Adventures in XML Updates

James Cheney
University of Edinburgh

Joint work with Michael Benedikt, Oxford
Problem

• Most databases change over time
• XQuery doesn’t handle this well
  • Can write “query” that copies data & makes small change
  • but this can be awkward or inefficient
  • and some “updates” only expressible with user defined functions
Update languages

- SQL has update expressions distinct from queries
- XML updates can’t be expressed easily/efficiently using XQuery
- W3C developing XQuery Update Facility
  - Goal: SQL-like updates for XML??
Problem

- XML/trees more complicated than tables
- Larger language design space
- Typechecking, static analysis ill-understood
Goal

• Want to **predict** effect of update on database (schema)
  • This isn’t addressed at all by standard (or any previous work)

• Problem statement:
  • Given **input schema** and **update**, calculate **output schema** that describes data after doing update

• Checking undecidable (and exact inference impossible) for sufficiently rich language
  • example: linear trees -> \{ a^n b^n \mid n \geq 0 \}
Functional Updates for XML
FLUX

- [C. ICFP '08]
- Goal: "functional" update language
  - clear semantics
  - straightforward typechecking
- Based on a "functionally" flavored database update language (Liefke and Davidson 1999, Buneman, C. & Vansummeren 2008)
A high-level update

insert as last into $doc/a
value <c/>;
delete $doc/a/b
A low-level update

```python
children[iter[a?
    children[left[insert <c/>]]
]];
children[iter[a?
    children[iter[b? delete]]
]]
```
A low-level update

```javascript
children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
```
A low-level update

```javascript
children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
```
A low-level update

children[iter[a?
    children[left[insert <c/>]]
  ]];
children[iter[a?
    children[iter[b? delete]]
  ]]

children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
An optimized low-level update

```
children[iter[a?
    children[left[insert <c/>];
      iter[b? delete]]]
]}
```
An optimized low-level update

```javascript
  children[iter[a?
    children[left[insert <c/>];
    iter[b? delete]]
  ]]
```
An optimized low-level update

```javascript
children[iter[a? children[left][insert <c/>]; iter[b? delete]]
]}
```
Core FLUX

- **Updates:**
  
  $$s ::= \text{skip} | s; s' | \text{if } e \text{ then } s \text{ else } s'$$
  
  $$| \quad \text{let } x = e \text{ in } s$$
  
  $$| \quad \text{insert } e | \text{delete } | \text{rename } n$$
  
  $$| \quad \text{snapshot } x \text{ in } s | \phi? s | d[s] | P(\bar{e})$$

  $$\phi ::= n | \text{node()} | \text{text()}}$$

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

- **Queries** $e$ a sublanguage

- **Recursive update procedures, queries**
Types

• XDuces-style regular expression types (Hosoya et al. 2003, 2005)

\[
\begin{align*}
\alpha & ::= \text{bool} \mid \text{string} \mid n[\tau] \\
\tau, \sigma & ::= \alpha \mid () \mid \tau\tau' \mid \tau, \tau' \mid \tau^* \mid X
\end{align*}
\]

• Main typing judgment: \( \Gamma \vdash \{\tau\} s \{\tau'\} \)
Atomic updates

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Gamma \vdash \{(())\} \text{ insert } e \{\tau\} \\
\Gamma \vdash \{\tau\} \text{ delete } \{()\} \\
\Gamma \vdash \{m[\tau]\} \text{ rename } n \{n[\tau]\}
\end{align*}
\]
Iteration

\[
\begin{align*}
\Gamma & \vdash_{\text{iter}} \{ \tau \} s \{ \tau' \} \\
\Gamma & \vdash \{ \tau \} \text{ iter}[s] \{ \tau' \} \\
\Gamma & \vdash_{\text{iter}} \{ () \} s \{ () \} \\
\Gamma & \vdash \{ \alpha \} s \{ \tau \} \\
\Gamma & \vdash_{\text{iter}} \{ \alpha \} s \{ \tau \} \\
\Gamma & \vdash_{\text{iter}} \{ \tau_1 \} s \{ \tau'_1 \} \quad \Gamma & \vdash_{\text{iter}} \{ \tau_2 \} s \{ \tau'_2 \} \\
\Gamma & \vdash_{\text{iter}} \{ \tau_1, \tau_2 \} s \{ \tau'_1, \tau'_2 \} \\
& \vdots
\end{align*}
\]
High-level language

• Core updates: easy to typecheck, painful to write

• Alternative syntax:

\[
\text{Upd ::= } \text{INSERT (BEFORE|AFTER) Path VALUE Expr} \\
\text{INSERT AS (FIRST|LAST) INTO Path VALUE Expr} \\
\text{DELETE [FROM] Path} \\
\text{RENAME Path TO Lab} \\
\text{REPLACE [IN] Path WITH Expr} \\
\text{UPDATE Path BY Stmt}
\]

\[
\text{Path ::= . | Lab | node() | text()} \\
\text{Path/Path | Var AS Path | Path[Expr]}
\]
Some results

- Soundness: type system correctly predicts schema after update
- High-level language & type system with sound and complete translation to core
  - translation typechecks iff source typechecks
- "Dead code" analysis
  - warning if a sub-update is statically == "skip"
Aftermath

• Hasn't influenced W3C, XML DB comm.
  • FLUX is less expressive
  • But maybe more optimizable?
  • More work could be done on this

• XQuery is already big & complicated
  • "Why should updates be any simpler?"
Typechecking for W3C's update language
Example

- W3C proposal has counterintuitive (?) semantics

  delete $x//a,
  insert <foo>bar</foo>
  before $x//a

- does not do what you (probably) expect
Example

delete $x//a,
insert <foo>bar</foo>
before $x//a
First **collect** updates

delete $x//a,
insert `<foo>bar</foo>`
before $x//a
First **collect** updates

delete $x//a,
insert `<foo>bar</foo>`
before $x//a
Then reorder & apply

delete $x//a,
insert <foo>bar</foo>
before $x//a
Then reorder & apply

delete $x//a,
insert <foo>bar</foo>
before $x//a
Then **reorder** & **apply**

```plaintext
delete $x//a,
insert <foo>bar</foo>
before $x//a
```
A trivial sound solution

- Ignore the update and input schema and produce output schema that says that output can have *any* structure.
  - It’s sound...
  - But not very exciting.
- Can we do better?
Overview of our approach

• Step 0: Calculate result types for queries
• Step 1: Calculate \textit{effects} of updates
• Step 2: \textbf{Apply} effects to input schema, “altering” it to output schema
Overview of our approach

- Step 0: Calculate result types for queries
- Step 1: Calculate **effects** of updates
- Step 2: **Apply** effects to input schema, “altering” it to output schema
- We’ll focus on step 2
We consider “flat” schemas

(close to tree automata)

Flat types are of the form

\[ \tau ::= () | T | \tau_1, \tau_2 | \tau_1|\tau_2 | \tau^* \]

Flat rules are of the form

\[ S \rightarrow a[\tau] \]

Schemas are sets of rules + “root” type
Effects

- Characterize behavior of updates
- Syntax:

$$\Omega ::= \emptyset \mid \Omega \cup \Omega' \mid \text{insert}(\tau, d, T) \mid \text{delete}(T) \mid \cdots$$

$$d ::= \text{into} \mid \text{into}_\text{as}_\text{first} \mid \text{into}_\text{as}_\text{last} \mid \text{before} \mid \text{after}$$

- Statically approximate run-time pending update list
- (largely the same as query typechecking)
Effects

• Characterize behavior of updates

• Syntax:

\[ \Omega ::= \emptyset | \Omega \cup \Omega' | \text{insert}(\tau, d, T) | \text{delete}(T) | \cdots \]

\[ d ::= \text{into} | \text{into}_\text{as}_\text{first} | \text{into}_\text{as}_\text{last} | \text{before} | \text{after} \]

• Statically approximate run-time pending update list
Effects

• Characterize behavior of updates

• Syntax

\[ \Omega ::= \emptyset | \Omega \cup \Omega' | \text{insert}(\tau, d, T) | \text{delete}(T) | \cdots \]

\[ d ::= \text{into} | \text{into}_\text{as}_\text{first} | \text{into}_\text{as}_\text{last} | \text{before} | \text{after} \]

• Statically approximate run-time pending update list
Effect inference

- We calculate a (conservative) upper bound on effect of update on given schema

\[
\begin{align*}
S & \rightarrow d[T,U] \\
T & \rightarrow c[A^*] \\
U & \rightarrow a[(B,A)^*]
\end{align*}
\]

- Inferred effect:

\[
\{ \text{delete}(A), \text{delete}(U), \\
\text{insert}(\text{Foo}, \text{before}, A), \\
\text{insert}(\text{Foo}, \text{before}, U) \}
\]

where Foo $\rightarrow$ foo[string]
Schema alteration

- Given input schema

\[
\begin{align*}
S & \rightarrow d[T,U] \\
T & \rightarrow c[A^+] \\
U & \rightarrow a[(B,A)^+] \\
A & \rightarrow a[] \\
B & \rightarrow b[]
\end{align*}
\]
Given **input schema** and **effect**
Schema alteration

- Given **input schema** and **effect**
- Want to calculate **output schema**

| S --> d[T, U] |
| T --> c[A*] |
| U --> a[(B, A)*)] |
| A --> a[] |
| B --> b[] |

{delete(A), delete(U), insert(Foo, before, A), insert(Foo, before, U)}

| Foo --> foo[string] |
| S' --> d[T’, (Foo*, U’)] |
| T’ --> c[(Foo*, A’)*] |
| U’ --> a[(B’, (Foo*, A’))*] |
| A’ --> a[]? |
| B’ --> b[] |
Stage 0: Copy the schema

- Make “fresh” copy of old schema types

S -> d[T,U]
T -> c[A*]
U -> a[(B,A)*]
A -> a[
B -> b[

{delete(A), delete(U),
insert(Foo, before, A),
insert(Foo, before, U)}

Foo -> foo[string]
S’ -> d[T’,U’]
T’ -> c[A’*]
U’ -> a[(B’,A’)*]
A’ -> a[
B’ -> b[}
Stage 0: Copy the schema

- Make “fresh” copy of old schema types

S -> d[T,U]
T -> c[A*]
U -> a[(B,A)*]
A -> a[
B -> b[

Foo -> foo[string]

S' -> d[T',U']
T' -> c[A'*]
U' -> a[(B',A')*]
A' -> a[
B' -> b[

Also any new types needed for data created by update:

{delete(A), delete(U),
 insert(Foo, before, A),
 insert(Foo, before, U)}
Stage 1: Inserts

• Inserts happen first:

\[
\begin{align*}
S & \rightarrow d[T,U] \\
T & \rightarrow c[A^*] \\
U & \rightarrow a[(B,A)^*] \\
A & \rightarrow a[] \\
B & \rightarrow b[] \\
\end{align*}
\]

\{delete(A), delete(U), insert(Foo, before, A), insert(Foo, before, U)\}

\[
\begin{align*}
\text{Foo} & \rightarrow \text{foo[string]} \\
S' & \rightarrow d[T', (\text{Foo}^*, U')] \\
T' & \rightarrow c[(\text{Foo}^*, A')^*] \\
U' & \rightarrow a[(B', (\text{Foo}^*, A'))^*] \\
A' & \rightarrow a[] \\
B' & \rightarrow b[] \\
\end{align*}
\]
Stage 1: Inserts

- Inserts happen first:

$S \rightarrow d[T,U]$
$T \rightarrow c[A^*]$
$U \rightarrow a[(B,A)^*]$
$A \rightarrow a[]$
$B \rightarrow b[]$

$\{\text{delete}(A), \text{delete}(U), \text{insert}(\text{Foo, before, A}), \text{insert}(\text{Foo, before, U})\}$

Foo $\rightarrow$ foo[string]
$S' \rightarrow d[T',(\text{Foo}^*,U')]$
$T' \rightarrow c[(\text{Foo}^*,A')]$
$U' \rightarrow a[(B',(\text{Foo}^*,A'))^*]$
$A' \rightarrow a[]$
$B' \rightarrow b[]$

Effects don’t say how many times insert might happen
Stage 2,3: replace, rename

- Replace and rename operations happen after inserts but before deletes.
- There aren’t any replace/rename ops in this example.
- So we’ll skip this step.
Stage 4: Deletes

• Deletes happen last:

\[
S \rightarrow d[T, U] \\
T \rightarrow c[A^*] \\
U \rightarrow a[(B, A)^*] \\
A \rightarrow a[] \\
B \rightarrow b[] \\
\]

{\text{delete}(A), \text{delete}(U), \text{insert}(\text{Foo}, \text{before}, A), \text{insert}(\text{Foo}, \text{before}, U)}

\[
\text{Foo} \rightarrow \text{foo[string]} \\
S' \rightarrow d[T', (\text{Foo}', U')] \\
T' \rightarrow c[(\text{Foo}', A')^*] \\
U' \rightarrow a[(B', (\text{Foo}', A'))^*]?
\]

A' \rightarrow a[]?

B' \rightarrow b[]
Cleanup

• Get rid of unneeded old types

Foo -> foo[string]
S’ -> d[T’,(Foo*,U’)]
T’ -> c[(Foo*,A’)*]
U’ -> a[(B’,(Foo*,A’))*]?
A’ -> a[]?
B’ -> b[]
Correctness

• Judge correctness w.r.t semantics of updates
  • **Problem**: W3C proposal lacks *formal semantics*
  • So we defined a semantics too
  • Uses standard ideas from operational semantics
    • Lots of cases, need to model “store” and memory allocation
    • See paper for details
Related work

- Typechecking XML queries
  - Colazzo et al. [JFP 2006], Colazzo & Sartiani [ICTCS 2010]
- XML query-update independence problem
  - Benedikt & C. [VLDB 09-10], others
- XML update analysis/optimization
  - Ghelli et al. [TODS 2008, SIGMOD 2008]
Future work

• Improving precision of typechecking, independence analysis

• Other type-based optimizations?

• Formalizing typechecking & other algorithms (Nominal Isabelle?)

• Checking validity of update optimizations

• Combining typechecking with more precise static analysis of paths
Conclusions

- Presented two different approaches to typechecking XML updates
- FLUX: simple semantics/typechecking, but not expressive enough for some applications
- W3C proposal for XML updates is complicated
  - Semantics ill-understood, and probably deserves further study