conventions such as identifying expressions modulo α-renaming.

To simplify the presentation, we split μXQ's projection operation $\bar{x}/\texttt{child} :: l$ into two expressions: child projection ($\bar{x}/\texttt{child}$) which returns the children of $\bar{x}$, and node name filtering ($e :: n$) which evaluates e to an arbitrary sequence and selects the nodes labeled n. Thus, the ordinary child axis expression $\bar{x}/\texttt{child} :: n$ is syntactic sugar for $(\bar{x}/\texttt{child}) :: n$ and the "wildcard" child axis is definable as $\bar{x}/\texttt{child} :: * = \bar{x}/\texttt{child}$. Built-in operations such as string equality may be provided as additional functions F.

Colazzo et al. [5] provided a denotational semantics of μXQ queries with the descendant axis but without recursive functions. This semantics is sound with respect to the typing rules in the next section and can be extended to handle recursive functions using operational techniques (as in the XQuery standard). However, we omit the semantics since it is not needed in the rest of the paper.

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

$$\frac{\bar{x}{:}\alpha \in \Gamma}{\Gamma \vdash \bar{x} : \alpha} \quad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{w \in \Sigma^*}{\Gamma \vdash w : \texttt{string}} \quad \frac{b \in \mathit{Bool}}{\Gamma \vdash b : \texttt{bool}}$$

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{()}} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash n[e] : n[\tau]} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e, e' : \tau, \tau'} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash c : \texttt{bool} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1|\tau_2} \quad \frac{\bar{x}{:}n[\tau] \in \Gamma}{\Gamma \vdash \bar{x}/\texttt{child} : \tau} \quad \frac{\Gamma \vdash e : \tau \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e :: n : \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e_2 : \tau_2}{\Gamma \vdash \texttt{for}\ \bar{x} \in e_1\ \texttt{return}\ e_2 : \tau_2} \quad \frac{F(\overline{\tau}) : \tau_0 \in \Delta \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash F(\overline{e}) : \tau_0} \quad \frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

$\boxed{\Gamma \vdash p\ \texttt{prog}}$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{query}\ e : \tau\ \texttt{prog}} \quad \frac{F \text{ not declared in } p \quad F(\overline{\tau}) : \tau_0 \in \Delta \quad \Gamma, \bar{x} : \overline{\tau} \vdash e : \tau_0 \quad \Gamma \vdash p\ \texttt{prog}}{\Gamma \vdash \texttt{declare function}\ F(\overline{\tau}) : \tau_0\ \{e\};\ p\ \texttt{prog}}$$

Fig. 1. Query and program well-formedness rules

$\boxed{\tau :: n \Rightarrow \tau'}$ $\qquad$ $\boxed{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau \to e : \tau'}$

$$\frac{}{n[\tau] :: n \Rightarrow n[\tau]} \quad \frac{E(X) :: n \Rightarrow \tau}{X :: n \Rightarrow \tau} \quad \frac{\alpha \neq n[\tau]}{\alpha :: n \Rightarrow \texttt{()}} \quad \frac{}{\texttt{()} :: n \Rightarrow \texttt{()}} \quad \frac{\tau_1 :: n \Rightarrow \tau_2}{\tau_1^* :: n \Rightarrow \tau_2^*}$$

$$\frac{\tau_1 :: n \Rightarrow \tau_1' \quad \tau_2 :: n \Rightarrow \tau_2'}{\tau_1, \tau_2 :: n \Rightarrow \tau_1', \tau_2'} \quad \frac{\tau_1 :: n \Rightarrow \tau_1' \quad \tau_2 :: n \Rightarrow \tau_2'}{\tau_1|\tau_2 :: n \Rightarrow \tau_1'|\tau_2'}$$

$$\frac{}{\Gamma \vdash \bar{x}\ \texttt{in}\ \texttt{()} \to e : \texttt{()}} \quad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ E(X) \to e : \tau}{\Gamma \vdash \bar{x}\ \texttt{in}\ X \to e : \tau} \quad \frac{\Gamma, \bar{x}{:}\alpha \vdash e : \tau}{\Gamma \vdash \bar{x}\ \texttt{in}\ \alpha \to e : \tau} \quad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1^* \to e : \tau_2^*}$$

$$\frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_1' \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_2 \to e : \tau_2'}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1, \tau_2 \to e : \tau_1', \tau_2'} \quad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_1' \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_2 \to e : \tau_2'}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1|\tau_2 \to e : \tau_1'|\tau_2'}$$

Fig. 2. Auxiliary judgments

3.1 Type system

Our type system for queries is essentially that introduced for μXQ by [5], excluding the path correctness component. We consider typing environments Γ and global declaration environments Δ, defined as follows:

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \bar{x}{:}\alpha \qquad \Delta ::= \cdot \mid \Delta, F(\overline{\tau}) : \tau_0$$

Note that in Γ, tree variables may only be bound to atomic types. As usual, we assume that variables in type environments are distinct; this convention implicitly constrains all inference rules. We also write $\Gamma <: \Gamma'$ to indicate that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\Gamma'(\hat{x}) <: \Gamma(\hat{x})$ for all $\hat{x} \in \mathrm{dom}(\Gamma)$.

The main typing judgment for queries is $\Gamma \vdash e : \tau$; we also define a program well-formedness judgment $\Gamma \vdash p\ \texttt{prog}$ which typechecks the bodies of functions. Following

[5], there are two auxiliary judgments, $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau \to s : \tau'$, used for typechecking `for`-expressions, and $\tau :: n \Rightarrow \tau'$, used for typechecking label matching expressions $e :: n$. The rules for these judgments are shown in Figures 1 and 2.

We consider the typing rules to be implicitly parameterized by a fixed global declaration environment Δ. Functions in XQuery have global scope so we assume that the declarations for all the functions declared in the program have already been added to Δ by a preprocessing pass. Additional declarations for built-in functions might be included in Δ as well.

The rules involving type variables in Figure 2 look up the variable's definition in E. These judgments only inspect the top-level of a type; they do not inspect the contents of element types $n[\tau]$. Since type definitions τ_0 have no top-level type variables, both judgments are terminating. (This was argued in detail by Colazzo et al. [5, Lem. 4.6].)

3.2 Examples

We first revisit the example in the introduction in order to illustrate the operation of the rules. Recall that $\bar{x}/*$ is translated to $\bar{x}/\texttt{child}$ in our core language.

$$\frac{\overset{\mathcal{D}}{\overline{\Gamma \vdash \bar{x}/\texttt{child} : b[]^*, c[]^?}} \quad \Gamma \vdash \bar{y}\ \texttt{in}\ b[]^*, c[]^? \to \bar{y} : b[]^*, c[]^?}{\Gamma \vdash \texttt{for}\ \bar{y} \in \bar{x}/\texttt{child}\ \texttt{return}\ \bar{y} : b[]^*, c[]^?}$$

where we define $\Gamma = \bar{x}{:}a[b[]^*, c[]^?]$ and subderivation $\mathcal{D}$ is

$$\mathcal{D} = \frac{\dfrac{\dfrac{\overline{\Gamma, \bar{y}{:}b[] \vdash \bar{y} : b[]}}{\Gamma \vdash \bar{y}\ \texttt{in}\ b[] \to \bar{y} : b[]}}{\Gamma \vdash \bar{y}\ \texttt{in}\ b[]^* \to \bar{y} : b[]^*} \quad \dfrac{\dfrac{\overline{\bar{x}{:}a[b[]^*, c[]], \bar{y}{:}c[] \vdash \bar{y} : c[]}}{\Gamma \vdash \bar{y}\ \texttt{in}\ c[] \to \bar{y} : c[]}}{\Gamma \vdash \bar{y}\ \texttt{in}\ c[]^? \to \bar{y} : c[]^?}}{\Gamma \vdash \bar{y}\ \texttt{in}\ b[]^*, c[]^? \to \bar{y} : b[]^*, c[]^?}$$

Note that this derivation does not use subsumption anywhere. Suppose we wished to show that the expression has type $b[]^*, (c[]^?|d[]^*)$, a supertype of the above type. There are several ways to do this. We could simply use subsumption at the end of the derivation. Alternatively, we could have used subsumption in one of the subderivations such as $\Gamma, \bar{y}{:}c[]^? \vdash \bar{y} : c[]^?$, to conclude, for example, that $\Gamma, \bar{y}{:}c[]^? \vdash \bar{y} : c[]^?|d[]^*$. This is valid since $c[]^? <: c[]^?|d[]^*$.

Suppose, instead, that we actually wanted to show that the above expression has type $(b[d[]^*]|c[]^?)^*$, also a supertype of the derived type. There are again several ways of doing this. Besides using subsumption at the end of the derivation, we could use it on $\Gamma \vdash \bar{x}/\texttt{child} : b[]^*, c[]^?$ to obtain $\Gamma \vdash \bar{x}/\texttt{child} : (b[d[]^*]|c[]^?)^*$. To complete the derivation, we would then need to replace derivation $\mathcal{D}$ with $\mathcal{D}'$:

$$\mathcal{D}' = \frac{\dfrac{\dfrac{\overline{\Gamma, \bar{y}{:}b[d[]^*] \vdash \bar{y} : b[d[]^*]}}{\Gamma \vdash \bar{y}\ \texttt{in}\ b[d[]^*] \to \bar{y} : b[d[]^*]} \quad \dfrac{\dfrac{\overline{\Gamma, \bar{y}{:}c[] \vdash \bar{y} : c[]}}{\Gamma \vdash \bar{y}\ \texttt{in}\ c[] \to \bar{y} : c[]}}{\Gamma \vdash \bar{y}\ \texttt{in}\ c[]^? \to \bar{y} : c[]^?}}{\Gamma \vdash \bar{y}\ \texttt{in}\ b[d[]^*]|c[]^? \to \bar{y} : b[d[]^*]|c[]^?}}{\Gamma \vdash \bar{y}\ \texttt{in}\ (b[d[]^*]|c[]^?)^* \to \bar{y} : (b[d[]^*]|c[]^?)^*}$$

Not only does $\mathcal{D}'$ have different structure than $\mathcal{D}$, but it also requires subderivations that were not syntactically present in $\mathcal{D}$.

The above example illustrates why eliminating uses of subsumption is tricky. If subsumption is used to weaken the type of the first argument of a `for`-expression according to $\tau_1' <: \tau_1$, then we need to know that we can transform the corresponding derivation $\mathcal{D}$ of $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2$ to a derivation of $\mathcal{D}'$ of $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1' \to e : \tau_2'$ for some $\tau_2' <: \tau_2$. But the derivations $\mathcal{D}$ and $\mathcal{D}'$ may bear little resemblance to one another.

Now we consider a typechecking a recursive query. Suppose we have[3] $\texttt{type}\ \mathit{Tree} = \mathit{tree}[\mathit{leaf}[\texttt{string}]|\mathit{node}[\mathit{Tree}^*]]$ and function definition

$$\begin{array}{l}\texttt{declare function}\ \mathit{leaves}(x : \mathit{Tree}) : \mathit{leaf}[\texttt{string}]^* \ \{\\ \quad x/\mathit{leaf}, \texttt{for}\ \bar{z} \in x/\mathit{node}/\mathit{tree}\ \texttt{return}\ \mathit{leaves}(\bar{z})\\ \};\end{array}$$

This uses a construct e/n that is not in core μXQ, but we can expand e/n to $\texttt{for}\ \bar{y} \in e\ \texttt{return}\ \bar{y}/\texttt{child} :: n$; thus, we can derive a rule

$$\frac{\Gamma \vdash e : l[\tau] \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e/n : \tau'} \iff \frac{\Gamma \vdash e : l[\tau] \quad \dfrac{\dfrac{}{\Gamma, \bar{y}{:}l[\tau] \vdash \bar{y}/\texttt{child} : \tau} \quad \tau :: n \Rightarrow \tau'}{\Gamma, \bar{y}{:}l[\tau] \vdash \bar{y}/\texttt{child} :: n : \tau'}}{\Gamma \vdash \texttt{for}\ \bar{y} \in e\ \texttt{return}\ \bar{y}/\texttt{child} :: n : \tau'}$$

Using this derived rule and the fact that $x : \mathit{Tree}$ and the definition of Tree, we can see that $x/\mathit{leaf} : \mathit{leaf}[\texttt{string}]$ and $x/\mathit{node} : \mathit{node}[\mathit{Tree}^*]$, and so $x/\mathit{node}/\mathit{tree} : \mathit{tree}[\mathit{leaf}[\texttt{string}]|\mathit{node}[\mathit{Tree}^*]]^*$. So the body of the `for`-loop can be typechecked with $\bar{z} : \mathit{tree}[\mathit{leaf}[\texttt{string}]|\mathit{node}[\mathit{Tree}^*]]$. To check the function call $\mathit{leaves}(\bar{z})$, we need subsumption to see that $\mathit{tree}[\mathit{leaf}[\texttt{string}]|\mathit{node}[\mathit{Tree}^*]] <: \mathit{Tree}$. It follows that $\mathit{leaves}(\bar{z}) : \mathit{leaf}[\texttt{string}]^*$, so the `for`-loop has type $(\mathit{leaf}[\texttt{string}]^*)^*$. Again using subsumption, we can conclude that

$$x/\mathit{leaf}, \mathit{leaves}(x/\mathit{node}/\mathit{tree}) : \mathit{leaf}[\texttt{string}], (\mathit{leaf}[\texttt{string}]^*)^* <: \mathit{leaf}[\texttt{string}]^* .$$

Notice that although we could have used subsumption in several more places, we really *needed* it in only two places: when typechecking a function call, and when checking the result of a function against its declared type.

3.3 Algorithmic Completeness and Decidability

The standard approach (see e.g. Pierce [16, Ch. 16]) to deciding declarative typechecking is to define algorithmic judgments that are syntax-directed and decidable, and then show that the algorithmic system is complete relative to the declarative system.

Definition 1 (Algorithmic derivations). *The algorithmic typechecking judgments $\Gamma \vdash\!\blacktriangleright e : \tau$ and $\Gamma \vdash\!\blacktriangleright \bar{x}\ \texttt{in}\ \tau_0 \to e : \tau$ are defined by taking the rules of Figures 1 and 2, removing the subsumption rule, and replacing the function application rule with*

$$\frac{F(\overline{\tau}) : \tau \in \Gamma \quad \Gamma \vdash\!\blacktriangleright e_i : \tau_i' \quad \tau_i' <: \tau_i}{\Gamma \vdash\!\blacktriangleright F(\overline{e}) : \tau}$$

[3] We use a somewhat artificial definition of Tree here to simplify the example.

It is straightforward to show that algorithmic derivability is decidable and sound with respect to the declarative system:

Lemma 1 (Decidability). *For any $\bar{x}, e, n$, there exist computable partial functions f_n, g_e, $h_{\bar{x},y}$ such that for any Γ, τ_0, we have:*

1. *$f_n(\tau_0)$ is the unique τ such that $\tau_0 :: n \Rightarrow \tau$.*
2. *$g_x(\Gamma)$ is the unique τ such that $\Gamma \vdash\!\blacktriangleright e : \tau$, when it exists.*
3. *$h_{\bar{x},e}(\Gamma, \tau_0)$ is the unique τ such that $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_0 \to e : \tau$, when it exists.*

Theorem 1 (Algorithmic Soundness). *(1) If $\Gamma \vdash\!\blacktriangleright e : \tau$ is derivable then $\Gamma \vdash e : \tau$ is derivable. (2) If $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_0 \to e : \tau$ is derivable then $\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_0 \to e : \tau$ is derivable.*

The main result of this section is the corresponding completeness property (Theorem 2 below). A typical proof of completeness involves showing by induction that occurrences of the subsumption rule can be "permuted" downwards in the proof past other rules, except for function applications where subtyping checks are performed. Completeness for μXQ requires strengthening this induction hypothesis. To see why, consider the rules:

$$\frac{\Gamma \vdash \overset{*}{e_1} : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash \overset{*}{e_1} : \tau_1 \quad \Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1 \to e_2 : \tau_2}{\Gamma \vdash \mathtt{for}\ \bar{x} \in e_1\ \mathtt{return}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash \overset{*}{e} : \tau \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e :: n : \tau'}$$

If the subderivation labeled $*$ in the above rules follows by subsumption, however, we cannot do anything to get rid of the subsumption rule using the induction hypotheses provided by Theorem 2. Instead we need an additional lemma that ensures that the judgments are all *downward monotonic*. Downward monotonicity means, informally, that if we replace the "input" types (including those in Γ) in a derivable judgment with subtypes, then the judgment remains derivable with a smaller "output" type.

The downward monotonicity property (Lemma 3 below) is *almost* easy to prove by direct structural induction (simultaneously on all judgments). The cases involving expression-directed typechecking rules are all straightforward inductive steps; however, for the cases involving type-directed judgments, the induction steps do not go through. The difficulty is illustrated by the following cases. For derivations of the form

$$\frac{\tau_1 :: n \Rightarrow \tau_2}{\tau_1^* :: n \Rightarrow \tau_2^*} \qquad \frac{\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2}{\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1^* \to e : \tau_2^*}$$

we are stuck: knowing that $\tau_1' <: \tau_1^*$ does not necessarily tell us anything about a subtyping relationship between τ_1' and τ_1. For example, if $\tau_1' = aa$ and $\tau_1 = a$, then we have $aa <: a^*$ but not $aa <: a$. Instead, we need to proceed by an analysis of the semantics of regular expression types and subtyping.

We briefly sketch the argument, which involves an excursion into the theory of regular languages over partially ordered alphabets. Here, the "alphabet" is the set of atomic types and the regular sets are the sets of sequences of atomic types that are subtypes of a type τ. The *homomorphic extension* of a (possibly partial) function $h : \mathit{Atom} \rightharpoonup \mathit{Type}$ on atomic types is defined as

$$\hat{h}(\,()\,) = () \qquad \hat{h}(\alpha) = h(\alpha) \qquad \hat{h}(\tau^*) = \hat{h}(\tau)^*$$
$$\hat{h}(\tau_1, \tau_2) = \hat{h}(\tau_1), \hat{h}(\tau_2) \quad \hat{h}(\tau_1|\tau_2) = \hat{h}(\tau_1)|\hat{h}(\tau_2) \quad \hat{h}(X) = \hat{h}(E(X))$$

(Note again that this definition is well-founded, since top-level type variables cannot be expanded indefinitely.) If h is partial, then $\hat{h}$ is defined only on types whose atoms are in $\mathrm{dom}(h)$. We can then show the following general property of partial homomorphic extensions. Detailed proofs are in a companion technical report [4].

Lemma 2. *If $h : Atom \rightharpoonup Type$ is downward monotonic, then its homomorphic extension $\hat{h} : Type \rightharpoonup Type$ is downward monotonic.*

Lemma 3 (Downward monotonicity). *(1) If $\tau_1 :: n \Rightarrow \tau_2$ and $\tau_1' <: \tau_1$ then $\tau_1' :: n \Rightarrow \tau_2'$ for some $\tau_2' <: \tau_2$. (2) If $\Gamma \vdash\!\blacktriangleright e : \tau$ and $\Gamma' <: \Gamma$ then $\Gamma' \vdash\!\blacktriangleright e : \tau'$ for some $\tau' <: \tau$. (3) If $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_1' \to e : \tau_2'$ for some $\tau_2' <: \tau_2$.*

Proof (Sketch). We work in terms of the partial functions f_n, g_e, and $h_{\bar{x},e}$ from Theorem 1. The lemma follows from the downward monotonicity of f_n, g_e, and $h_{\bar{x},e}$ in their type and context arguments. For (1), we show that $f_n = \hat{F}_n$ where $F_n(\alpha) = n[\tau]$ if $\alpha = n[\tau]$, $F_n(\alpha) = \mathtt{()}$ otherwise; observe that F_n is total and monotone. For parts (2) and (3), we strengthen the induction hypothesis by showing that g_e is downward monotonic and that $h_{\bar{x},e}(\Gamma, -) = \hat{g}_e(\Gamma, x{:}(-))$ by simultaneous induction on the structure of algorithmic derivations. The downward monotonicity of $h_{\bar{x},e}(\Gamma, -)$ (which is needed in part (2)) follows again from Lemma 3.

Theorem 2 (Algorithmic Completeness). *(1) If $\Gamma \vdash e : \tau$ then there exists $\tau' <: \tau$ such that $\Gamma \vdash\!\blacktriangleright e : \tau'$. (2) If $\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2$ then there exists $\tau_2' <: \tau_2$ such that $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2'$.*

Proof. Induction on the structure of derivations, appealing to Lemma 3 as necessary.

4 Update language

We now introduce the core FLUX update language, which extends the syntax of queries with statements $s \in Stmt$, procedure names $P \in PSym$, tests $\phi \in Test$, directions $d \in Dir$, and two new cases for programs:

$$
\begin{array}{l}
s ::= \mathtt{skip} \mid s; s' \mid \mathtt{if}\ e\ \mathtt{then}\ s\ \mathtt{else}\ s' \mid \mathtt{let}\ x = e\ \mathtt{in}\ s \mid P(\overline{e}) \\
\quad \mid\ \mathtt{insert}\ e \mid \mathtt{delete} \mid \mathtt{rename}\ n \mid \mathtt{snapshot}\ x\ \mathtt{in}\ s \mid \phi?s \mid d[s] \\
\phi ::= n \mid * \mid \mathtt{bool} \mid \mathtt{string} \qquad d ::= \mathtt{left} \mid \mathtt{right} \mid \mathtt{children} \mid \mathtt{iter} \\
p ::= \cdots \mid \mathtt{update}\ s : \tau \Rightarrow \tau' \mid \mathtt{declare\ procedure}\ P(\overline{x} : \overline{\tau}) : \tau \Rightarrow \tau'\ \{s\}\mathtt{;}\ p
\end{array}
$$

FLUX is based on a novel *functional*, *local* approach to updates which carefully controls side-effects; it is based on ideas from a database update language called CPL+ introduced by Liefke and Davidson [15]. Each update statement operates on a part of the mutable store (or database) that is "in focus". This locality helps ensure that updates are deterministic and relatively easy to typecheck.

Updates include standard constructs such as the no-op `skip`, sequential composition, conditionals, and `let`-binding. *Atomic updates* directly modify the focused part of

$$\boxed{\sigma; v \vdash s \Rightarrow^{\mathtt{U}} v'}$$

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma;\ () \vdash \mathtt{insert}\ e \Rightarrow^{\mathtt{U}} v} \qquad \frac{\sigma[x := v]; v \vdash s \Rightarrow^{\mathtt{U}} v'}{\sigma; v \vdash \mathtt{snapshot}\ x\ \mathtt{in}\ s \Rightarrow^{\mathtt{U}} v'} \qquad \frac{\sigma; v \vdash s \Rightarrow^{\mathtt{U}} v'}{\sigma; n[v] \vdash \mathtt{children}[s] \Rightarrow^{\mathtt{U}} n[v']}$$

$$\frac{\sigma;\ () \vdash s \Rightarrow^{\mathtt{U}} v'}{\sigma; v \vdash \mathtt{left}[s] \Rightarrow^{\mathtt{U}} v', v} \qquad \frac{\sigma; t_1 \vdash S \Rightarrow^{\mathtt{U}} v_1' \quad \sigma; v_2 \vdash \mathtt{iter}[s] \Rightarrow^{\mathtt{U}} v_2'}{\sigma; t_1, v_2 \vdash \mathtt{iter}[s] \Rightarrow^{\mathtt{U}} v_1', v_2'} \qquad \frac{}{\sigma;\ () \vdash \mathtt{iter}[s] \Rightarrow^{\mathtt{U}}\ ()}$$

Fig. 3. Operational semantics of selected updates

the tree. The atomic update operations include insertion `insert` e, which inserts a value into an empty input; deletion `delete`, which deletes the focused input; and `rename` n, which renames the focused input provided it is a single tree.

Tests are operations $\phi?s$ that perform s if the type of the input focus matches the type test ϕ, otherwise do nothing. The node label test n matches tree type $n[\tau]$; the wildcard test $*$ matches tree types $m[\tau]$ for any m; and tests `bool` and `string` match the respective base types. The ? operator binds tightly; for example, $\phi?s; s' = (\phi?s); s'$.

The navigation updates $d[s]$ move the focus to another (smaller) part of the tree, and perform s on the new focus. The `left` and `right` directions focus on the empty sequence "before" or "after" the current focus, which may be a sequence. The `children` direction focuses on the child sequence of a tree. The `iter` direction focuses on each singular value in a sequence.

The `snapshot` operation `snapshot` x `in` s binds x to the input focus value and then applies an update s. Note that `snapshot` is the only way to read from the mutable store, and that the value of x is immutable, so no aliasing ensues.

We lack space to formalize the full semantics of updates. Figure 3 shows some illustrative operational semantics rules, defining the judgment $\sigma; v \vdash s \Rightarrow^{\mathtt{U}} v'$ whose informal meaning is "given immutable environment σ, s updates mutable store v to v'". Here, σ is an environment mapping (tree) variables to (tree) values. The remaining rules, along with additional explanation and examples, may be found in [3].

We distinguish between *singular* (unary) updates which apply only when the context is a tree value and *plural* (multi-ary) updates which apply to a sequence. Tests $\phi?s$ are always singular. The `children` operator applies a plural update to all of the children of a single node; the `iter` operator applies a singular update to all of the elements of a sequence. Other updates can be either singular or plural in different situations. Our type system tracks multiplicity as well as input and output types in order to ensure that updates are well-behaved.

4.1 Type system

In typechecking updates, we extend the global declaration context Δ with procedure declarations:

$$\Delta ::= \cdots \mid \Delta, P(\overline{\tau}) : \tau_1 \Rightarrow \tau_2$$

There are two typing judgments for updates: singular well-formedness $\Gamma \vdash^{1} \{\alpha\}\ s\ \{\tau'\}$ (that is, in type environment Γ, update s maps tree type α to type τ'), and plural well-

$\boxed{\Gamma \vdash^a \{\tau\}\ s\ \{\tau'\}}$

$$\frac{}{\Gamma \vdash^a \{\tau\}\ \texttt{skip}\ \{\tau\}} \quad \frac{\Gamma \vdash^a \{\tau\}\ s\ \{\tau'\} \quad \Gamma \vdash^a \{\tau'\}\ s'\ \{\tau''\}}{\Gamma \vdash^a \{\tau\}\ s; s'\ \{\tau''\}} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash^a \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash^a \{\tau_1\}\ \texttt{let}\ x = e\ \texttt{in}\ s\ \{\tau_2\}}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash^a \{\tau\}\ s\ \{\tau_1\} \quad \Gamma \vdash^a \{\tau\}\ s'\ \{\tau_2\}}{\Gamma \vdash^a \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s'\ \{\tau_1|\tau_2\}} \quad \frac{\Gamma, x{:}\tau \vdash^a \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^a \{\tau\}\ \texttt{snapshot}\ x\ \texttt{in}\ s\ \{\tau'\}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash^* \{()\}\ \texttt{insert}\ e\ \{\tau\}} \quad \frac{}{\Gamma \vdash^a \{\tau\}\ \texttt{delete}\ \{()\}} \quad \frac{}{\Gamma \vdash^1 \{n'[\tau]\}\ \texttt{rename}\ n\ \{n[\tau]\}}$$

$$\frac{\alpha <: \phi \quad \Gamma \vdash^1 \{\alpha\}\ s\ \{\tau\}}{\Gamma \vdash^1 \{\alpha\}\ \phi?s\ \{\tau\}} \quad \frac{\alpha \not<: \phi}{\Gamma \vdash^1 \{\alpha\}\ \phi?s\ \{\alpha\}} \quad \frac{\Gamma \vdash^* \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^1 \{n[\tau]\}\ \texttt{children}[s]\ \{n[\tau']\}}$$

$$\frac{\Gamma \vdash^* \{()\}\ s\ \{\tau'\}}{\Gamma \vdash^a \{\tau\}\ \texttt{left}[s]\ \{\tau', \tau\}} \quad \frac{\Gamma \vdash^* \{()\}\ s\ \{\tau'\}}{\Gamma \vdash^a \{\tau\}\ \texttt{right}[s]\ \{\tau, \tau'\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^* \{\tau\}\ \texttt{iter}[s]\ \{\tau'\}}$$

$$\frac{\Gamma \vdash^a \{\tau_1\}\ s\ \{\tau_2'\} \quad \tau_2' <: \tau_2}{\Gamma \vdash^a \{\tau_1\}\ s\ \{\tau_2\}} \quad \frac{P(\overline{\tau}) : \sigma \Rightarrow \sigma_2 \in \Delta \quad \sigma_1 <: \sigma \quad \Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash^a \{\sigma_1\}\ P(\overline{e})\ \{\sigma_2\}}$$

$\boxed{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}}$

$$\frac{}{\Gamma \vdash_{\texttt{iter}} \{()\}\ s\ \{()\}} \quad \frac{\Gamma \vdash^1 \{\alpha\}\ s\ \{\tau\}}{\Gamma \vdash_{\texttt{iter}} \{\alpha\}\ s\ \{\tau\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{E(X)\}\ s\ \{\tau\}}{\Gamma \vdash_{\texttt{iter}} \{X\}\ s\ \{\tau\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1^*\}\ s\ \{\tau_2^*\}}$$

$$\frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_1'\} \quad \Gamma \vdash_{\texttt{iter}} \{\tau_2\}\ s\ \{\tau_2'\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1, \tau_2\}\ s\ \{\tau_1', \tau_2'\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_1'\} \quad \Gamma \vdash_{\texttt{iter}} \{\tau_2\}\ s\ \{\tau_2'\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1|\tau_2\}\ s\ \{\tau_1'|\tau_2'\}}$$

$\boxed{\Gamma \vdash p\ \texttt{prog}}$

$$\frac{\Gamma \vdash^* \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash \texttt{update}\ s : \tau_1 \Rightarrow \tau_2\ \texttt{prog}} \quad \frac{\begin{array}{c}P \text{ not declared in } p \\ P(\overline{\tau}) : \sigma_1 \Rightarrow \sigma_2 \in \Delta \quad \Gamma, \overline{x}{:}\overline{\tau} \vdash^* \{\sigma_1\}\ s\ \{\sigma_2\} \quad \Gamma \vdash p\ \texttt{prog}\end{array}}{\Gamma \vdash \texttt{declare procedure}\ P(\overline{x} : \overline{\tau}) : \tau_1 \Rightarrow \tau_2\ \{s\};\ p\ \texttt{prog}}$$

Fig. 4. Update and additional program well-formedness rules

formedness $\Gamma \vdash^* \{\tau\}\ s\ \{\tau'\}$ (that is, in type environment Γ, update s maps type τ to type τ'). Several of the rules are parameterized by a multiplicity $a \in \{1, *\}$. In addition, there is an auxiliary judgment $\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}$ for typechecking iterations. The rules for update well-formedness are shown in Figure 4. We also need an auxiliary subtyping relation involving atomic types and tests: we say that $\alpha <: \phi$ if $[\![\alpha]\!] \subseteq [\![\phi]\!]$. This is characterized by the rules:

$$\frac{}{\texttt{bool} <: \texttt{bool}} \quad \frac{}{\texttt{string} <: \texttt{string}} \quad \frac{}{n[\tau] <: n} \quad \frac{}{n[\tau] <: *}$$

Remark 1. In most other XML update proposals (including XQuery! [12] and the draft XQuery Update Facility [2]), side-effecting update operations are treated as *expressions* that return `()`. Thus, we could perhaps typecheck such updates as expressions of type `()`. This would work fine as long as the values reachable from the free variables in Γ never change; however, the updates available in these languages can and do change the

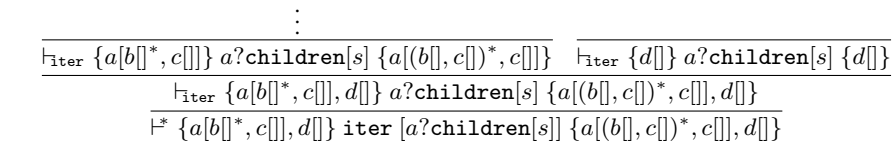

Fig. 5. Example partial update derivation, where $s = \texttt{iter}\ [b?\texttt{right}\ \texttt{insert}\ c[]]$

values of variables. Thus, to make this approach sound Γ may need to be updated to take these changes into account, perhaps using a judgment $\Gamma \vdash e : () \mid \Gamma'$, where Γ' is the updated type environment reflecting the types of the variables after evaluating side-effects in e. This approach quickly becomes difficult to manage, especially if it is possible for different variables to "alias", or refer to overlapping parts of the data accessible from Γ, and adding side-effecting functions further complicates matters.

This is *not* the approach to update typechecking that is taken in FLUX. Updates are syntactically distinct from queries, and a FLUX update typechecking judgment such as $\Gamma \vdash^a \{\tau\}\ s\ \{\tau'\}$ assigns an update much richer type information that describes the type of part of the database before and after running s. The values of variables bound in Γ are immutable in the variable's scope, so their types do not need to be updated. Similarly, procedures must be annotated with expected input and output types. We do not believe that these annotations are burdensome in a database setting since a typical update procedure would be expected to preserve the (usually fixed) type of the database.

4.2 Examples

The interesting typing rules are those involving `iter`, tests, and `children`, `left`/`right`, and `insert`/`rename`/`delete`. The following example should help illustrate how the rules work for these constructs. Consider the high-level update:

```
insert after a/b value c[]
```

which can be translated to the following core FLUX statement:

$$\texttt{iter}\ [a?\texttt{children}\ [\texttt{iter}\ [b?\ \texttt{right}\ \texttt{insert}\ c[]]]]$$

Intuitively, this update inserts a c after every b under a top-level a. Now consider the input type $a[b[]^*, c[]], d[]$. Clearly, the output type *should* be $a[(b[], c[])^*, c[]], d[]$. To see how FLUX can assign this type to the update, consider the derivation shown in Figure 5.

As a second example, consider the procedure declaration

$$\begin{array}{l}\texttt{declare procedure}\ \mathit{leafupd}(x{:}\texttt{string}) : \mathit{Tree} \Rightarrow \mathit{Tree}\ \{\\ \quad \texttt{iter}[\texttt{children}[\texttt{iter}[\mathit{leaf}?\texttt{children}[\texttt{delete}; \texttt{insert}\ x];\\ \qquad\qquad \mathit{node}?\texttt{children}[\texttt{iter}[\mathit{leafupd}(x)]]]]]\\ \};\end{array}$$

This procedure updates all leaves of a tree to x. As with the recursive query discussed in Section 3.2, this procedure requires subtyping to typecheck the recursive call. We also

$$\begin{array}{c}
\mathit{leafupd}(\texttt{string}) : \mathit{Tree} \Rightarrow \mathit{Tree} \in \Delta \quad \mathit{tree}[...] <: \mathit{Tree} \quad x{:}\texttt{string} \vdash x : \texttt{string} \\ \hline
x{:}\texttt{string} \vdash^{1} \{\mathit{tree}[\mathit{leaf}[\texttt{string}]|\mathit{node}[\mathit{Tree}^*]]\}\ \mathit{leafupd}(x)\ \{\mathit{Tree}\} \\ \hline
x{:}\texttt{string} \vdash_{\texttt{iter}} \{\mathit{tree}[\mathit{leaf}[\texttt{string}]|\mathit{node}[\mathit{Tree}^*]]\}\ \mathit{leafupd}(x)\ \{\mathit{Tree}\} \\ \hline
x{:}\texttt{string} \vdash_{\texttt{iter}} \{\mathit{Tree}\}\ \mathit{leafupd}(x)\ \{\mathit{Tree}\} \\ \hline
x{:}\texttt{string} \vdash_{\texttt{iter}} \{\mathit{Tree}^*\}\ \mathit{leafupd}(x)\ \{\mathit{Tree}^*\} \\ \hline
x{:}\texttt{string} \vdash^{*} \{\mathit{Tree}^*\}\ \texttt{iter}[\mathit{leafupd}(x)]\ \{\mathit{Tree}^*\} \\ \hline
x{:}\texttt{string} \vdash^{1} \{\mathit{node}[\mathit{Tree}^*]\}\ \texttt{children}[\texttt{iter}[\mathit{leafupd}(x)]]\ \{\mathit{node}[\mathit{Tree}^*]\} \\ \hline
x{:}\texttt{string} \vdash^{1} \{\mathit{node}[\mathit{Tree}^*]\}\ \mathit{node}?\texttt{children}[\texttt{iter}[\mathit{leafupd}(x)]]\ \{\mathit{node}[\mathit{Tree}^*]\}
\end{array}$$

Fig. 6. Partial derivation for body of *leafupd*

need subtyping to check that the return type of the expression matches the declaration. A partial typing derivation for part of the body of the procedure involving a recursive call is shown in Figure 6.

4.3 Algorithmic Completeness and Decidability

To prove update typechecking decidable, we must again carefully control the use of subsumption. The appropriate algorithmic typechecking judgment is defined as follows:

Definition 2 (Algorithmic derivations for updates). *The algorithmic typechecking judgments $\Gamma \mathrel{|\!\!\blacktriangleright}^{a} \{\tau\}\ s\ \{\tau'\}$ and $\Gamma \mathrel{|\!\!\blacktriangleright}_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}$ are obtained by taking the rules in Figure 4, removing the subsumption rule, and replacing the procedure call rule with*

$$\frac{P(\overline{\sigma}) : \sigma \Rightarrow \sigma' \in \Delta \quad \tau <: \sigma \quad \Gamma \mathrel{|\!\!\blacktriangleright} \overline{e} : \overline{\tau} \quad \overline{\tau} <: \overline{\sigma}}{\Gamma \mathrel{|\!\!\blacktriangleright}^{a} \{\tau\}\ P(\overline{e})\ \{\sigma'\}}$$

Moreover, all subderivations of expression judgments in an algorithmic derivation of an update judgment must be algorithmic.

The proof of completeness of algorithmic update typechecking has the same structure as that for queries. Again, proof details are in the technical report [4].

Lemma 4 (Decidabilty for updates). *Let a, s be given. Then there exist computable functions $j_{a,s}$ and k_s such that:*

1. *$j_{a,s}(\Gamma, \tau_1)$ is the unique τ_2 such that $\Gamma \mathrel{|\!\!\blacktriangleright}^{a} \{\tau_1\}\ s\ \{\tau_2\}$, if it exists.*
2. *$k_s(\Gamma, \tau_1)$ is the unique τ_2 such that $\Gamma \mathrel{|\!\!\blacktriangleright}_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$, if it exists.*

Theorem 3 (Algorithmic soundness for updates). *(1) If $\Gamma \mathrel{|\!\!\blacktriangleright}^{*} \{\tau\}\ s\ \{\tau'\}$ is derivable then $\Gamma \vdash^{*} \{\tau\}\ s\ \{\tau'\}$ is derivable. (2) If $\Gamma \mathrel{|\!\!\blacktriangleright}_{\texttt{iter}} \{\tau\}\ e\ \{\tau'\}$ is derivable then $\Gamma \vdash_{\texttt{iter}} \{\tau\}\ e\ \{\tau'\}$ is derivable.*

Lemma 5 (Downward monotonicity for updates). *(1) If $\Gamma \mathrel{|\!\!\blacktriangleright}^{a} \{\tau_1\}\ s\ \{\tau_2\}$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \mathrel{|\!\!\blacktriangleright}^{a} \{\tau_1'\}\ s\ \{\tau_2'\}$ for some $\tau_2' <: \tau_2$. (2) If $\Gamma \mathrel{|\!\!\blacktriangleright}_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \mathrel{|\!\!\blacktriangleright}_{\texttt{iter}} \{\tau_1'\}\ s\ \{\tau_2'\}$ for some $\tau_2' <: \tau_2$.*

Theorem 4 (Algorithmic completeness for updates). *(1) If $\Gamma \vdash^{a} \{\tau_1\}\ s\ \{\tau_2\}$ then there exists $\tau_2' <: \tau_2$ such that $\Gamma \mathrel{|\!\!\blacktriangleright}^{a} \{\tau_1\}\ s\ \{\tau_2'\}$. (2) If $\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$ then there exists $\tau_2' <: \tau_2$ such that $\Gamma \mathrel{|\!\!\blacktriangleright}_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2'\}$.*

5 Related and future work

This work is directly motivated by our interest in using regular expression types for XML updates, using richer typing rules for iteration as found in μXQ [5]. Fernandez, Siméon and Wadler [8] earlier considered an XML query language with more precise typechecking for iteration, but this proposal required additional type annotations; we only require annotations on function or procedure declarations.

For brevity, the core languages in this paper omitted many features of full XQuery, such as the descendant, attribute, parent and sibling axes. The attribute axis is straightforward since attributes always have text content. In μXQ, the descendant axis was supported by assigning $\bar{x}/\texttt{descendant-or-self}$ the type formed by taking the union of all (finitely many) tree types that are reachable from the type of $\bar{x}$. XQuery handles other axes by discarding type information. Our algorithmic completeness proof still appears to work if these axes are added with XQuery- or μXQ-style typing rules.

FLUX's functional, local approach to updates draws on ideas first explored in the CPL+ database update language by Liefke and Davidson [15] (unfortunately this work is not well-known even in the database community). This approach is fundamentally different from the other XML update language proposals of which we are aware (such as XQuery! [11] and the draft W3C XQuery Update Facility [2]). Most such proposals contemplate adding unrestricted side-effecting update operations as additional XQuery expressions, which would undermine many of XQuery's advantages as a purely functional language, such as clear semantics and equational optimization laws. Moreover, to the best of our knowledge, static typechecking and subtyping have not even been considered for these languages and seem likely to encounter difficulties for reasons we outlined in Section 4.1 and discussed in more depth in [3].

On the other hand, XQuery! and related proposals are clearly more expressive than FLUX, and have been incorporated into mature XQuery implementations such as Galax [7]. Although we currently have a prototype that implements the core typechecking algorithm described here as well as the operational semantics described in [3], further work is needed to develop a robust implementation inside an XML database system and evaluate scalability, optimization, and high-level update language design issues.

6 Conclusions

Static typechecking is important in a database setting because type (or "schema") information is useful for optimizing queries and avoiding expensive run-time checks or re-validation. The XQuery standard, like other XML programming languages, employs regular expression types and subtyping. However, its approach to typechecking iteration constructs is imprecise, due to the use of "factoring" which discards information about the order of elements in the result of an iteration operation such as a `for`-loop. While this imprecision may not be harmful for typical queries, it is disastrous for typechecking updates that are supposed to preserve the type of the database.

In this paper we have considered more precise typing disciplines for XQuery-style iterative queries and updates in the core languages μXQ and FLUX respectively. In order to ensure that these type systems are well-behaved and that typechecking is decidable, it

is important to prove the completeness of an algorithmic presentation of typechecking in which the use of subtyping rules is limited so that typechecking remains syntax-directed. We have shown how to do so for the core μXQ and FLUX languages, and believe the proof technique will extend to handle other features not included in the paper. These results provide a solid foundation for subtyping in XML queries and updates.

Acknowledgments Thanks to Peter Buneman and Stijn Vansummeren for many discussions on update languages. The author was supported by EPSRC grant R37476.

References

1. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63, New York, NY, USA, 2003. ACM Press.
2. Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery update facility. W3C Working Draft, July 2006. `http://www.w3c.org/TR/xqupdate/`.
3. James Cheney. LUX: A lightweight, statically typed XML update language. In *ACM SIGPLAN Workshop on Programming Language Technology and XML (PLAN-X 2007)*, pages 25–36, 2007.
4. James Cheney. Regular expression subtyping for XML query and update languages. Technical report, arXiv.org, 2008. `arXiv:0801.0714v1`.
5. Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.
6. Denise Draper, Peter Fankhauser, Mary Fernndez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery-semantics/`.
7. Mary F. Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080, 2003.
8. Mary F. Fernandez, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 263–300, London, UK, 2001. Springer-Verlag.
9. Alain Frisch. OCaml + XDuce. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 192–200, New York, NY, USA, 2006. ACM Press.
10. Vladimir Gapeyev, François Garillot, and Benjamin C. Pierce. Statically typed document transformation: An Xtatic experience. In Giuseppe Castagna and Mukund Raghavachari, editors, *PLAN-X*, pages 2–13. BRICS, 2006.
11. G. Ghelli, K. Rose, and J. Siméon. Commutativity analysis in XML update languages. In Dan Suciu and Thomas Schwentick, editors, *ICDT*, pages 374–388, January 2007.
12. Giorgio Ghelli, Christopher Re, and Jérôme Siméon. XQuery!: An XML query language with side effects. In *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 178–191. Springer, 2006.
13. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technology*, 3(2):117–148, 2003.
14. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
15. Hartmut Liefke and Susan B. Davidson. Specifying updates in biomedical databases. In *SSDBM*, pages 44–53, 1999.
16. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.