# Querying and storing XML

Week 7
Typechecking and Static Analysis
March 5-8, 2013



# Regular expression types



# Why type-check XML transformations?

- Goal: Check that valid input -> valid output
  - e.g. check that XSLT/XQuery produces valid HTML
- Can we prove this once and for all?
  - rather than revalidating after every run?
- Typechecking can also support other goals:
  - optimization/static analysis
  - identifying silly bugs ("dead" code that never matches anything; missing cases)



# Example: XSLT

- Turns a sequence of People into a table
- HTML Tables have to have at least one row!

```
<xsl:output method="html"/>
  <xsl:template match="/person">
   <html>
    <head><title>Phone Book</title></head>
    <body>
     <table frame="box" rules="all">
      <xsl:apply-templates/>
     </table>
    </body>
   </html>
  </xsl:template>
```

```
<xsl:template match="record">
  <tr>
   <td><xsl:copy-of select="name"/></td>
   <td><xsl:copy-of select="phone"/></td>
  </tr>
 </xsl:template>
```

# Influential approach: XDuce

- A functional language for transforming XML based on pattern matching

```
type Person = person[name[String],
                     email[String]*,
                     phone[String]?]

fun person2row(var p as Person) : Tr =
  match p with
    person[name[var name],
           email[var email]] ->
    tr[td[name],td[email]]
  | ...
```



# XDuce, continued

```
fun people2rows(var xs as Person+) : Tr+ =
  match xs with
    var p as Person, var ps as Person*
      -> person2row p,people2rows ps
  | var p as Person -> person2row p

fun people2table(var xs as Person+) : Table =
  table[people2rows xs]
```



# Regular expression types

- $\tau$ ::= String | X | a[$\tau$] | $\tau$, $\tau$ | $\tau$|$\tau$ | $\tau$*
  - a[$\tau$] means "element labeled a with content $\tau$"
  - X is a *type variable*
- Can model DTDs, XML Schemas as special case
  - at least as far as element structure goes
  - does not model attributes, or XML Schema interleaving



# Recursive type definitions

- Consider type definitions

  type $X_1$ = $\tau_1$

  type $X_2$ = $\tau_2$ ...

- Definitions may be recursive, but recursion must be "guarded" by element tag
- e.g. `type X = X,X` not allowed

# Meaning of types

- Suppose E(X) is the definition of X for each type variable X (i.e. `type X = E(X)` declared)
- Define $[\![T]\!]_E$ as follows:
  - $[\![X]\!]_E = [\![E(X)]\!]_E$
  - $[\![\text{String}]\!]_E = \Sigma^*$
  - $[\![a[T]]\!]_E = \{a[v] \mid v \in [\![T]\!]_E\}$
  - $[\![T_1,T_2]\!]_E = \{v_1,v_2 \mid v_1 \in [\![T_1]\!]_E,\ v_2 \in [\![T_2]\!]_E\}$
  - $[\![T_1|T_2]\!]_E = [\![T_1]\!]_E \cup [\![T_2]\!]_E$
  - $[\![T^*]\!]_E = \{v_1,\ldots,v_n \mid v_1,\ldots,v_n \in [\![T]\!]_E\ \}$



# DTDs as a special case

- DTD rules:

```
a → b,c*,(d|e)?,f
b → c,d  ...
```

- becomes:

```
type A = a[B,C*,(D|E)?,F]
type B = b[C,D] ...
```

- (just introduce new type `Elt` for each element name `elt`)
  - Same idea works for XML Schemas



# Complications

- Typechecking undecidable for any Turing-complete language
  - and for many simpler ones
- Schema languages for XML use regular languages/tree automata
  - decision problems typically EXPTIME-complete
- Nevertheless, efficient-in-practice techniques can be developed



# Other checks

- Typing: All results are contained in declared types
- Exhaustiveness: every possible input matches *some* pattern

```
match (x:Person) with
  person[name[var n]] -> ...
```

- Irredundancy: no pattern is "vacuous"

```
match (x:Person) with
  person[name[var n],var es as Email*] -> ...
| person[name[var n],email[var e]] -> ...
```

- All can be reduced to **subtype** checks

# Subtyping as inclusion

- We can define a semantic subtyping relation:
  - $\tau <: \tau' \Leftrightarrow [\![\tau]\!]_E \subseteq [\![\tau']\!]_E$
- i.e. every tree matching $\tau$ also matches $\tau'$.
- How can we check this?
- One solution: *tree automata*

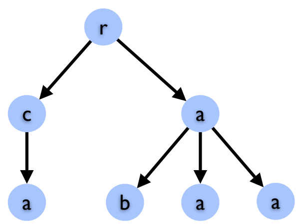


# Type inclusion & tree automata



# XML trees as binary trees

- First-child, next-sibling encoding

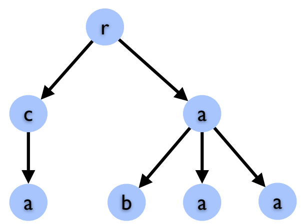


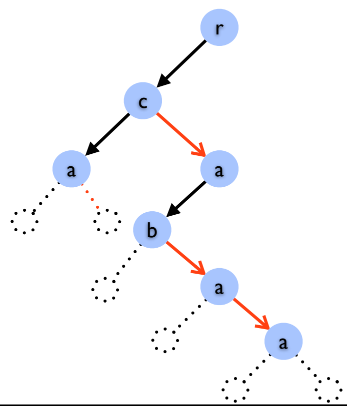

- Leaves are "nil" / ()



# XML trees as binary trees

- First-child, next-sibling encoding




- Leaves are "nil" / ()

# Binary tree automata (bottom-up)

- $M = (Q,\Sigma,\delta,q_0,F)$ is a (bottom-up) binary tree automaton if:
  - $q_0 \in Q$ -- initial state (leaf nodes)
  - $F \subseteq Q$ -- set of final states
  - $\delta : \Sigma \times Q \times Q \rightarrow Q$ -- transition function



# Binary tree automata (top-down)

- $M = (Q,\Sigma,\delta,q_0,F)$ is a (top-down) binary tree automaton if:
  - $q_0 \in Q$ -- initial state (root node)
  - $F \subseteq Q$ -- set of final states (for all leaf nodes)
  - $\delta : Q \times \Sigma \rightarrow Q \times Q$ -- transition function
- Determinstic top-down TA are strictly less expressive than bottom-up TA
  - however, nondeterministic versions equivalent



# DTDs and ambiguity

- Recall DTDs have to be "unambiguous"
- Example: NOT `(a+b)+(a+c)`
  - but `a+b+c` is equivalent
- NOT `(a+b)*a`
  - but `(b*a)+` is equivalent
- These restrictions (and similar ones for XML Schemas) lead to deterministic, top-down (tree) automata.



# Translating types

# Translating types

- First give each subexpression its own type name

```
type Person = person[name[String],
                     email[String]*,
                     phone[String]?]
```



# Translating types

- First give each subexpression its own type name

```
type Person = person[name[String],
                     email[String]*,
                     phone[String]?]

type Person = person[Name,
                     Emails,
                     PhoneOpt]
type Name     = name[String]
type Emails   = email[String], Emails | ()
type PhoneOpt = phone[String] | ()
```



# To binary form

- Next reorganize into binary steps

```
type Person  = person[Q1],Empty
type Q1 = name[String], Q2
        | name[String], Q3
type Q2 = email[String],Q2
        | email[String],Q3
type Q3 = phone[String],Empty | ()
type Empty = ()
```



# To tree automaton

```
type Person  = person[Q1],Empty
type Q1 = name[String], Q2
        | name[String], Q3
type Q2 = email[String],Q2
        | email[String],Q3
type Q3 = phone[String],Empty | ()
type Empty = ()
```

Q = {Person,Q1,Q2,Q3,String,Empty}
$\Sigma$ = {person,name,email,phone,string}
$q_0$ = Person
F = {Empty,Q3,String}

# To tree automaton

```
type Person  = person[Q1],Empty
type Q1 = name[String], Q2
        | name[String], Q3
type Q2 = email[String],Q2
        | email[String],Q3
type Q3 = phone[String],Empty | ()
type Empty = ()
```

Q = {Person,Q1,Q2,Q3,Stri

Σ = {person,name,email,ph

$q_0$ = Person

F = {Empty,Q3,String}

| $q_1$ | $q_2$ | a | $\delta(q_1,q_2,a)$ |
|---|---|---|---|
| Q1 | Empty | person | Person |
| String | Q2 | name | Q1 |
| String | Q3 | name | Q1 |
| String | Q2 | email | Q2 |
| String | Q3 | email | Q3 |
| String | Empty | phone | Q3 |



# Checking containment

- Like sequential case, tree automata / languages closed under union, intersection, negation
  - some of the constructions incur exponential blowup
- Containment testing is EXPTIME-complete
  - Naive algorithm: test $L(M) \subseteq L(N)$ by constructing automaton M' for $L(M) \cap L(\overline{N})$, testing emptiness
  - Does not perform well on problems encountered in programming
  - Hosoya, Vouillon & Pierce [2005] give practical algorithm aimed at XDuce typing problems



# Types for XQuery



# Motivation

[Collazzo et al. 2006]

- For XDuce, want to ensure valid result
  - find silly bugs in pattern matching
  - ensure exhaustiveness & irredundancy
- For XQuery, still want result validity, but different issues
  - functions, but no pattern matching (as such)
  - but does have for/iteration & path expressions
  - parts of queries can have "silly" bugs

# Example

- Schema (we'll use XDuce notation):

```
type Person = person[name[String],
                     email[String]*,
                     phone[String]?]
```

- Query: return all name/phone of people with email

```
for $y in $x/person[emial]
return $y/name,$y/phone
```

- **Silly bug**: "emial" misspelled; path will never select anything (in this type anyway)



# XQuery core language: µXQ

$$e ::= \texttt{()} \mid e, e' \mid n[e] \mid w \mid x \mid \texttt{let}\ x = e\ \texttt{in}\ e' \\ \mid\ b \mid \texttt{if}\ c\ \texttt{then}\ e\ \texttt{else}\ e' \mid \bar{x} \mid \bar{x}/\texttt{child} \mid e :: n \mid \texttt{for}\ \bar{x} \in e\ \texttt{return}\ e'$$

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

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \bar{x}{:}\alpha$$

Tree variables $\bar{x}$ have single tree values $\bar{v}$
Forest variables x have arbitrary values v



# Typing rules for XQuery: Basics

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

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{()}} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash n[e] : n[\tau]} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e, e' : \tau, \tau'}$$



# Typing rules for XQuery: Let, If

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash c : \texttt{bool} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1 | \tau_2}$$

# Path steps

$$\frac{\bar{x}{:}n[\tau] \in \Gamma}{\Gamma \vdash \bar{x}/\texttt{child} : \tau} \qquad \frac{\Gamma \vdash e : \tau \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e :: n : \tau'}$$

$$\frac{}{n[\tau] :: n \Rightarrow n[\tau]}$$

$$\frac{E(X) :: n \Rightarrow \tau}{X :: n \Rightarrow \tau} \qquad \frac{\alpha \neq n[\tau]}{\alpha :: n \Rightarrow ()}$$

$$\frac{}{() :: n \Rightarrow ()} \qquad \frac{\tau_1 :: n \Rightarrow \tau_2}{\tau_1^* :: n \Rightarrow \tau_2^*}$$

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

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



# Comprehensions: simple idea (what XQuery does)

$$\frac{\begin{array}{c}\text{statEnv } \mathbf{|\text{-}}\ \mathit{Expr}_1 : \underline{\mathit{Type}}_1 \\ \text{statEnv } \mathbf{|\text{-}}\ \mathit{VarName}_1\ \textbf{of var expands to}\ \mathit{Variable}_1 \\ \text{statEnv + varType(}\mathit{Variable}_1 \text{ => prime(}\underline{\mathit{Type}}_1\text{))}\ \mathbf{|\text{-}}\ \mathit{Expr}_2 : \underline{\mathit{Type}}_2\end{array}}{\text{statEnv } \mathbf{|\text{-}} \text{ for \$}\mathit{VarName}_1 \text{ in } \mathit{Expr}_1 \text{ return } \mathit{Expr}_2 : \underline{\mathit{Type}}_2 \cdot \text{quantifier(}\underline{\mathit{Type}}_1\text{)}}$$

| | | |
|---|---|---|
| prime(*FormalItemType*) | = | *FormalItemType* |
| prime(empty) | = | none |
| prime(none) | = | none |
| prime(*Type*₁ , *Type*₂) | = | prime(*Type*₁) I prime(*Type*₂) |
| prime(*Type*₁ & *Type*₂) | = | prime(*Type*₁) I prime(*Type*₂) |
| prime(*Type*₁ I *Type*₂) | = | prime(*Type*₁) I prime(*Type*₂) |
| prime(*Type*?) | = | prime(*Type*) |
| prime(*Type**) | = | prime(*Type*) |
| prime(*Type*+) | = | prime(*Type*) |

| | | |
|---|---|---|
| quantifier(*FormalItemType*) | = | 1 |
| quantifier(empty) | = | ? |
| quantifier(none) | = | 1 |
| quantifier(*Type*₁ , *Type*₂) | = | quantifier(*Type*₁) , quantifier(*Type*₂) |
| quantifier(*Type*₁ & *Type*₂) | = | quantifier(*Type*₁) , quantifier(*Type*₂) |
| quantifier(*Type*₁ I *Type*₂) | = | quantifier(*Type*₁) I quantifier(*Type*₂) |
| quantifier(*Type*?) | = | quantifier(*Type*) · ? |
| quantifier(*Type**) | = | quantifier(*Type*) · * |
| quantifier(*Type*+) | = | quantifier(*Type*) · + |



# Example

```
type Doc = a[ b[d[]]*, c[e[], f[]]? ]
prime( b[d[]]*, c[e[], f[]]? )       = b[d[]]|c[e[],f[]]
quantifier( b[d[]]*, c[e[], f[]]? ) = *
```

```
$doc : Doc ⊢ $doc/a/* : b[d[]]*, c[e[], f[]]?
$doc : Doc, $x : b[d[]] | c[e[], f[]] ⊢ $x/* : d[] | e[],f[]
```

---

```
$doc : Doc ⊢ for $x in $doc/a/* return $x/* : (d[] | e[],f[])*
```

**Overapproximation:**

`e,f,d` and `e,f,e,f` can never happen



# Comprehensions: more precisely

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \rightarrow e_2 : \tau_2}{\Gamma \vdash \texttt{for}\ \bar{x} \in e_1\ \texttt{return}\ e_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash \bar{x}\ \texttt{in}\ () \rightarrow e : ()} \qquad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ E(X) \rightarrow e : \tau}{\Gamma \vdash \bar{x}\ \texttt{in}\ X \rightarrow e : \tau}$$

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

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

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

# Example

```
type Doc = a[ b[d[]]*, c[e[], f[]]? ]
prime( b[d[]]*, c[e[], f[]]? )      = b[d[]]|c[e[],f[]]
quantifier( b[d[]]*, c[e[], f[]]? ) = *
```

```
$doc : Doc ⊢ $doc/a/* : b[d[]]*, c[e[], f[]]?
$doc : Doc ⊢ $x in b[d[]]* , c[e[], f[]]? → $x/* : d[]*, (e[],f[])?
--------------------------------------------------------------------
$doc : Doc ⊢ for $x in $doc/a/* return $x/* : d[]*, (e[],f[])?
```

Remembers more of the regexp structure
(but overall system still approximate)



# Path errors

- Detect *path errors* by:
  - Noticing when a non-() expression has type ()
  - Tracking which parts of expression have this property
  - (propagating sets of expression locations through typing judgements)

(TypeEmpty)
$$\frac{WF(E;\,\Gamma \vdash_\beta () : ((); \emptyset))}{E;\,\Gamma \vdash_\beta () : ((); \emptyset)}$$

(TypeAtomic)
$$\frac{WF(E;\,\Gamma \vdash_\beta b : (B; \emptyset))}{E;\,\Gamma \vdash_\beta b : (B; \emptyset)}$$

(TypeVarLet)
$$\frac{x : T \in \Gamma \quad WF(E;\,\Gamma \vdash_\beta x : (T; \emptyset))}{E;\,\Gamma \vdash_\beta x : (T; \emptyset)}$$

(TypeVarFor)
$$\frac{\overline{x} : T \in \Gamma \quad WF(E;\,\Gamma \vdash_\beta \overline{x} : (T; \emptyset))}{E;\,\Gamma \vdash_\beta \overline{x} : (T; \emptyset)}$$

(TypeElem)
$$\frac{E;\,\Gamma \vdash_{\beta.0} Q : (T; \mathscr{S})}{E;\,\Gamma \vdash_\beta l[Q] : (l[T]; \mathscr{S})}$$

(TypeForest)
$$\frac{E;\,\Gamma \vdash_{\beta.0} Q_1 : (T_1; \mathscr{S}_1) \quad E;\,\Gamma \vdash_{\beta.1} Q_2 : (T_2; \mathscr{S}_2)}{E;\,\Gamma \vdash_\beta Q_1, Q_2 : (T_1, T_2; \mathscr{S}_1 \cup \mathscr{S}_2)}$$



# Path errors

- For expressions that visit same subexpression multiple times
  - only error if subexpression *never* produces useful output
  - take *intersection*
- e.g. for $x in $doc/person[emial] ...
  - not an error if $doc : person[email[...] | emial[...]]



# Subtleties

- [Collazzo et al. 2006] also considers *splitting*
- E.g. considering a[$T_1$|...|$T_n$] as a[$T_1$]|...|a[$T_n$]
- This complicates system, but can make results more precise

# Schema-based query-update independence analysis



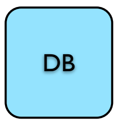
# Motivation

[Benedikt & Cheney, 2009]

- Suppose we want to maintain multiple (materialized) views or constraints
  - expressed by queries $Q_1$, $Q_2$, ...
- When the database is updated
  - If we can determine (quickly) that query and update are *independent*
  - then can skip view/constraint maintenance

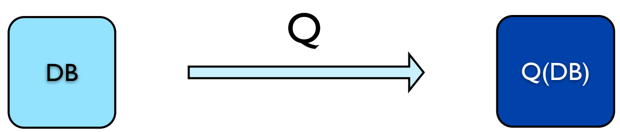


# Query-update independence

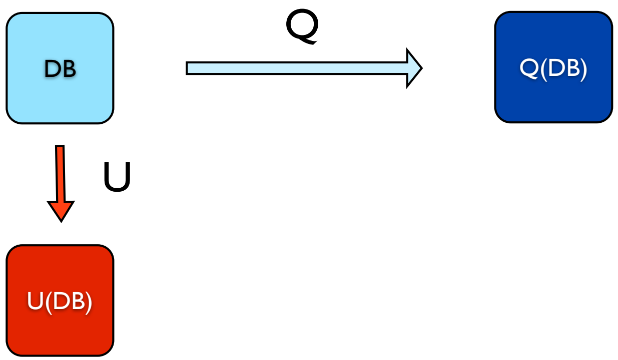





# Query-update independence

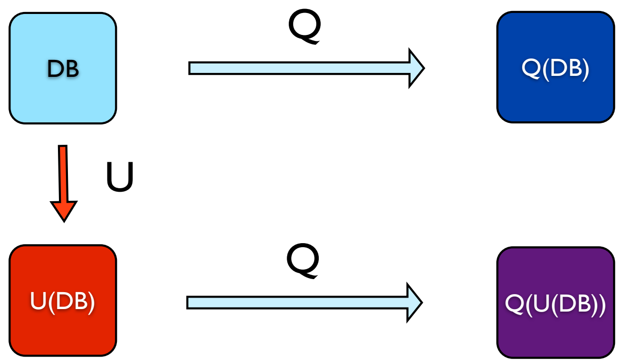


# Query-update independence



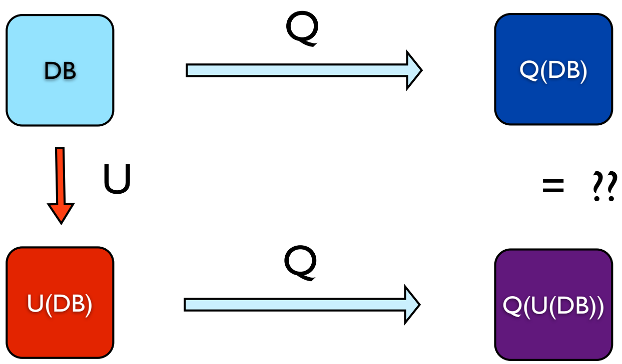



# Query-update independence






# Query-update independence

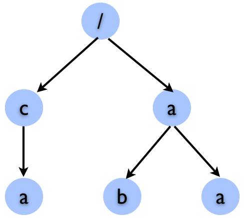





# Approach

- a **static** analysis
- that safely detects **independence**
- and takes **schema** into account
- for XQuery **Update** with all XPath axes
- **fast** enough to be useful as an optimization

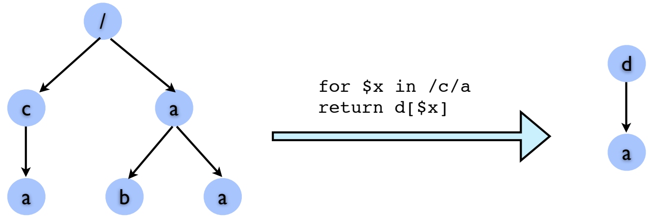
# Independence example




```
for $x in /c/a
return d[$x]
```

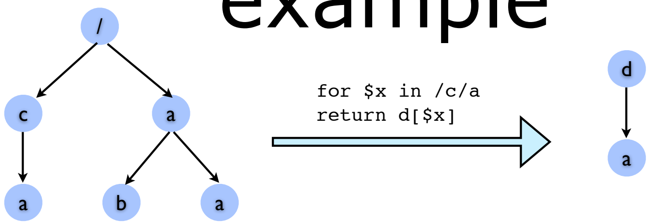


# Independence example



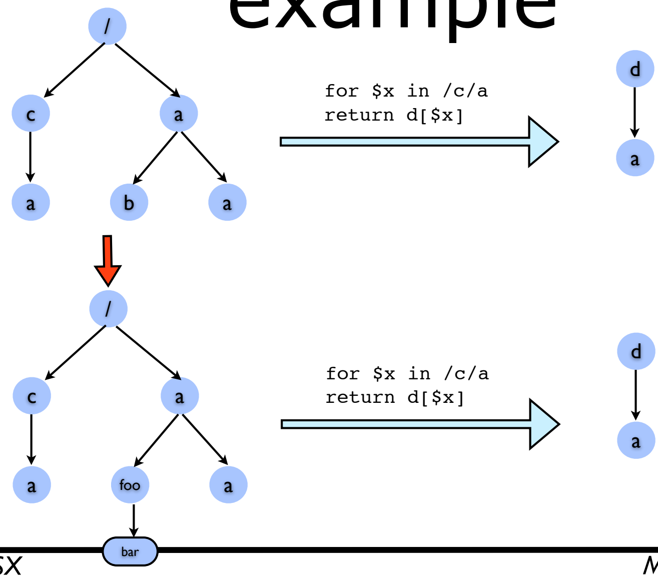



# Independence example

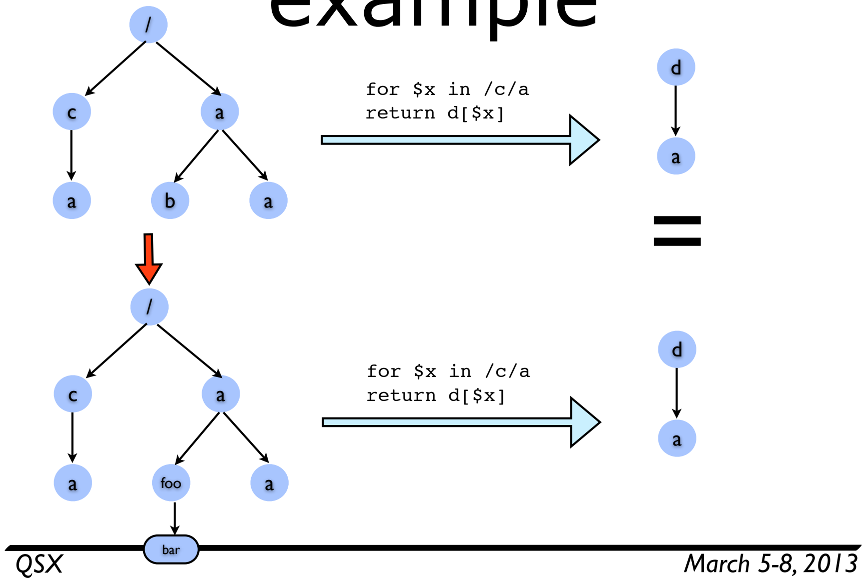





# Independence example

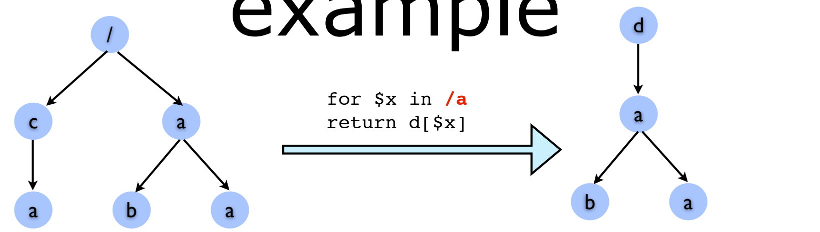


# Independence example



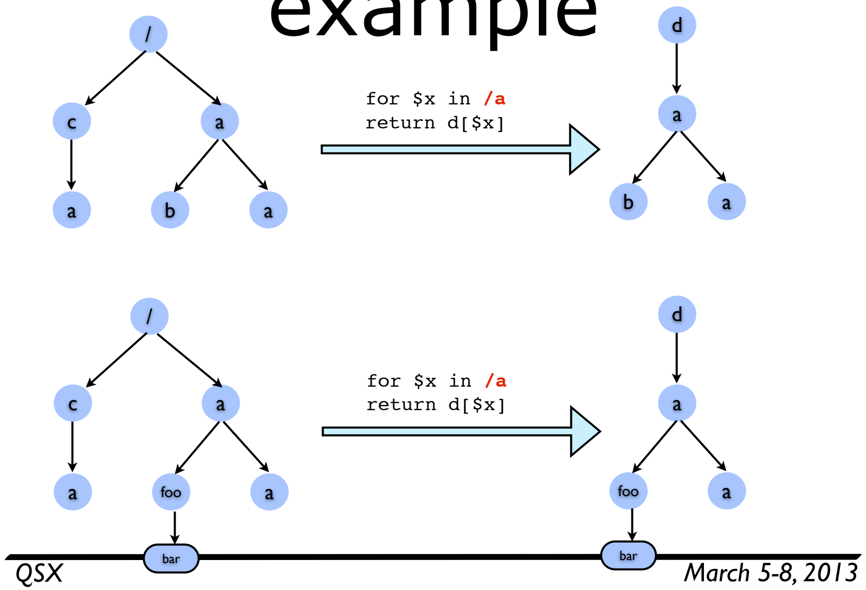



# Independence **non-** example



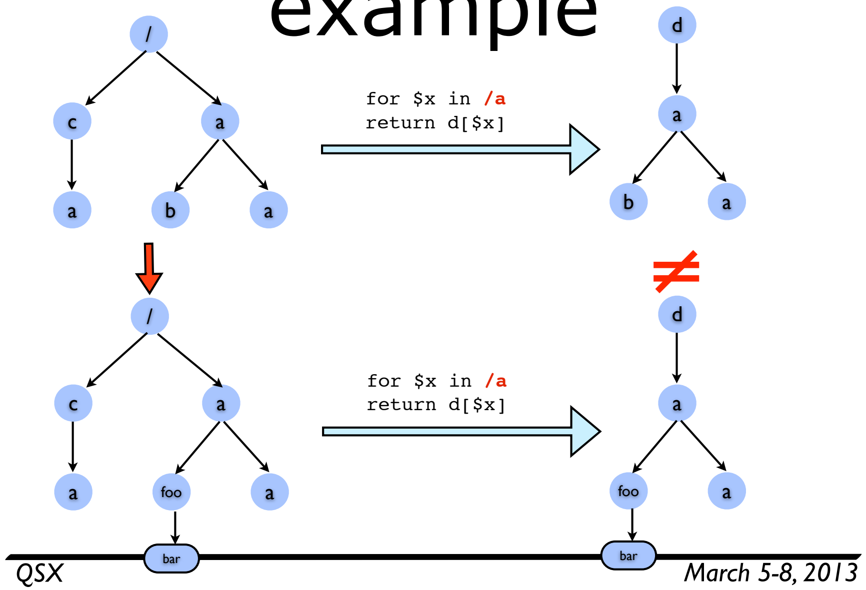



# Independence **non-** example






# Independence **non-** example



# Approach

- Exploits a **schema** *S* that describes the input
- Statically calculate:
  - *c* = **Copied nodes** of *Q*
  - *a* = **Accessed Nodes** of Q
  - *u* = **Updated Nodes** of U
- *c, a, u* are sets of type names in *S*

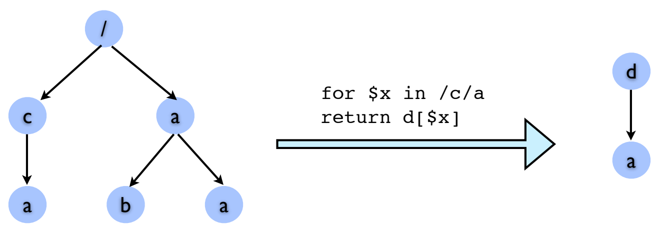


# Independence test

- We show that Q and I are independent modulo S if:
- *a* is disjoint from *u*
  - that is, the update has no impact on **accessed** nodes
- and *c//** is disjoint from *u*
  - that is, the update does not impact any copied nodes **or their descendants**

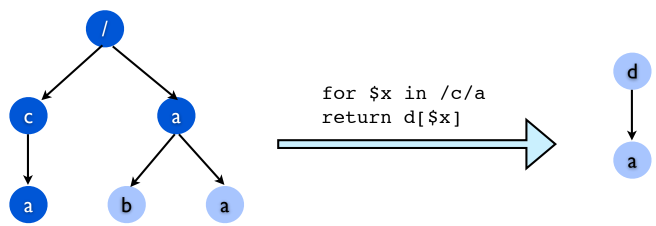


# Analysis example

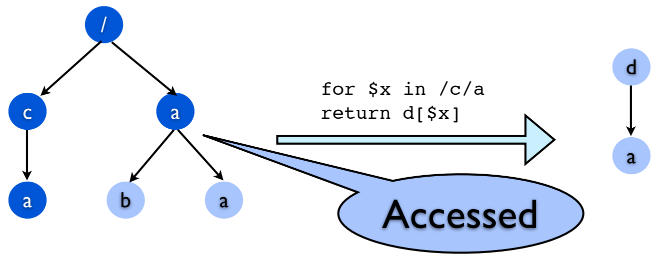





# Analysis example

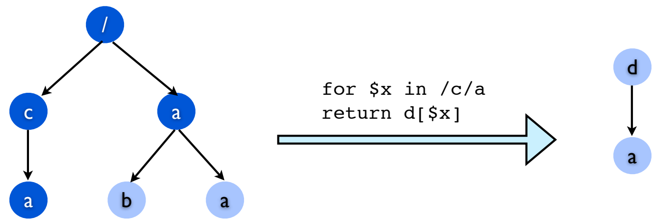


# Analysis example



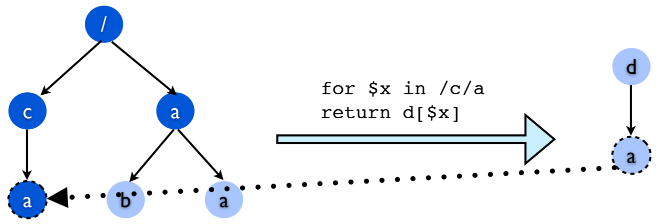



# Analysis example



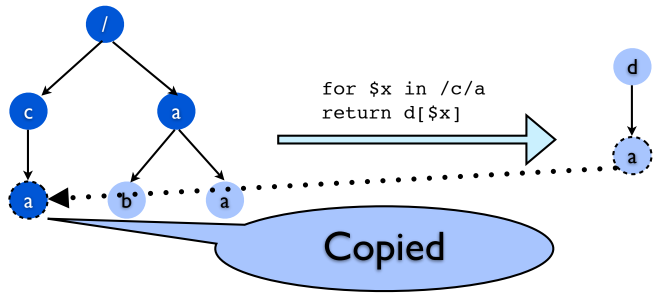



# Analysis example

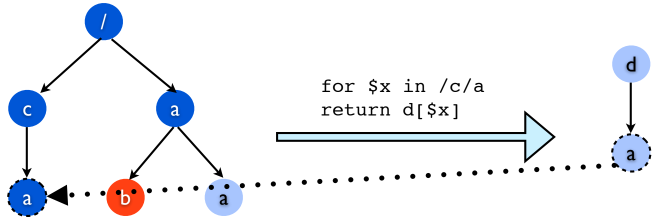





# Analysis example

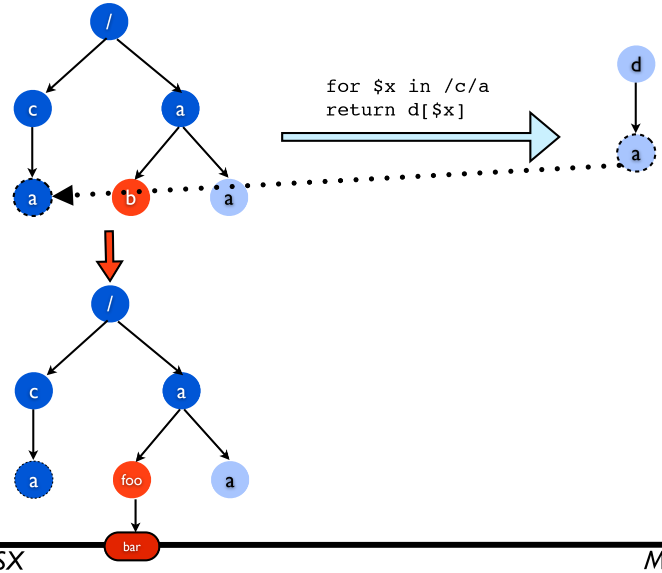


# Analysis example



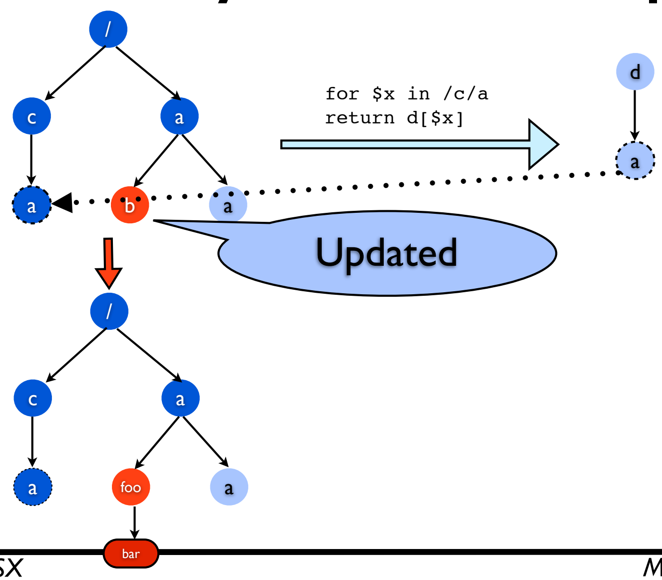



# Analysis example



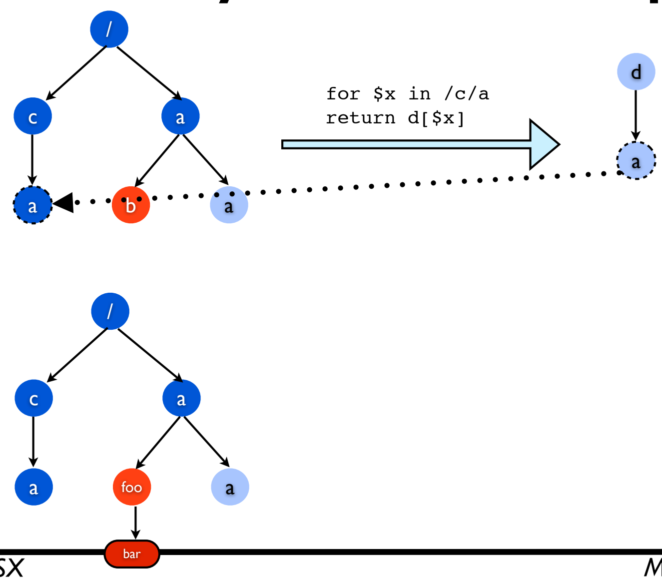



# Analysis example

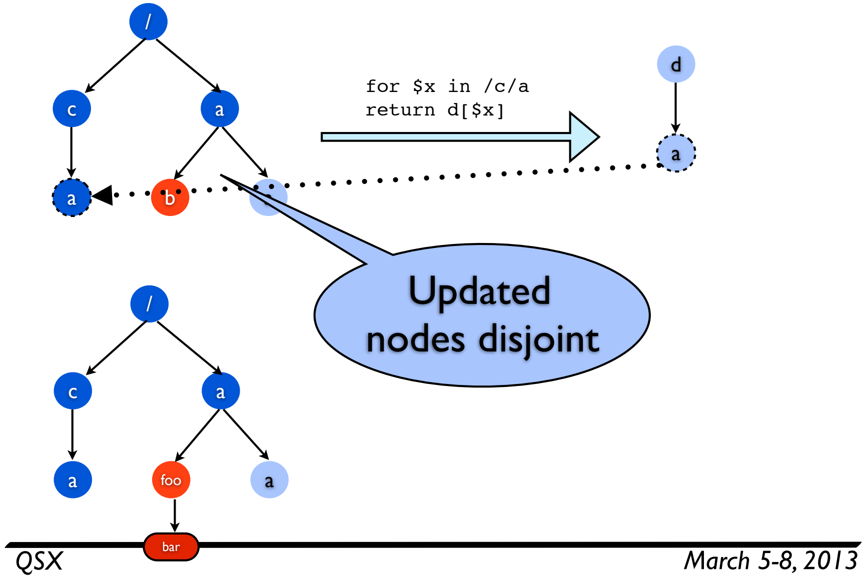





# Analysis example

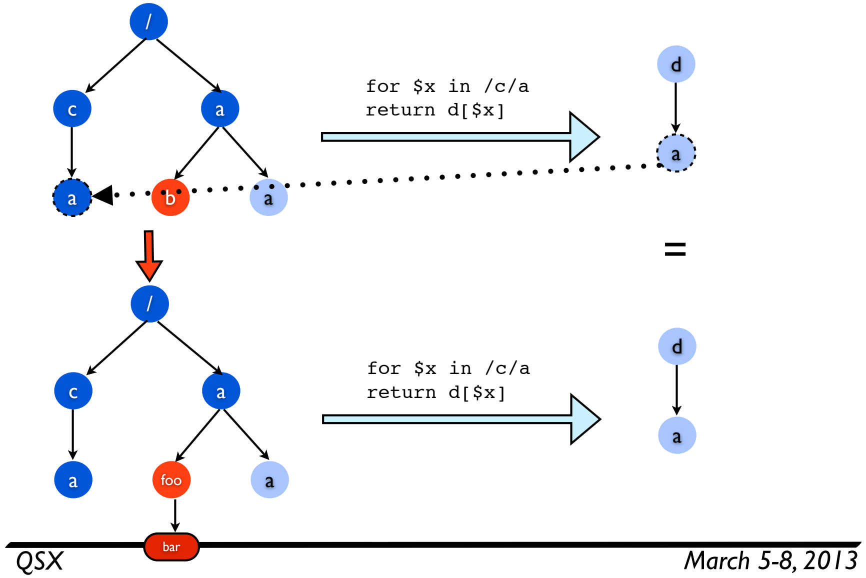


# Analysis example



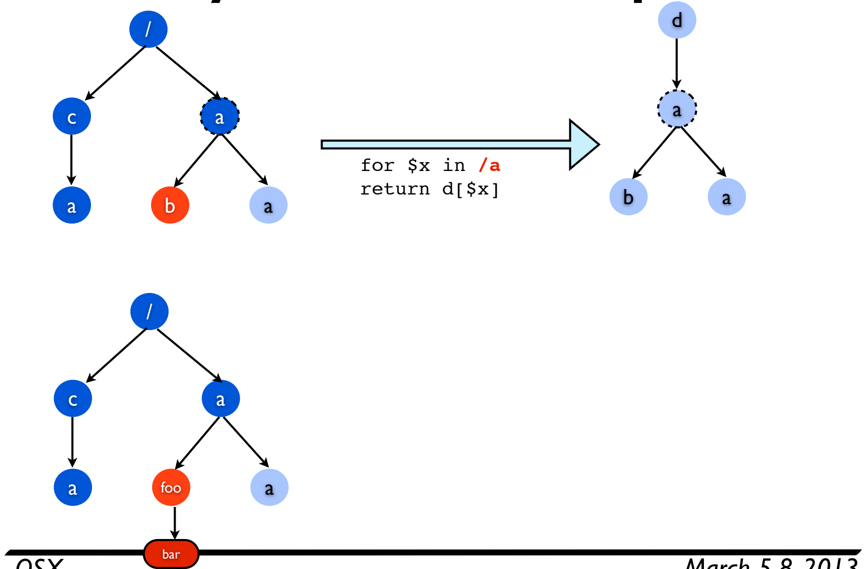



# Analysis example



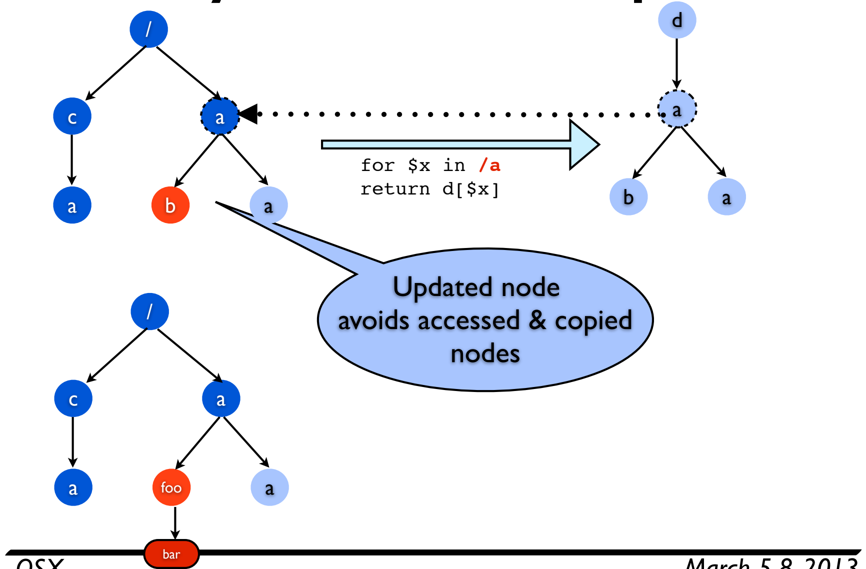



# Analysis example II

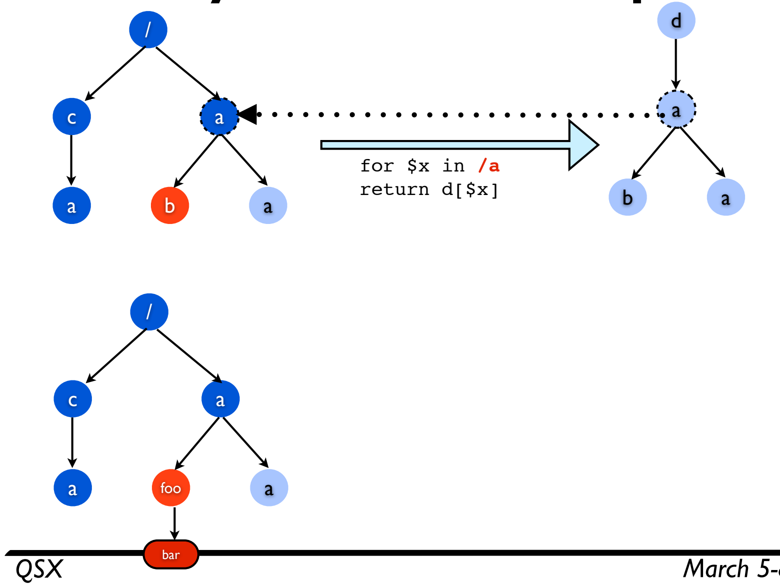





# Analysis example II

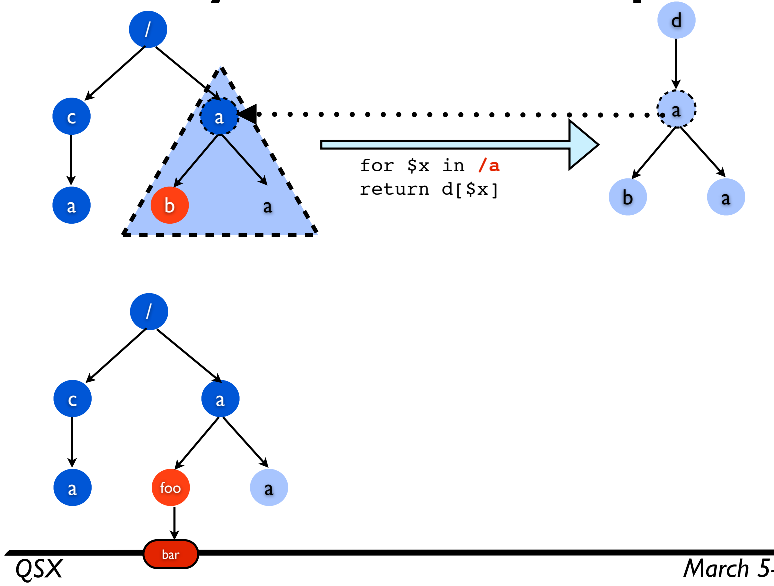


# Analysis example II



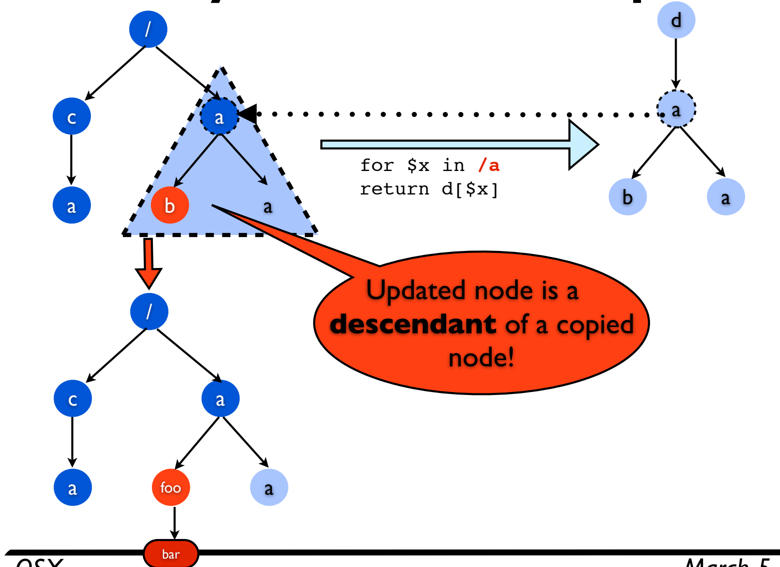



# Analysis example II



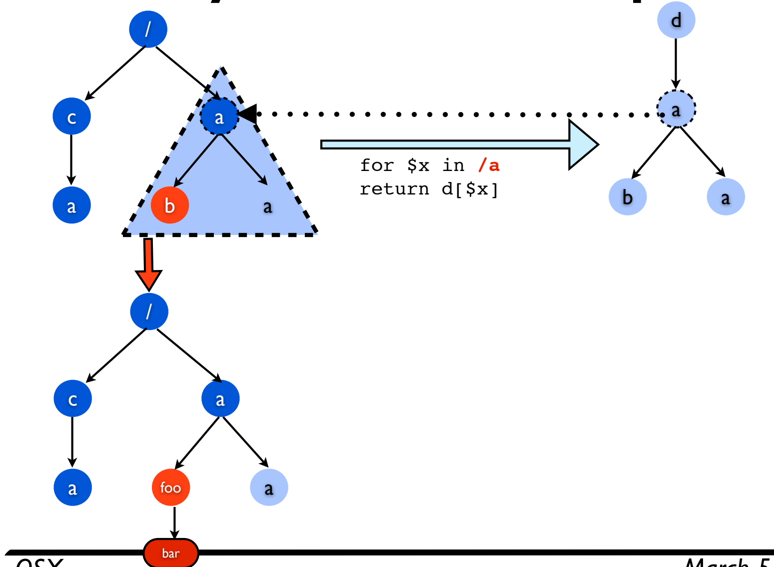



# Analysis example II

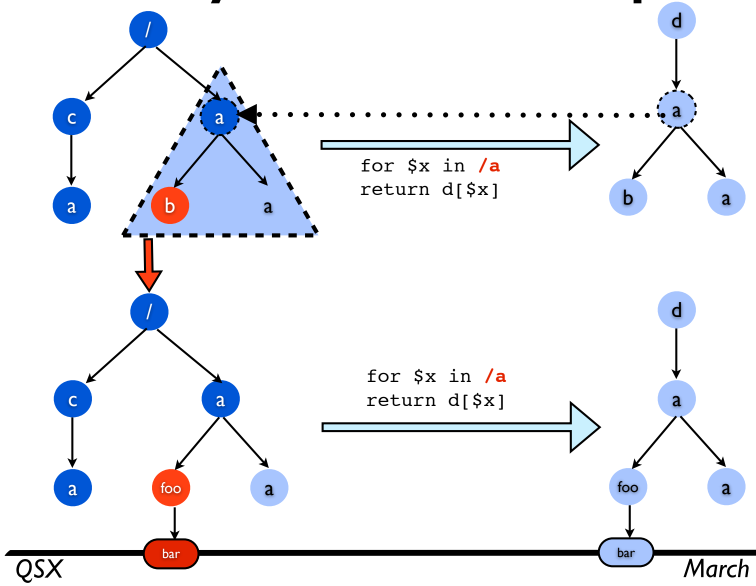





# Analysis example II

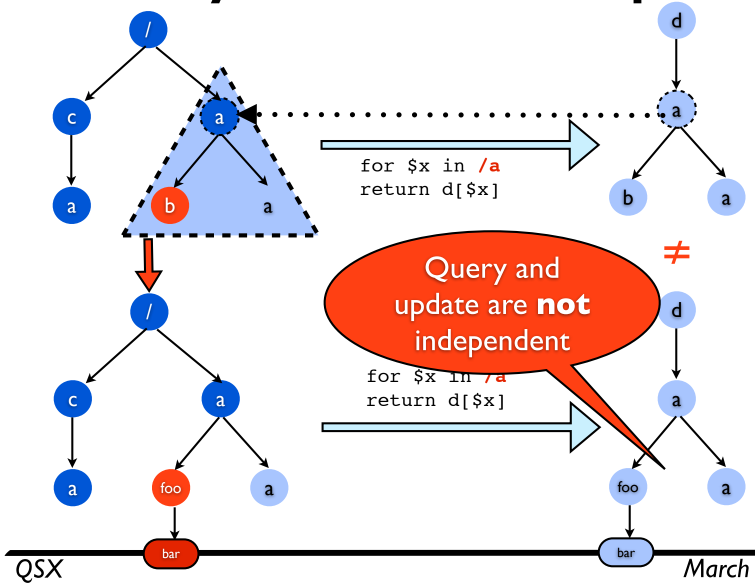


# Analysis example II






# Analysis example II






# Correctness

- We can use **type names** for sets of accessed, copied, updated nodes
- Symbolically evaluate query/update over **schema**
- However, there is a complication:

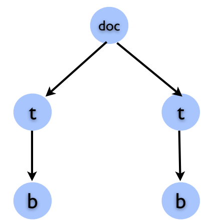
**Aliasing**

- How do we know that it is safe to use type names to refer to and change "parts" of schema?



# Problem

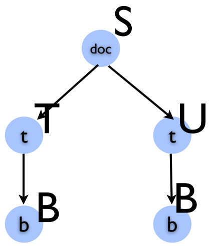
- For example:




$$
\begin{array}{lcl}
S & \to & doc[T^*, U, T^*] \\
T & \to & t[A|B] \\
U & \to & t[B|C] \\
A & \to & a[] \\
B & \to & b[] \\
C & \to & c[]
\end{array}
$$

# Problem

- For example:

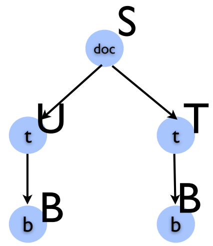



$$\begin{array}{lcl} S & \rightarrow & doc[T^*, U, T^*] \\ T & \rightarrow & t[A|B] \\ U & \rightarrow & t[B|C] \\ A & \rightarrow & a[] \\ B & \rightarrow & b[] \\ C & \rightarrow & c[] \end{array}$$



# Problem

- For example:




$$\begin{array}{lcl} S & \rightarrow & doc[T^*, U, T^*] \\ T & \rightarrow & t[A|B] \\ U & \rightarrow & t[B|C] \\ A & \rightarrow & a[] \\ B & \rightarrow & b[] \\ C & \rightarrow & c[] \end{array}$$

- has multiple typings



# Solution: Closure under Aliasing

- Keep track of which type names “alias”
  - or, can refer to the same node
  - currently, based on tree language overlap test
- Close sets a, c, u under aliasing
  - example: any T could be a U, and vice versa
- Note: This may be overkill
  - unnecessary (has no effect) for DTDs
  - probably unnecessary for unambiguous XML Schema



# Summary

- Independence analysis can statically determine whether view needs to be maintained
- Subsequent work explores more precise static information:
  - path-based independence analysis ("destabilizers") - no schema required [Cheney & Benedikt VLDB 2010]
  - independence based on more precise type analysis [Bidoit-Tollu et al. VLDB 2012]
- Future work:
  - combine path- and schema-based approach?
  - what about XML views of relations?
  - can static analysis help with incremental maintenance?

# Next week

- Provenance
  - Why and Where: A characterization of data provenance
  - Provenance management in curated databases
  - Annotated XML: queries and provenance