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 ineffcent
  - 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

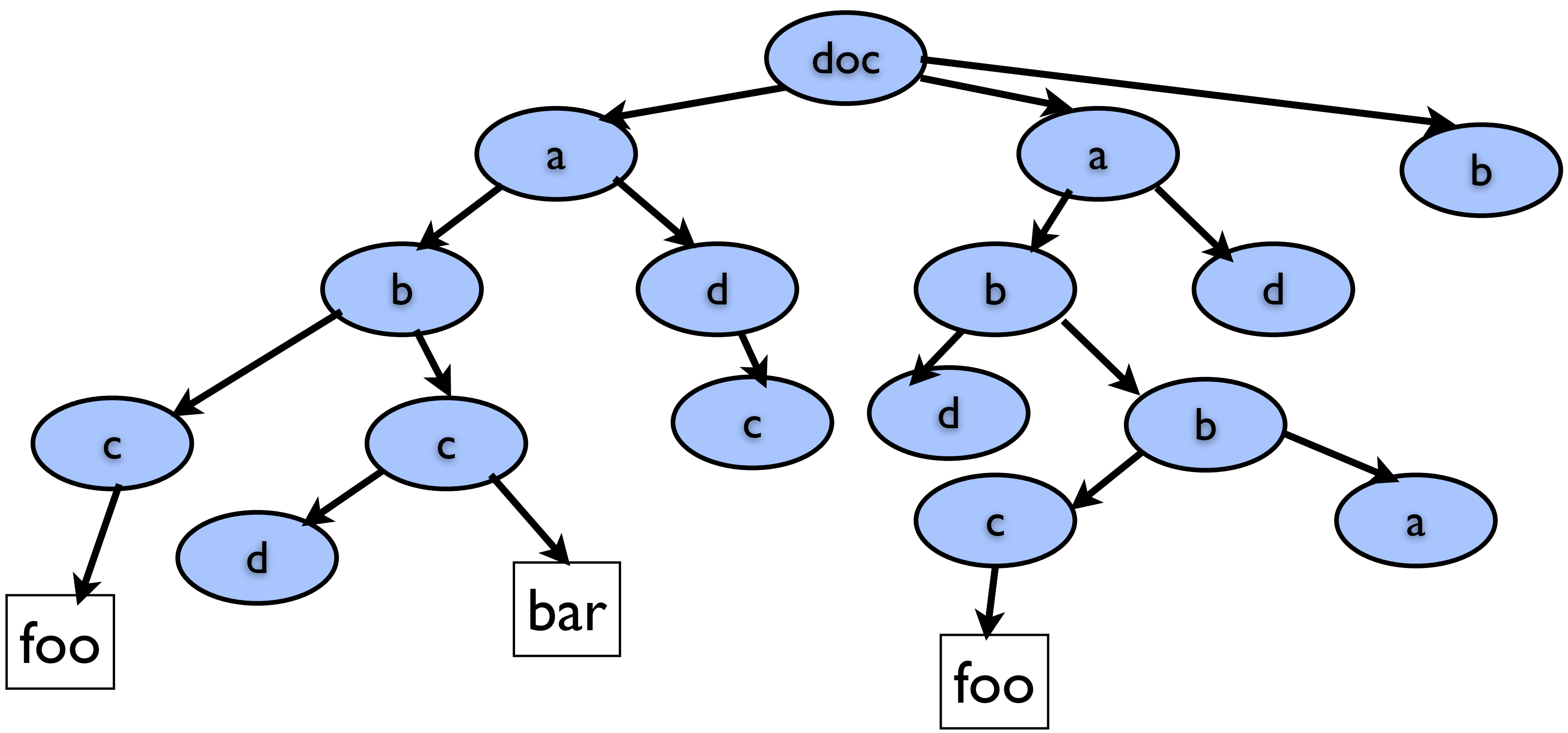
- Want to **predict** effect of update on database (schema)
    - This isn't addressed at all by standard (or any previous work)
- Problem statement:
    - Given **input schema** and **update**, calculate **output schema** that describes data after doing update
- Checking undecidable (and exact inference impossible) for sufficiently rich language
    - example: linear trees -> $\{ a^n b^n \mid n \geq 0 \}$

Functional Updates for XML

FLUX

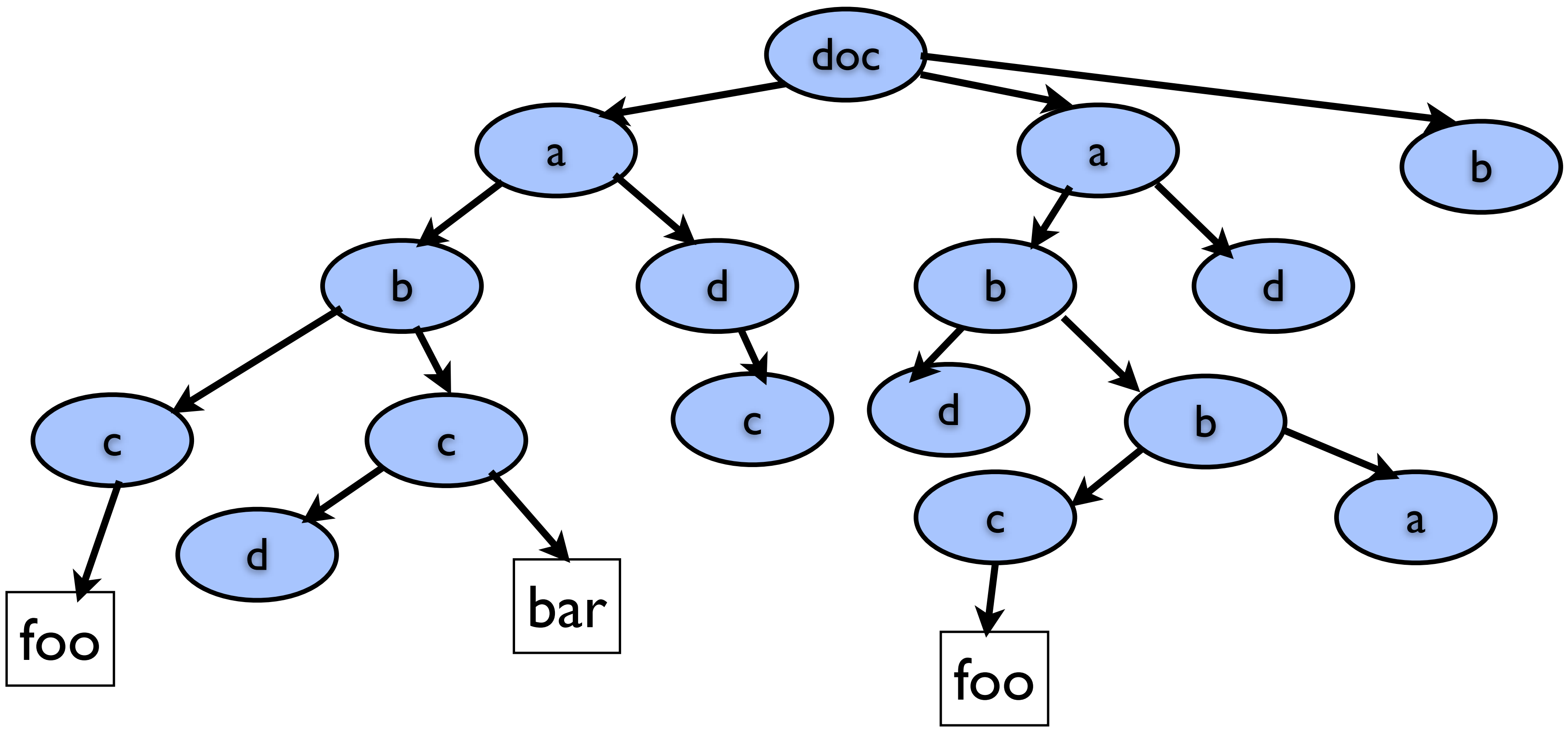
- [C. ICFP '08]
- Goal: "functional" update language
  - clear semantics
  - straightforward typechecking
- Based on a "functionally" flavored database update language (Liefke and Davidson 1999, Buneman, C. & Vansummeren 2008)

A high-level update



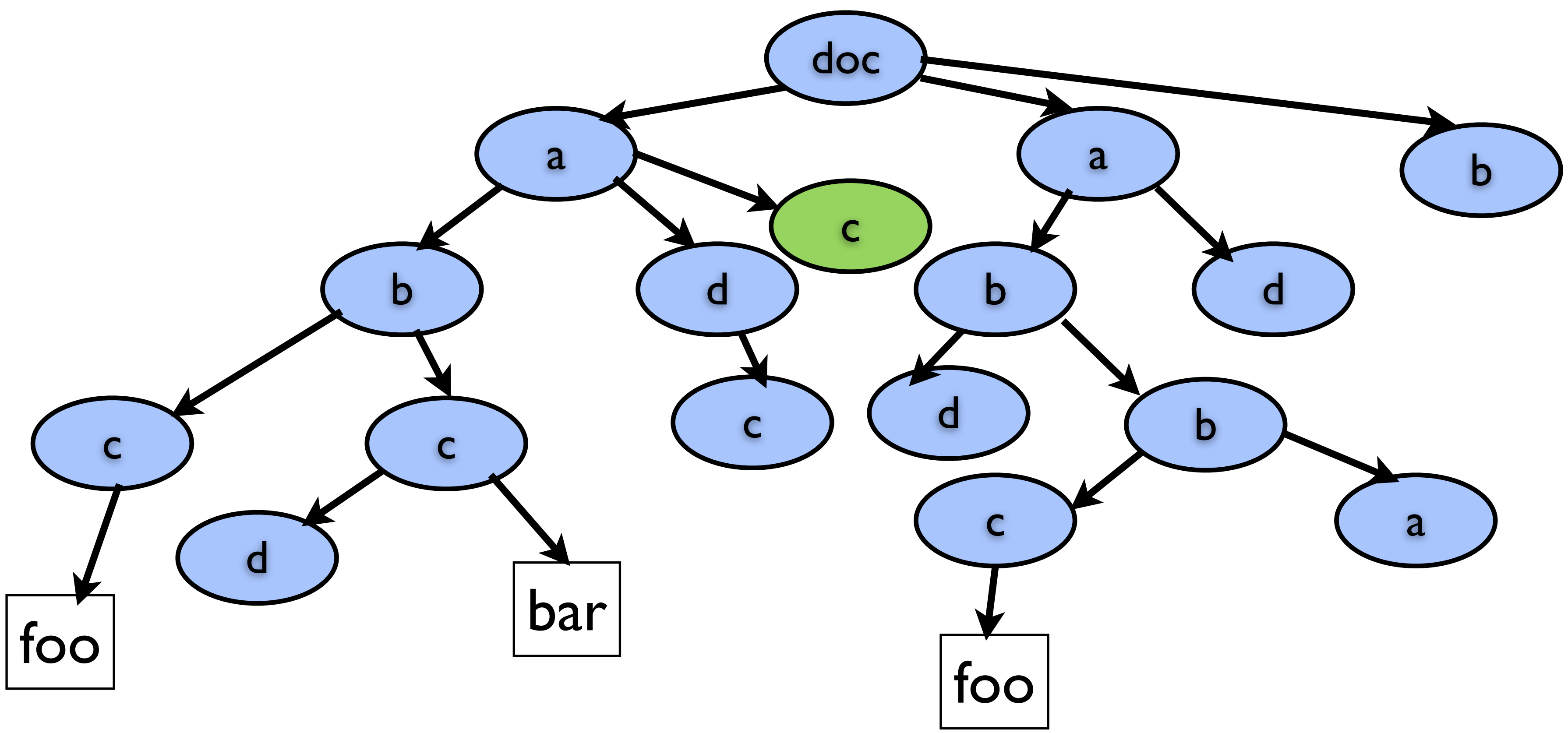

```
insert as last
into $doc/a
value <c/>;
delete $doc/a/b
```

A low-level update



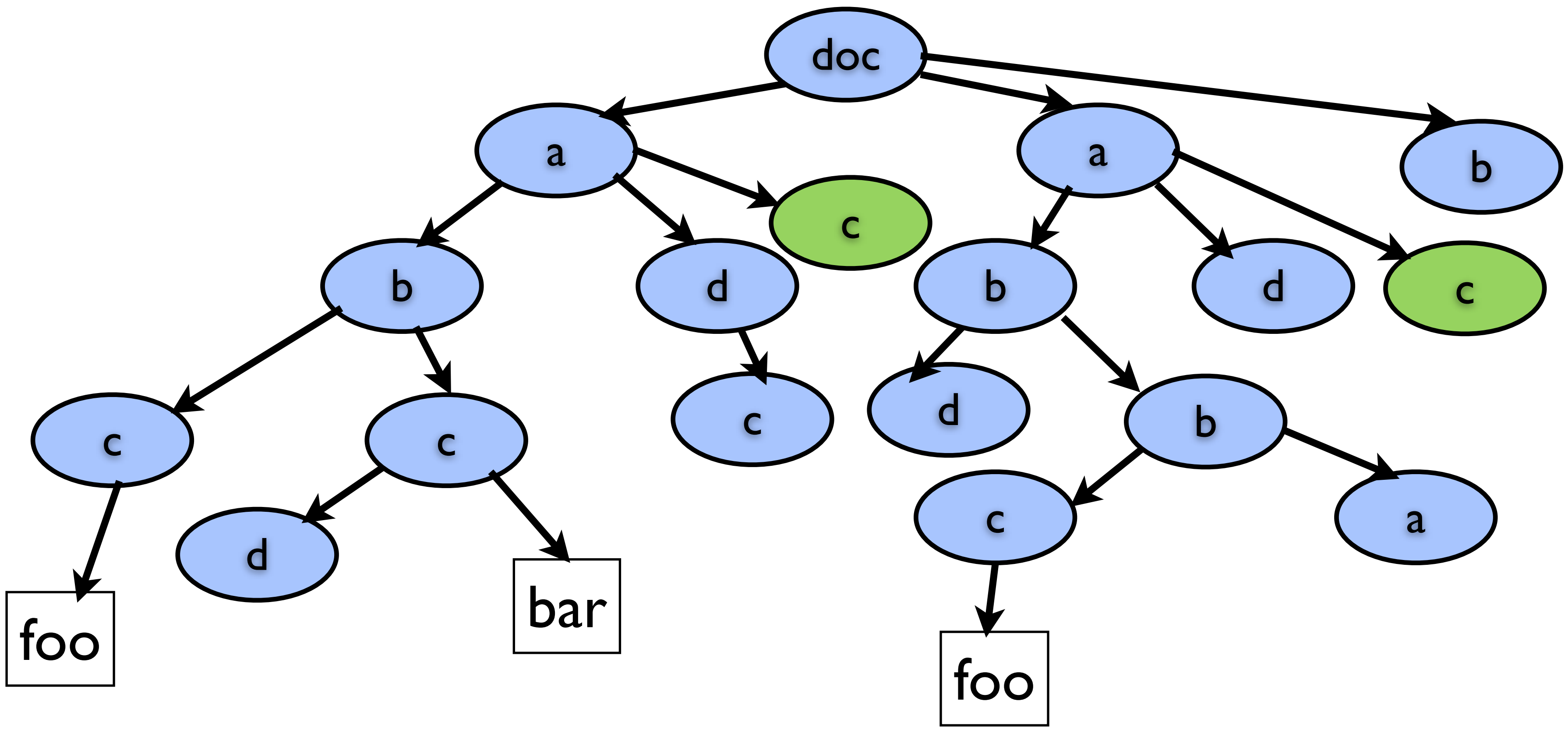

```
children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
```

A low-level update



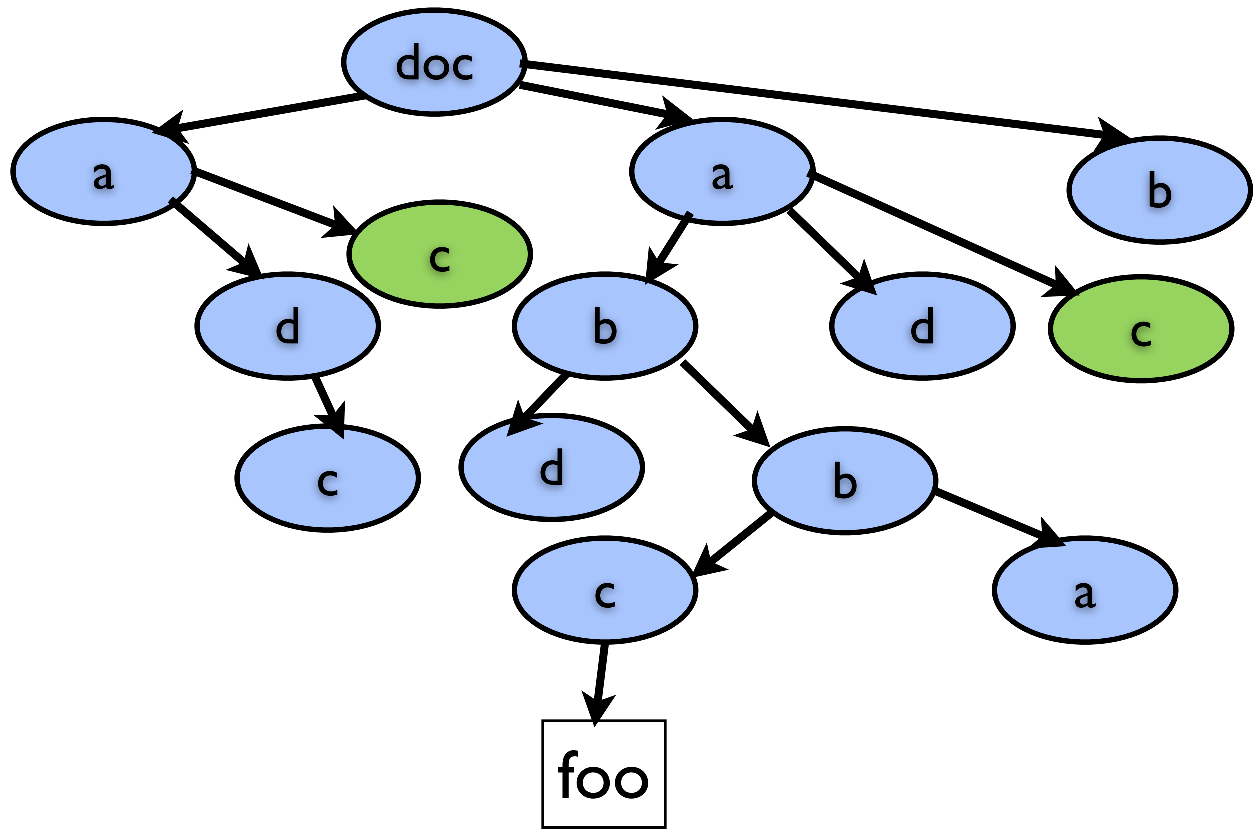

```
children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
```

A low-level update



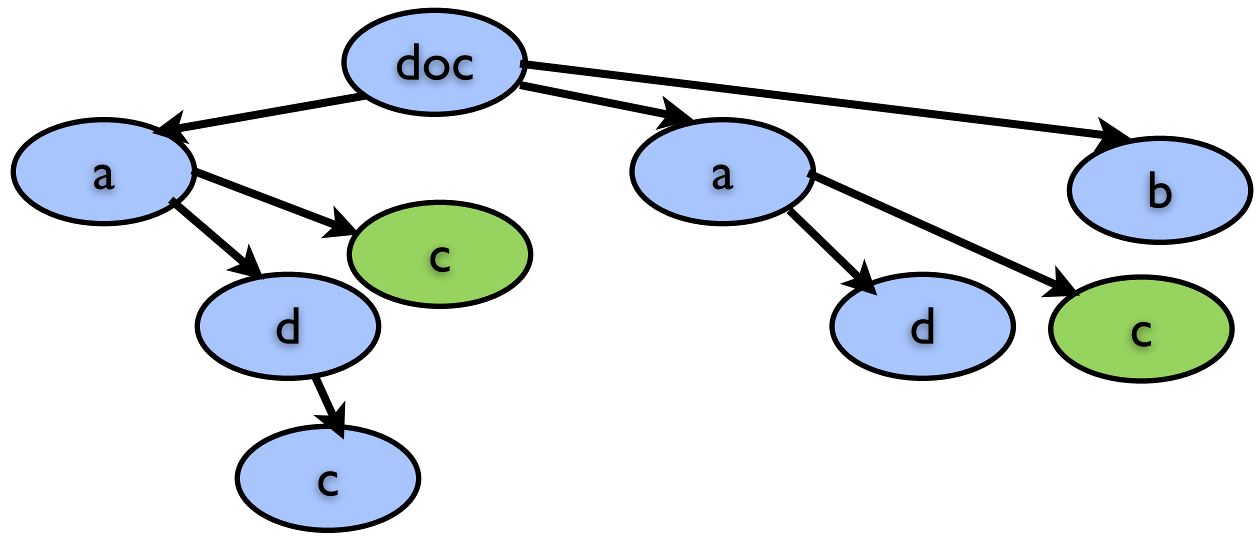

```
children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
```

A low-level update



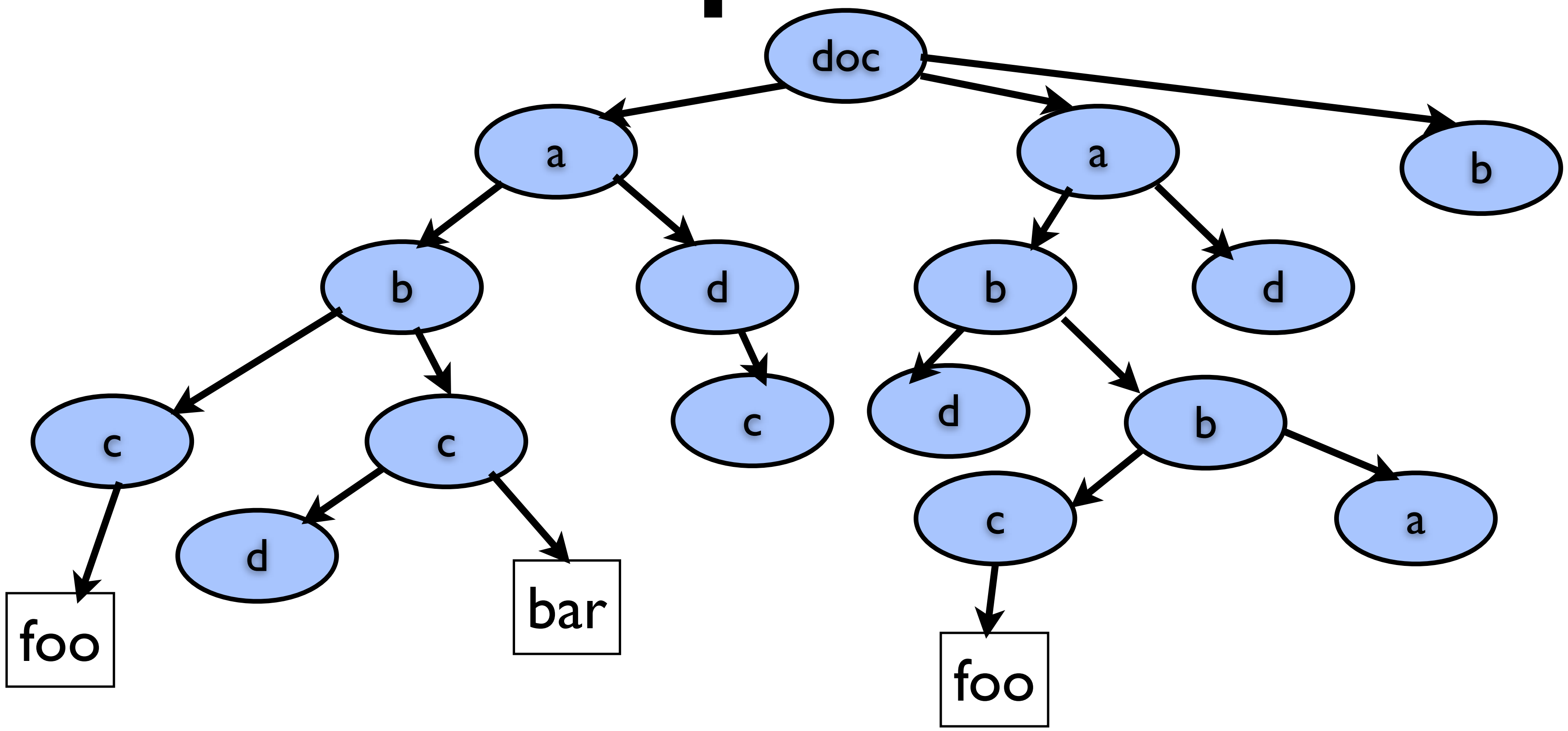

```
children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
```

A low-level update



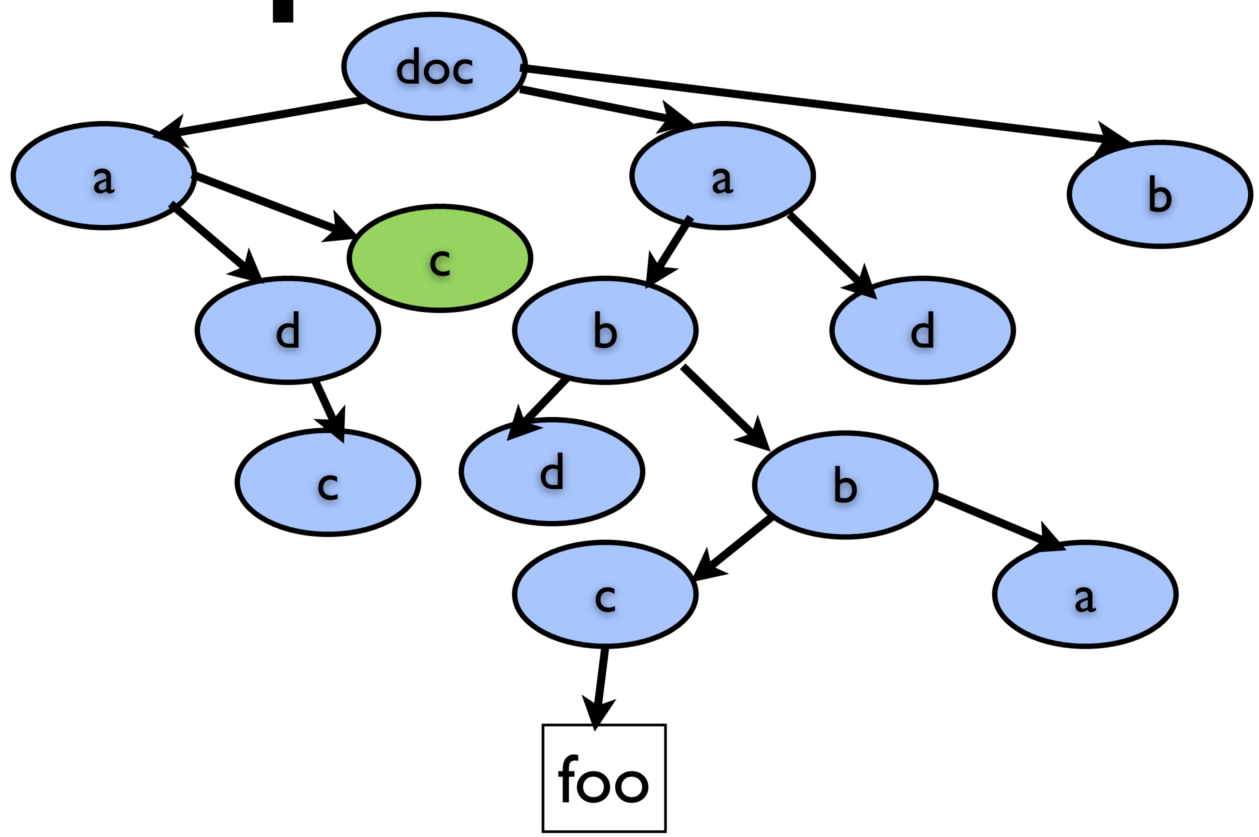

```
children[iter[a?
  children[left[insert <c/>]]
]];
children[iter[a?
  children[iter[b? delete]]
]]
```

An optimized low-level update



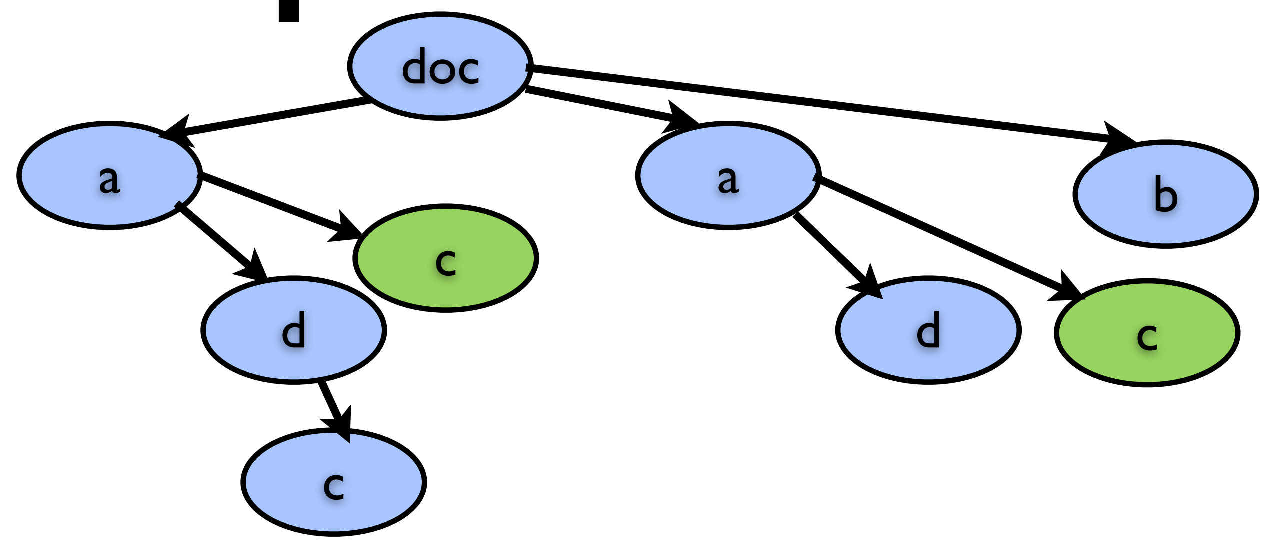

```
children[iter[a?
  children[left[insert <c/>];
           iter[b? delete]]
]]
```

An optimized low-level update




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

An optimized low-level update




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

Core FLUX

- Updates:

$$
\begin{array}{rcl}
s & ::= & \texttt{skip} \mid s; s' \mid \texttt{if } e \texttt{ then } s \texttt{ else } s' \\
& \mid & \texttt{let } x = e \texttt{ in } s \\
& \mid & \texttt{insert } e \mid \texttt{delete} \mid \texttt{rename } n \\
& \mid & \texttt{snapshot } x \texttt{ in } s \mid \phi?s \mid d[s] \mid P(\vec{e}) \\
\phi & ::= & n \mid \texttt{node()} \mid \texttt{text()} \\
d & ::= & \texttt{left} \mid \texttt{right} \mid \texttt{children} \mid \texttt{iter}
\end{array}
$$

- Queries e a sublanguage
- Recursive update procedures, queries

Types

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

$$\begin{array}{rcl} \alpha & ::= & \texttt{bool} \mid \texttt{string} \mid n[\tau] \\ \tau, \sigma & ::= & \alpha \mid \texttt{()} \mid \tau|\tau' \mid \tau, \tau' \mid \tau^* \mid X \end{array}$$

- Main typing judgment: $\Gamma \vdash \{\tau\}\ s\ \{\tau'\}$

Atomic updates

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{()\}\ \texttt{insert}\ e\ \{\tau\}}$$

$$\frac{}{\Gamma \vdash \{\tau\}\ \texttt{delete}\ \{()\}}$$

$$\frac{}{\Gamma \vdash \{m[\tau]\}\ \texttt{rename}\ n\ \{n[\tau]\}}$$

Iteration

$$\frac{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash \{\tau\}\ \texttt{iter}[s]\ \{\tau'\}}$$

$$\frac{}{\Gamma \vdash_{\texttt{iter}} \{()\}\ s\ \{()\}}$$

$$\frac{\Gamma \vdash \{\alpha\}\ s\ \{\tau\}}{\Gamma \vdash_{\texttt{iter}} \{\alpha\}\ s\ \{\tau\}}$$

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

$$\vdots$$

High-level language

- Core updates: easy to typecheck, painful to write
- Alternative syntax:

$$
\begin{array}{lcl}
\mathit{Upd} & ::= & \texttt{INSERT}\ (\texttt{BEFORE}|\texttt{AFTER})\ \mathit{Path}\ \texttt{VALUE}\ \mathit{Expr} \\
 & | & \texttt{INSERT AS}\ (\texttt{FIRST}|\texttt{LAST})\ \texttt{INTO}\ \mathit{Path}\ \texttt{VALUE}\ \mathit{Expr} \\
 & | & \texttt{DELETE}\ [\texttt{FROM}]\ \mathit{Path} \\
 & | & \texttt{RENAME}\ \mathit{Path}\ \texttt{TO}\ \mathit{Lab} \\
 & | & \texttt{REPLACE}\ [\texttt{IN}]\ \mathit{Path}\ \texttt{WITH}\ \mathit{Expr} \\
 & | & \texttt{UPDATE}\ \mathit{Path}\ \texttt{BY}\ \mathit{Stmt} \\
\mathit{Path} & ::= & .\ |\ \mathit{Lab}\ |\ \texttt{node()}\ |\ \texttt{text()} \\
 & | & \mathit{Path}/\mathit{Path}\ |\ \mathit{Var}\ \texttt{AS}\ \mathit{Path}\ |\ \mathit{Path}[\mathit{Expr}]
\end{array}
$$

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

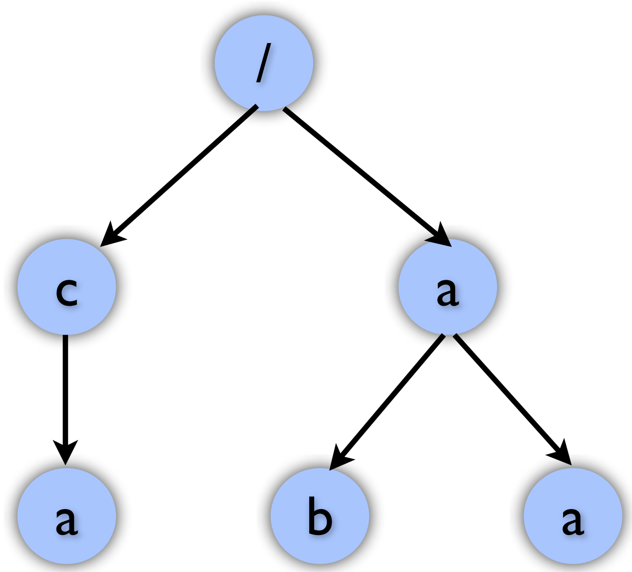
Example

- W3C proposal has counterintuitive (?) semantics

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

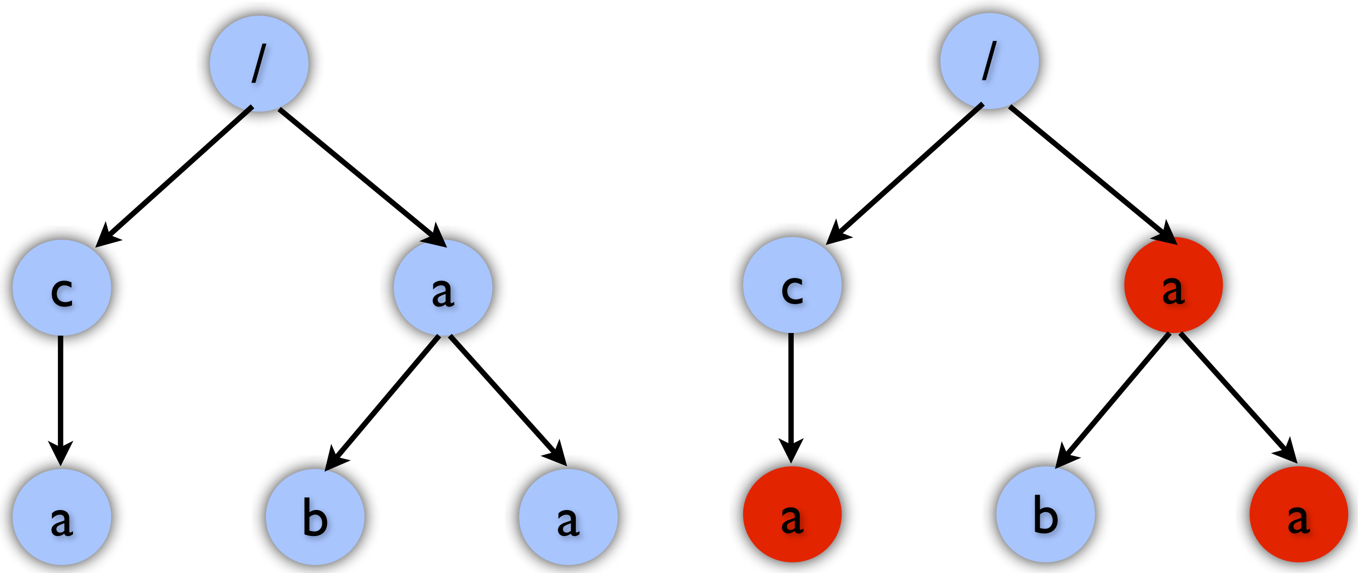
- does **not** do what you (probably) expect

Example



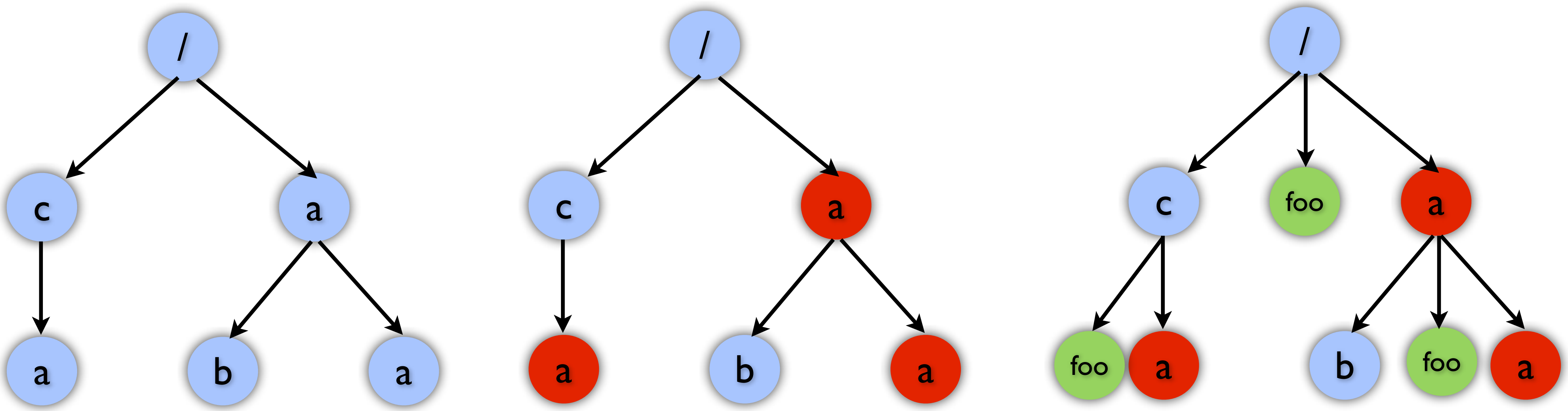

```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

First **collect** updates



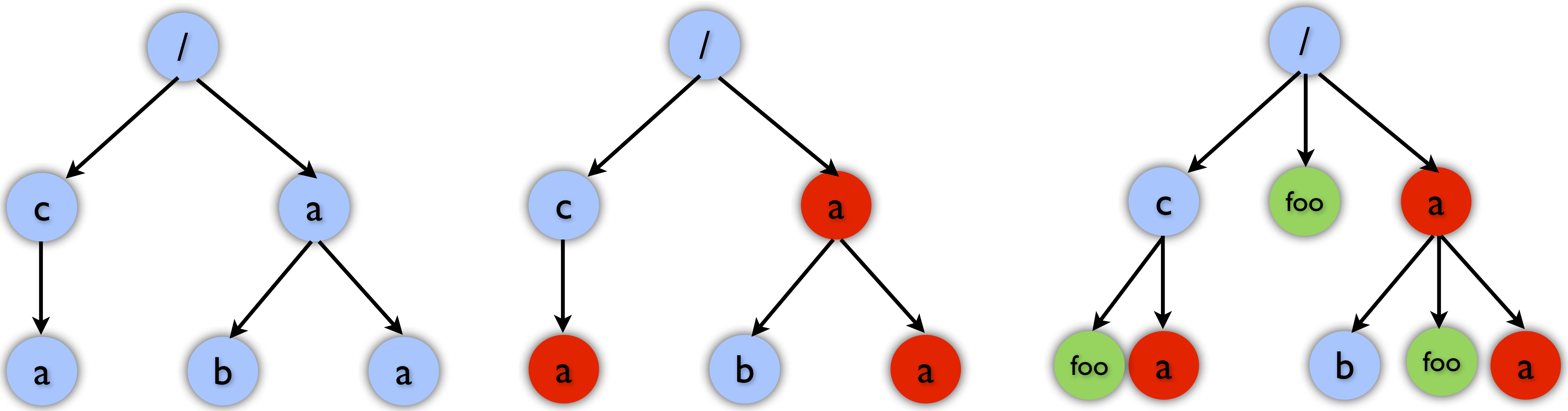

```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

First **collect** updates



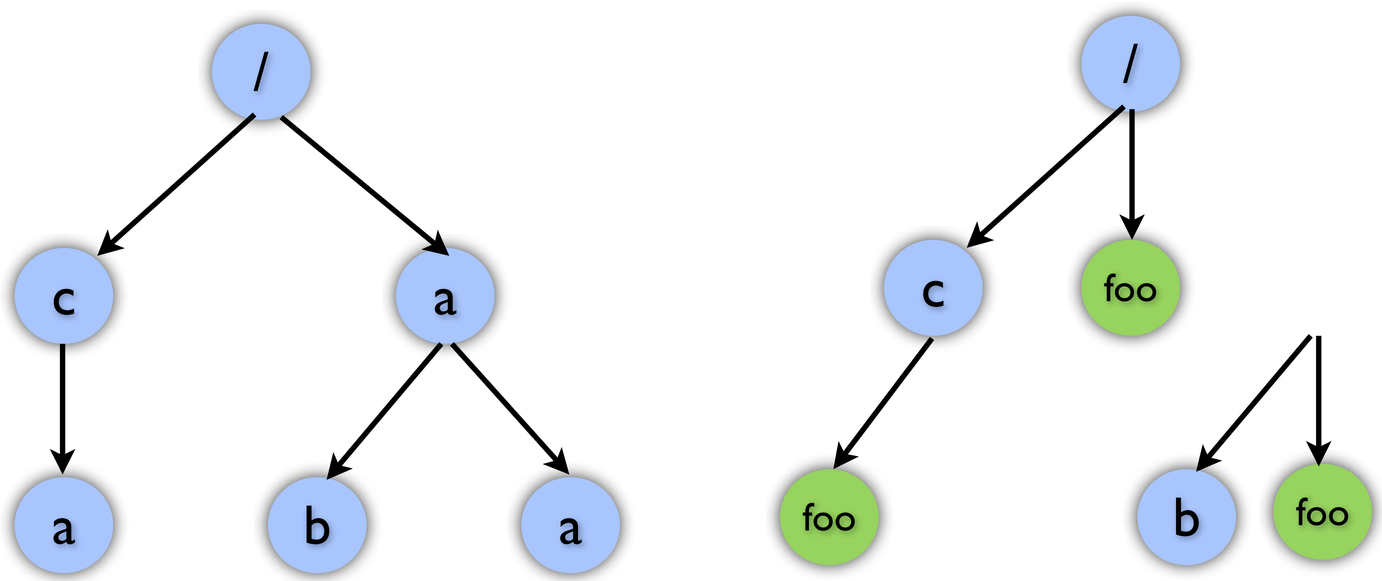

```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

Then **reorder & apply**



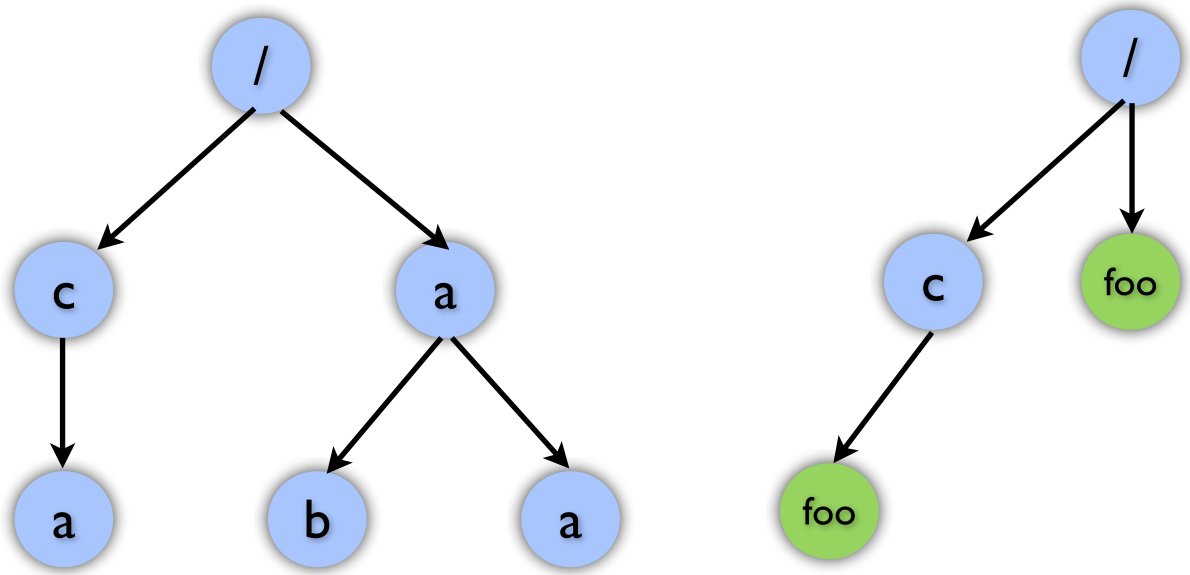

```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

Then **reorder & apply**




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

Then **reorder & apply**




```
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 **effects** of updates
- Step 2: **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

Schemas

- We consider “flat” schemas
  - (close to tree automata)
- Flat types are of the form

$$\tau ::= () \mid T \mid \tau_1, \tau_2 \mid \tau_1 | \tau_2 \mid \tau^*$$

- Flat rules are of the form

$$S \to a[\tau]$$

- Schemas are sets of rules + “root” type

Effects

- Characterize behavior of updates
- Syntax:

$$\begin{array}{rcl} \Omega & ::= & \emptyset \mid \Omega \cup \Omega' \mid insert(\tau, d, T) \mid delete(T) \mid \cdots \\ d & ::= & into \mid into\_as\_first \mid into\_as\_last \mid before \mid after \end{array}$$

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

Effects

- Characterize behavior of updates
- Syntax:

T is a **type name**

$$\begin{array}{rcl} \Omega & ::= & \emptyset \mid \Omega \cup \Omega' \mid insert(\tau, d, T) \mid delete(T) \mid \cdots \\ d & ::= & into \mid into\_as\_first \mid into\_as\_last \mid before \mid after \end{array}$$

- Statically approximate run-time pending update list

Effects

- Characterize behavior of updates
- Syntax

τ is a **regular expression type**

$$\begin{array}{lcl} \Omega & ::= & \emptyset \mid \Omega \cup \Omega' \mid insert(\tau, d, T) \mid delete(T) \mid \cdots \\ d & ::= & into \mid into\_as\_first \mid into\_as\_last \mid before \mid after \end{array}$$

- Statically approximate run-time pending update list

Effect inference

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

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

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

- Inferred effect:

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

where Foo -> foo[string]
```

Schema alteration

- Given **input schema**

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

Schema alteration

- Given **input schema** and **effect**

```
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)}
```

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[]
```

```
Foo -> foo[string]
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)}
```

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[]
```

Also any new types needed for data created by update!

```
{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 1: Inserts

- Inserts happen first:

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

```
Foo -> foo[string]
S′ -> d[T′,(Foo*,U′)]
T′ -> c[(Foo*,A′)*]
U′ -> a[(B′,(Foo*,A′))*]
A′ -> a[]
B′ -> b[]
```

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

Stage 1: Inserts

- Inserts happen first:

```
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[]
```

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

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

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

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