

# Typechecking XML Updates

James Cheney

University of Edinburgh

FUN IN THE AFTERNOON IV  
November 22, 2007

## Some words of wisdom

*When any new language design project is nearing completion, there is always a mad rush to get new features added before standardization. The rush is mad indeed, because it leads into a trap from which there is no escape. A feature which is omitted can always be added later, when its design and its implications are well understood. A feature which is included before it is fully understood can never be removed later.*

—C. A. R. Hoare, 1980

# Why study (XML) updates?

- XML query languages and transformation languages are good at:
  - Selecting part of the document (XPath, XQuery)
  - Restructuring documents (XDuce, CDuce, XSLT)
- But bad at:
  - Changing **part** of a document **while leaving the rest unchanged**
- You can **do** it...
- but it's painful.
- And probably not very efficient compared to in-place updates.

# Current proposals

- Current W3C draft **XQuery Update Facility** takes a direct approach: **add imperative, side-effecting update operations**
- It allows aliasing and side-effecting updates to aliased values.
- This can get messy fast, e.g.:

```
for $x in $doc//*  
for $y in $doc//*  
return (do (insert $x into $y;  
           delete $x))
```

- What does this do? What type(s) does it have?

# Current proposals

- Current W3C draft **XQuery Update Facility** takes a direct approach: **add imperative, side-effecting update operations**
- It allows aliasing and side-effecting updates to aliased values.
- This can get messy fast, e.g.:

```
for $x in $doc//*
for $y in $doc//*
return (do (insert $x into $y;
           delete $x))
```

- What does this do? What type(s) does it have?
  - Depends strongly on traversal order...
  - Unlike in XQuery, loop **cannot** necessarily be reordered.

# Wait a second...

- Adding updates to XQuery **naively** seems to negate the benefits of XQuery's **purely-functional** design
  - Clear semantics
  - Unspecified evaluation order
  - Static typechecking
  - Optimizability based on equational laws
- Does the world need another imperative language?
- After all, SQL manages to support “in-place” updates without aliasing, traversal order dependence, etc...
- (Not that SQL is a paragon of language design either).

# Introducing FLUX

- or, Functional Lightweight Udates for XML
- I thought about calling it “Simple Updates for XML” but that doesn’t yield as nice an acronym.

# An example

- Adding an author

```
UPDATE books/book BY
  INSERT AFTER author
  VALUE <author>Charles Dickens</author>
WHERE name = "Through the Looking-Glass"
```

- Type:

```
books[book[author[string], title[string], year[string]]*]
→ books[book[author[string]*, title[string], year[string]]*]
```



# Core language

- Core FLUX consists of the following constructs:

Expressions	$e ::= \dots$
Tests	$\phi ::= n \mid * \mid \text{bool} \mid \text{string}$
Directions	$d ::= \text{left} \mid \text{right} \mid \text{children} \mid \text{iter}$
Statements	$s ::= \text{skip} \mid s; s' \mid \text{if } e \text{ then } s \text{ else } s'$ $\quad \mid \text{let } x = e \text{ in } s \mid \phi?s \mid d[s]$ $\quad \mid \text{insert } e \mid \text{delete} \mid \text{snapshot } x \text{ in } s$

- Expressions* are (core) XQuery expressions (we used  $\mu\text{XQ}$ )
- Tests* allow us to examine the structure of the tree
- Directions* allow us to move somewhere else in the tree
- Statements* perform tests, moves, basic updates, or combinations of updates

# Core language: Expressions

- *Expressions* are (core) XQuery expressions

$$\begin{aligned} e ::= & () \mid e, e' \mid n[e] \mid w \mid x \mid \text{let } x = e \text{ in } e' \\ & \mid \text{true} \mid \text{false} \mid \text{if } c \text{ then } e \text{ else } e' \mid e = e' \\ & \mid x \mid x/\text{child} \mid e :: n \mid \text{for } x \in e \text{ return } e' \end{aligned}$$

- We treat queries as a “black box”, reusing the  $\mu\text{XQ}$  core language of Colazzo, Ghelli, Manghi and Sartiani (ICFP 2004)

# Core language: Tests

- *Tests* allow us to examine the structure of the tree

$$\phi ::= n \mid * \mid \text{bool} \mid \text{string}$$

- $n$  succeeds when we are at a tree labeled with  $n$
- $*$  succeeds when we are at *any* tree node
- `bool`, `string` succeed when we are at a (boolean, string) data node
- Corresponding statement  $\phi?s$  means: “If test  $\phi$  succeeds, do  $s$ , otherwise do nothing.”

# Core language: Directions

- *Directions* allow us to move somewhere else in the tree

$$d ::= \text{left} \mid \text{right} \mid \text{children} \mid \text{iter}$$

- `left`, `right` move to the beginning or end of the current sequence.
- `children` moves to the child sequence of the current tree node
- `iter` moves to *each element* of the current sequence, in parallel.
- $d[u]$  means: “move according to  $d$ , then do  $u$ ”

# Core language: Statements

- *Statements* perform tests, moves, basic updates, or combinations of updates

$$\begin{aligned} s \quad ::= & \text{ skip } | s; s' | \text{ if } e \text{ then } s \text{ else } s' \\ & | \text{ let } x = e \text{ in } s | \phi?s | d[s] \\ & | \text{ insert } e | \text{ delete } | \text{ snapshot } x \text{ in } s \end{aligned}$$

- skip, sequence, if: standard

# Core language: Statements

- *Statements* perform tests, moves, basic updates, or combinations of updates

$$s ::= \text{skip} \mid s; s' \mid \text{if } e \text{ then } s \text{ else } s' \\ \mid \text{let } x = e \text{ in } s \mid \phi?s \mid d[s] \\ \mid \text{insert } e \mid \text{delete} \mid \text{snapshot } x \text{ in } s$$

- `let`: binds a variable  $x$  to the result of an XQuery expression  $e$
- `let`-bound variable values are *immutable*; semantically, this makes a copy of  $e$
- This is important for avoiding aliasing.

# Core language: Statements

- *Statements* perform tests, moves, basic updates, or combinations of updates

$$\begin{aligned}
 s \quad ::= & \quad \text{skip} \mid s; s' \mid \text{if } e \text{ then } s \text{ else } s' \\
 & \quad \mid \text{let } x = e \text{ in } s \mid \phi?s \mid d[s] \\
 & \quad \mid \text{insert } e \mid \text{delete} \mid \text{snapshot } x \text{ in } s
 \end{aligned}$$

- insert  $e$ : inserts expression  $e$  at current position
- delete: deletes current selection
- test  $\phi?s$ , move  $d[s]$  already discussed

# Core language: Statements

- *Statements* perform tests, moves, basic updates, or combinations of updates

$$s ::= \text{skip} \mid s; s' \mid \text{if } e \text{ then } s \text{ else } s' \\ \mid \text{let } x = e \text{ in } s \mid \phi?s \mid d[s] \\ \mid \text{insert } e \mid \text{delete} \mid \text{snapshot } x \text{ in } s$$

- snapshot: binds  $x$  to *current value* of the context.
- Note: Like let-bound variables, snapshot variables are *immutable*.
- snapshot is the *only way* to read from the mutable store



# Semantics

- Values are trees/forests:

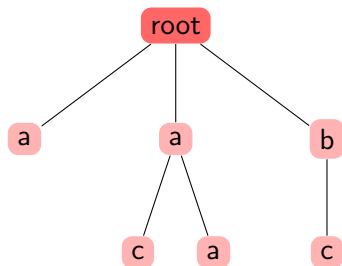
$$t ::= \text{string} \mid \text{bool} \mid a[f]$$
$$f ::= () \mid t, f$$

- Semantics of values, updates is purely *value-based*
- **We do not even mention “node ids”**
- Semantics straightforward; rather than bore you with details, here's a graphical example

# Example

## Example update

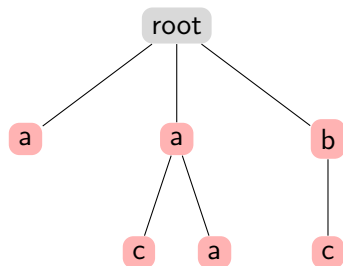
```
children[iter[a?children[left [insert b[]]]]]
```



# Example

## Example update

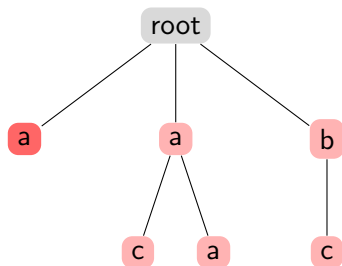
```
children[iter[a?children[left [insert b[]]]]]
```



# Example

## Example update

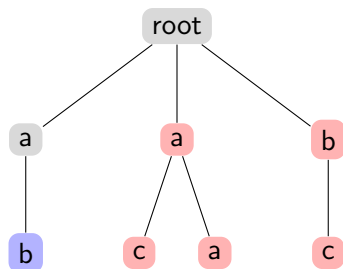
```
children[iter[a?children[left [insert b[]]]]]
```



# Example

## Example update

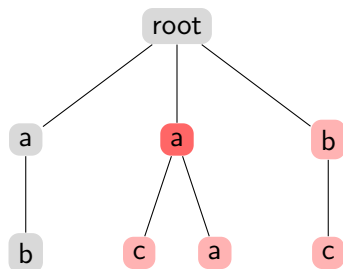
```
children[iter[a?children[left [insert b[]]]]]
```



# Example

## Example update

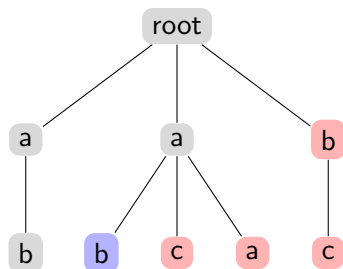
```
children[iter[a?children[left [insert b[]]]]]
```



# Example

## Example update

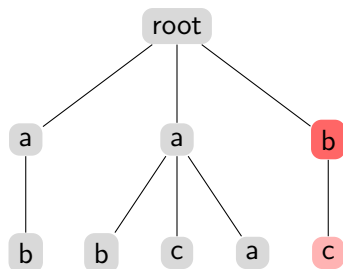
```
children[iter[a?children[left [insert b[]]]]]
```



# Example

## Example update

```
children[iter[a?children[left [insert b[]]]]]
```

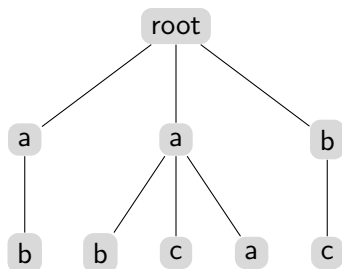




# Example

## Example update

```
children[iter[a?children[left [insert b[]]]]]
```



# Static typing

- FLUX core operations can all be **statically typed**
- We use **XDuce-style regular expression types**

$$\tau ::= \text{string} \mid \text{bool} \mid a[\tau] \mid \tau, \tau' \mid \tau|\tau' \mid \tau^* \mid \epsilon$$

with **inclusion-based subtyping**

- Key idea: To typecheck an `iter[u]` operation at  $\tau$ , typecheck  $u$  at each *singleton tree component* of  $\tau$ , and combine the results
- Examples:

$$\text{iter}[a?\text{left}[\text{insert } b]] : (a[c])^* \mid (a, b)^* \Rightarrow (b, a[c])^* \mid (b, a, b)^*$$

$$\text{iter}[a?\text{children}[\text{delete}]] : (a[c])^* \mid (a, b)^* \Rightarrow (a[])^* \mid (a, b)^*$$

# Static typing

- Judgment  $\Gamma \vdash^1 \{\tau\} u \{\tau'\}$ :  $u$  updates singular input  $\tau$  to output  $\tau'$  given variables typed by  $\Gamma$
- Judgment  $\Gamma \vdash^* \{\tau\} u \{\tau'\}$  similar, but expects a *sequence* of type  $\tau$

## Children

$$\frac{\Gamma \vdash^* \{\tau\} s \{\tau'\}}{\Gamma \vdash^1 \{n[\tau]\} \text{children}[s] \{n[\tau']\}}$$

## Snapshot

$$\frac{\Gamma, x:\tau \vdash^a \{\tau\} s \{\tau'\}}{\Gamma \vdash^a \{\tau\} \text{snapshot } x \text{ in } s \{\tau'\}} \quad a \in \{1, *\}$$

# Static typing

- $\Gamma \vdash_{\text{iter}} \{\tau\} s \{\tau'\}$  an auxiliary judgment, meaning “iterating  $s$  changes  $\tau$  to  $\tau'$ ”
- Here,  $\alpha$  is “atomic” ( $n[\tau]$ , `bool`, `string`)

## Iteration

$$\frac{\Gamma \vdash_{\text{iter}} \{\tau\} s \{\tau'\}}{\Gamma \vdash^* \{\tau\} \text{iter}[s] \{\tau'\}} \qquad \frac{\Gamma \vdash^1 \{\alpha\} s \{\tau\}}{\Gamma \vdash_{\text{iter}} \{\alpha\} s \{\tau\}}$$

$$\frac{}{\Gamma \vdash_{\text{iter}} \{()\} s \{()\}} \qquad \frac{\Gamma \vdash_{\text{iter}} \{\tau_1\} s \{\tau'_1\} \quad \Gamma \vdash_{\text{iter}} \{\tau_2\} s \{\tau'_2\}}{\Gamma \vdash_{\text{iter}} \{\tau_1, \tau_2\} s \{\tau'_1, \tau'_2\}}$$

$$\vdots$$

# Static typing

- For test typechecking,  $\alpha <: \phi$  means that some value of type  $\alpha$  matches test  $\phi$

## Test

$$\frac{\alpha <: \phi \quad \Gamma \vdash^1 \{\alpha\} s \{\tau\}}{\Gamma \vdash^1 \{\alpha\} \phi?s \{\tau\}} \quad \frac{\alpha \not<: \phi}{\Gamma \vdash^1 \{\alpha\} \phi?s \{\alpha\}}$$

- Atomic updates

## Insert/delete

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash^* \{()\} \text{insert } e \{\tau\}} \quad \frac{}{\Gamma \vdash^* \{\tau\} \text{delete } \{()\}}$$

# Static typing

- Subtyping is by regular (tree) language inclusion.
- Can re-use Hosoya, Vouillon and Pierce's subtyping algorithm.
- Type soundness holds (proof not hard).
- Currently, consider type checking with respect to *fixed input type*
- This is reasonable for DB applications because schema usually given in advance
- Type inference/principal typing would be nice to have though.

# Deciding typechecking

- In the presence of **subtyping/subsumption**, typechecking is no longer **syntax-directed**, and an expression may have many types.
- Usual solution: Define an **algorithmic** system that is syntax directed and restricts the use of subsumption
- Show that arbitrary derivations can be **normalized** to algorithmic ones by “permuting subsumption downwards”
- For FLUX, this mostly works.
- But straightforward induction **fails** for  $\Gamma \vdash_{\text{iter}} \{\tau\} \leq \{\tau'\}$  judgment
- Requires a trickier “semantic” argument (considering structure of regular expression types)

## Related Work

- Liefke, Davidson [SSDBM 1999] — introduced an update language for complex object databases that heavily influenced FLUX
- Collazzo, Ghelli, Manghi, Sartiani [ICFP 2004] — defined  $\mu$ XQ and type system which we have re-used
- Many “imperative XQuery” approaches to updates (XQuery!, XQueryP, XQueryU, XQuery Update Facility).
- Zarfaty/Gardner/Calcagno - Logics for reasoning about low-level (DOM-like) operations on XML-like trees.
- My update type system is reminiscent of “arrows” (and also Hoare Type Theory); maybe some formal relationship



# Conclusions

- FLUX is work in progress
- Presented a **high-level** update language
- Discussed **core language** with **sound type system** and **decidable typechecking**
- Hope I've convinced you that we can have **updates** without the full complications of **imperative programming**.