Bidirectional Data Transformation by Calculation

Hugo Pacheco

HASLab
INESC TEC & University of Minho, Braga, Portugal
Former

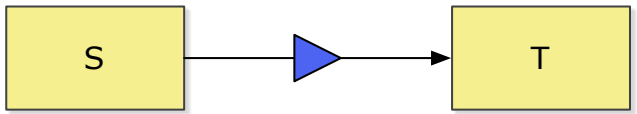
55th ToPS Seminar

Tokyo - December 18th 2012

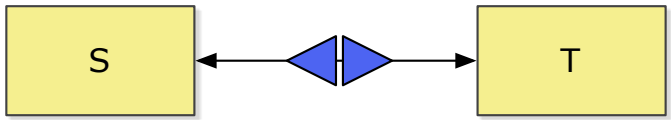
Outline

Data Transformations

- data transformations abound in software engineering
- essential to convert data between different formats

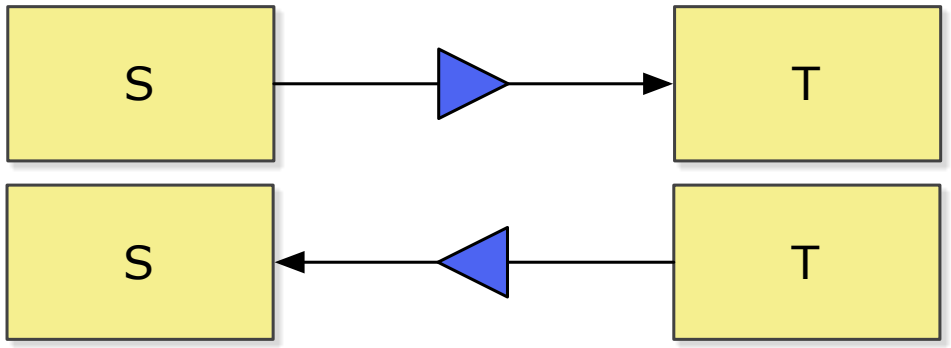


- in real model-driven software engineering scenarios, we often need to run a transformation in both directions



- a bidirectional transformation (BX)

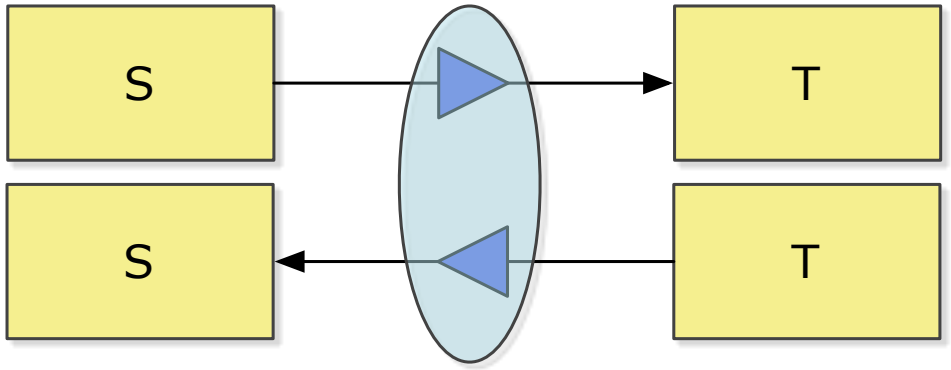
(*Ad hoc*) Bidirectional Transformations



Manual design: two separate transformations

- expensive
- error-prone
- a maintenance problem

Bidirectional Languages

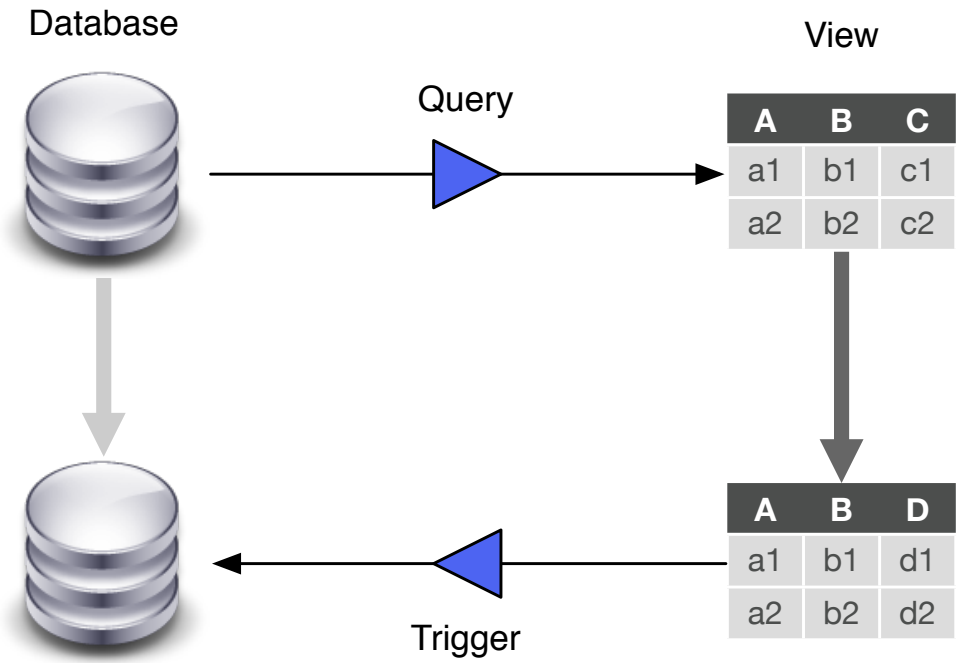


Combinatorial design: the same specification denotes both

- nice syntax
- clean semantics
- compositional

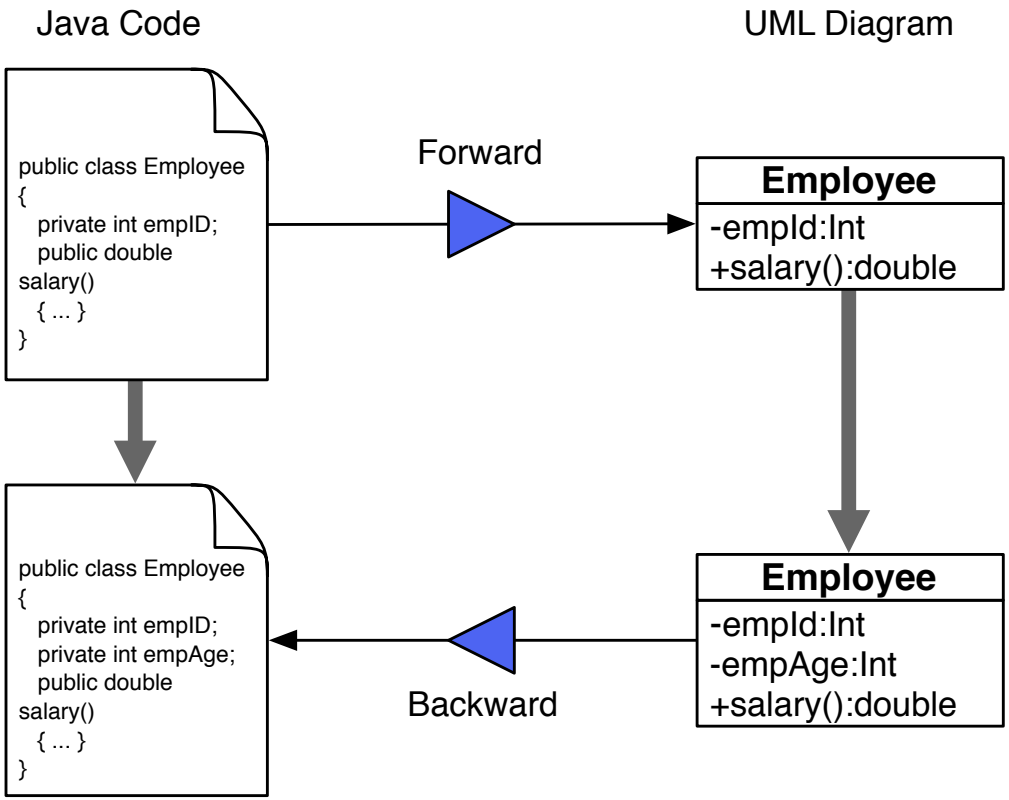
Bidirectional Languages exist for ...

...databases...



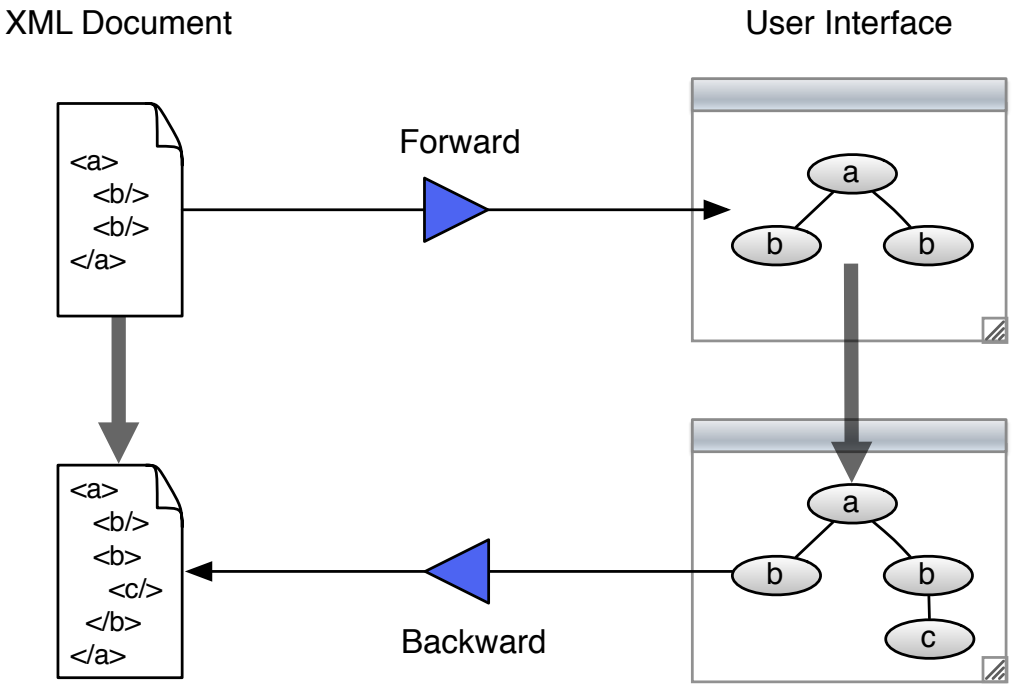
Bidirectional Languages exist for ...

...model-driven software engineering...



Bidirectional Languages exist for ...

...user interfaces...

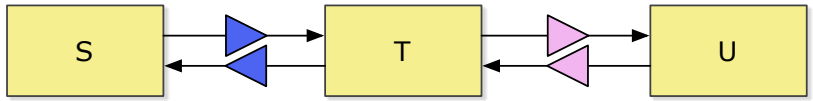


...etc

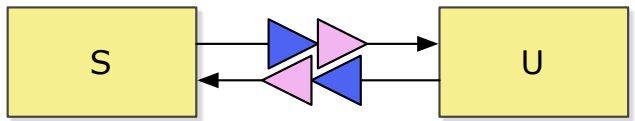
1 Efficiency

Motivation - Optimization

- combinatorial approaches build complex transformations by composition



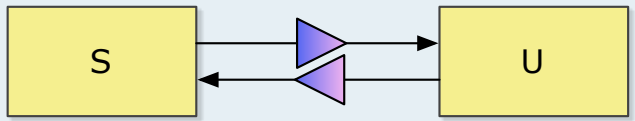
- composition ⇒ cluttering ⇒ inefficiency!



- a serious implementation of BXs needs to be efficient

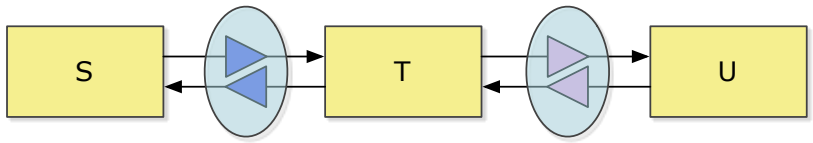
Question

- how to optimize bidirectional transformations?

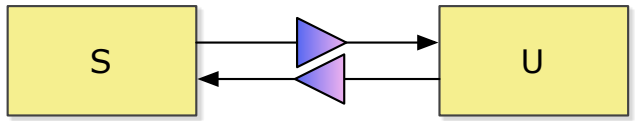


Motivation - Is BX optimization really hard?

- write the two transformations in a language with support for optimization



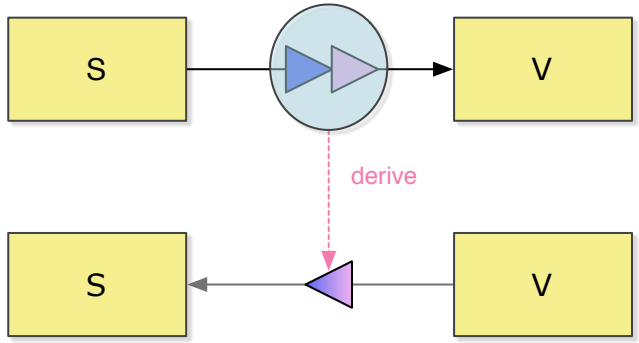
- optimize both independently



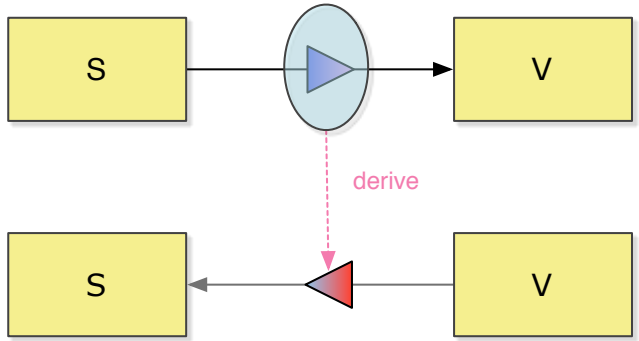
- twice the effort!
- no longer a single program!

Motivation - Is BX optimization really hard?

- write the forward transformation in a language with support for optimization and derive the backward transformation



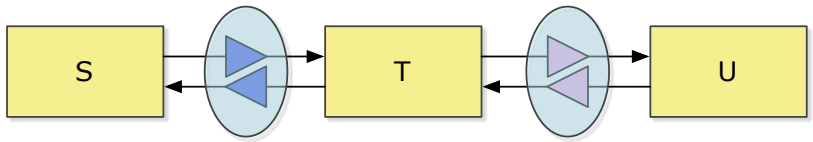
- optimize the forward transformation and derive the other



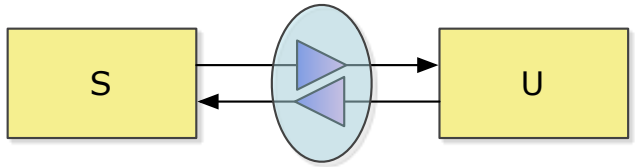
- different semantics!

Motivation - A better solution

- write the BX in a language with support for optimization



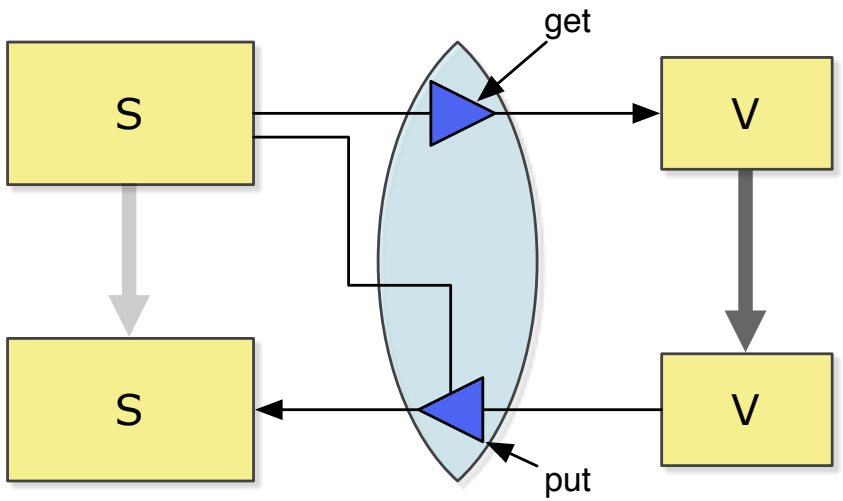
- optimize bidirectional programs



- optimized BX program
- same semantics

Bidirectional Lenses

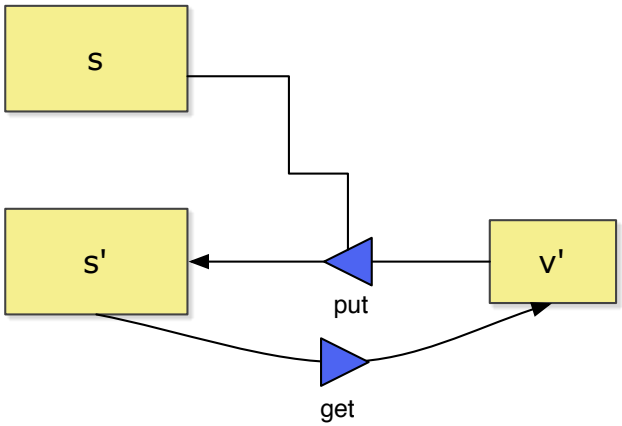
- Foster et al. proposed the framework of lenses [POPL'05,TOPLAS]
- lenses are one of the most popular BX frameworks



The Lens Laws

- PutGet law

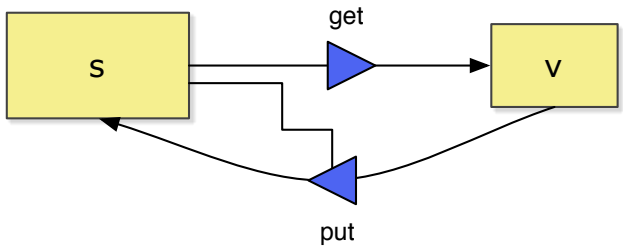
  put must translate view updates exactly.



$$get\ (put\ v'\ s) = v'$$

- GetPut law

  put must preserve null view updates.



$$put\ (get\ s)\ s = s$$

A Point-free Design

- A data domain (algebraic data types)

$$\textbf{data}\ [a] = [] \mid a : [a]$$
$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

- A language syntax

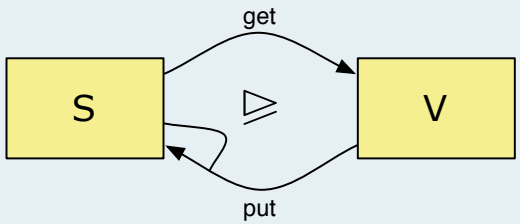
$$\mathit{id} : A \to A$$
$$\circ : (B \to C) \to (A \to B) \to (A \to C)$$
$$\pi_1 : A \times B \to A$$
$$\times : (A \to C) \to (B \to D) \to (A \times B \to C \times D)$$

- A set of calculation laws

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \textsc{Comp-Assoc}$$
$$\pi_1 \circ (f \vartriangle g) = f \wedge \pi_2 \circ (f \vartriangle g) = g \qquad \textsc{Prod-Cancel}$$
$$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i \qquad \textsc{Prod-Absor}$$

Point-free Lenses

Lens



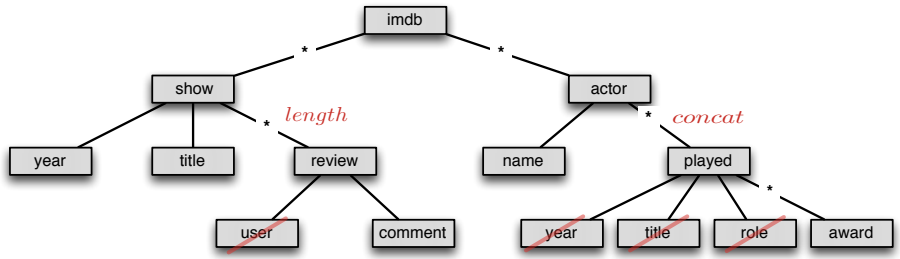
$$get : S \to V$$
$$put : V \times S \to S$$
$$get \circ put = \pi_1$$
$$put \circ (get \triangle id) = id$$

Language of point-free lens combinators

$$\begin{aligned}
Lens &::= id \mid Lens \circ Lens \mid !^c \mid Prod \mid Sum \mid Iso \mid Rec \\
Prod &::= \pi_1^b \mid \pi_2^a \mid Lens \times Lens \\
Sum &::= Lens \mathbin{\bullet\nabla} Lens \mid Lens \mathbin{\nabla_\bullet} Lens \mid Lens + Lens \\
&\quad \mid inl \nabla Lens \mid Lens \nabla inr \\
Iso &::= assocl \mid assocr \mid coassocl \mid coassocr \\
&\quad \mid swap \mid coswap \mid distl \mid distr \\
Rec &::= in_F \mid out_F \mid F\ Lens \mid (\!|Lens|\!)_F \mid [\!(Lens)\!]_F
\end{aligned}$$

Point-free Lens Examples

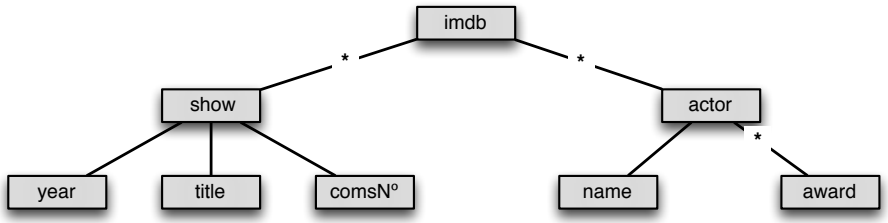
$$
\begin{aligned}
length^a &= [\![(id + \pi_2^a) \circ out]\!] &&: [A] \trianglerighteq Nat \\
map\ f &= (\!|in \circ (id + f \times id)|\!) &&: [A] \trianglerighteq [B] \\
concat &= (\!|\ldots|\!) &&: [[A]] \trianglerighteq [A]
\end{aligned}
$$



$$ex = map\ show \times map\ actor$$

$$show = id \times (id \times length \circ map\ (id \times \pi_{Comment}))$$

$$actor = id \times concat \circ map\ \pi_{Awards}$$



Point-free Lens Calculus

- lift the point-free laws to lenses

$$f = g \Leftrightarrow \begin{cases} get_f & = & get_g \\ put_f & = & put_g \end{cases}$$

- point-free lens laws

$$\pi_1^a \circ (f \times g) = f \circ \pi_1^{createf\ a} \qquad \textsc{Prod-Cancel}$$

$$(f \bullet\!\nabla\ g) \circ (h + i) = f \circ h \bullet\!\nabla\ g \circ i \qquad \textsc{Sum-Absor}$$

$$f \circ (\!|g|\!)_F = (\!|h|\!)_F \Leftarrow f \circ g = h \circ F\ f \qquad \textsc{Cata-Fusion}$$

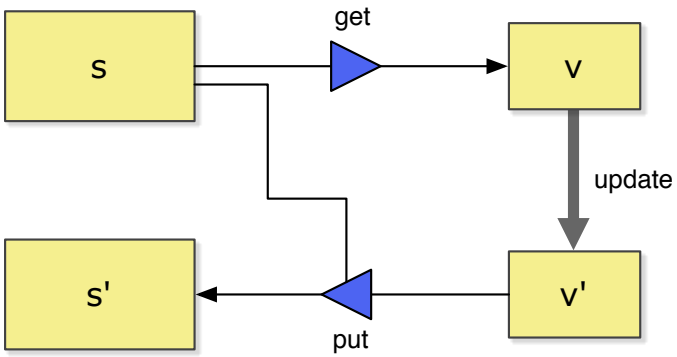
- fusion examples

$$length^a \circ map\ f = length^{createf\ a} \qquad \textsc{length-Map}$$

$$concat \circ map\ f = concatMap\ f \qquad \textsc{concat-Map}$$

Point-free Lens Calculus - Tupling

- put has redundant computations of get
- fuse *get* and *put* int a single function - Takeichi [IFP'09]



$$get : S \to V$$
$$put : S \to V \to S$$
$$get \triangle put : S \to V \times (V \to S)$$

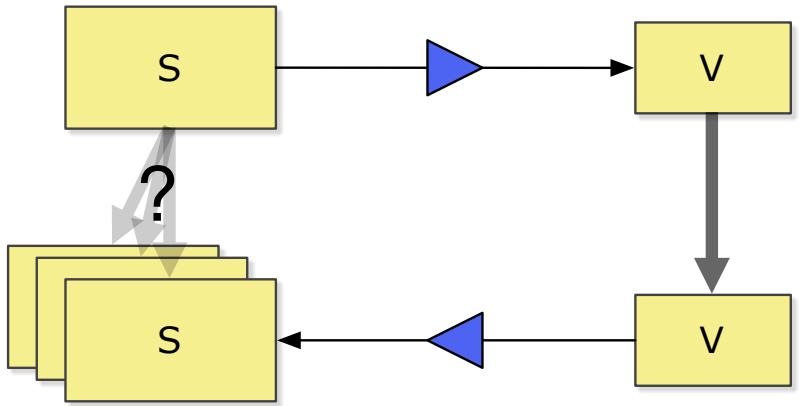
- Fokkinga's Mutu Tupling theorem

$$f \triangle g = (\!|\phi \triangle \psi|\!)_F \Leftarrow \begin{cases} f = \phi \circ F\ (f \triangle g) \circ out_F \\ g = \psi \circ F\ (f \triangle g) \circ out_F \end{cases}$$

2 Configurability

Motivation - Configurability

- for non-bijective transformations, an update may have many corresponding updates



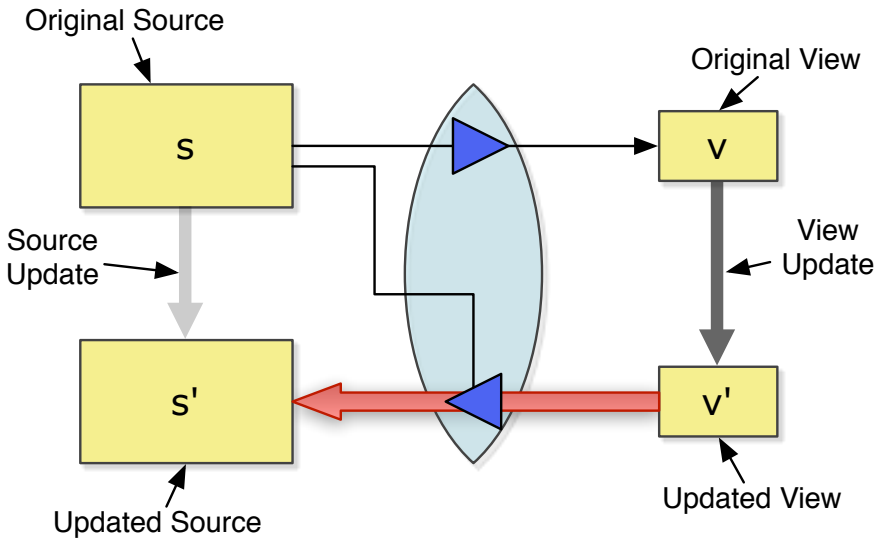
- our point-free lens language provides one possible update
- may not match the user's intentions!

Question

- how to allow users to choose a suitable update?

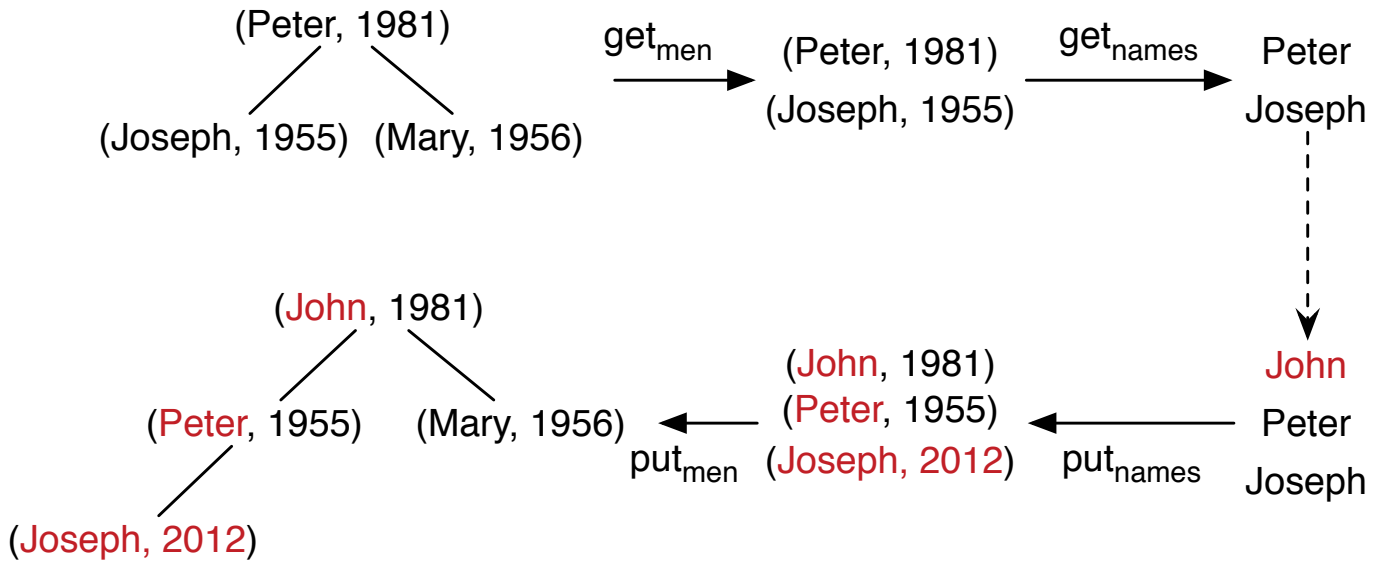
State-based Lenses

State-based framework: *put* takes the modified view state



- no information about the actual update
- *put* has to "guess" the intended change of the update

State-based Lens Example



data *Tree a* = *Empty* | *Node a* (*Tree a*) (*Tree a*)
type *Person* = (*Name*, *Birth*)

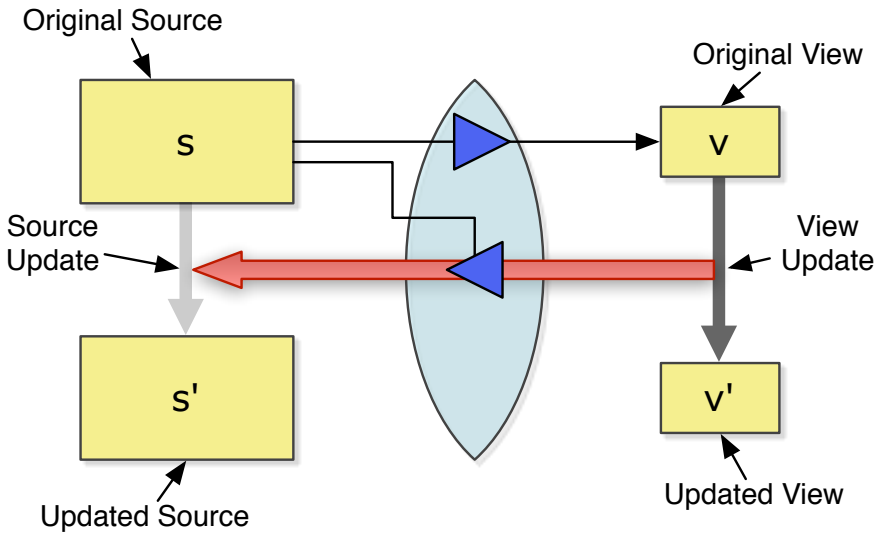
men : *Tree Person* ⩥ [*Person*]
men Empty = []
men (*Node p f m*) = *p* : *men f*

names : [*Person*] ⩥ [*Name*]
$names = map\ \pi_1^{const\ 2012}$

- positional behavior: birth years? Mary?

Operation-based Lenses

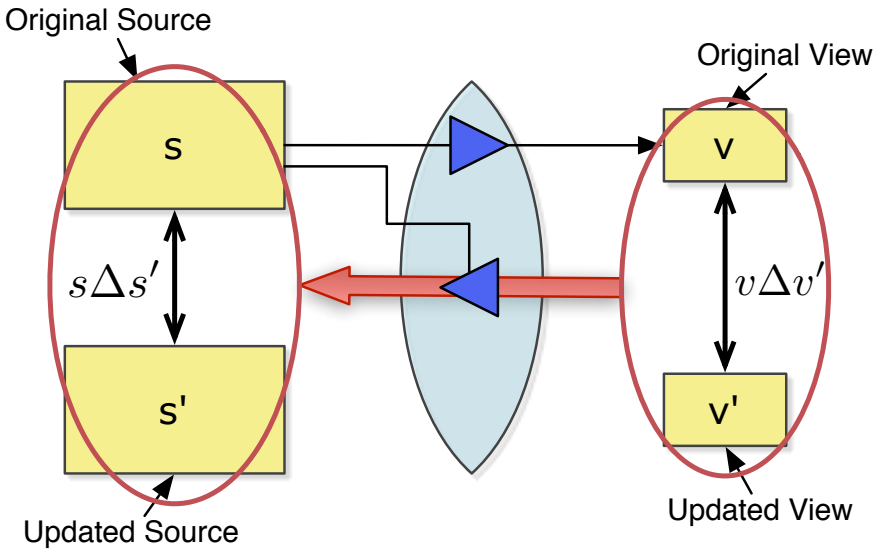
Operation-based framework: *put* takes a representation of the view update



- some knowledge about the actual update
- *put* can infer the intended change

Delta-based Lenses

Delta-based framework: *put* takes a delta

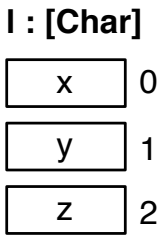


- Diskin et al. proposed an abstract framework of delta lenses [JOT]
- deltas model "change"
- delta = description of an update

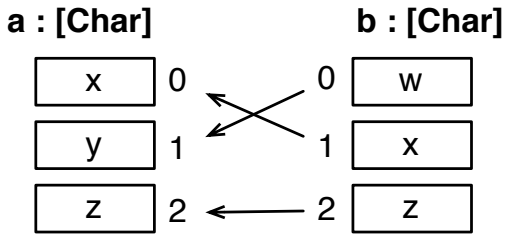
Point-free Delta Lenses

- we instantiate the abstract framework of Diskin et al.
- we introduce a notion of deltas for algebraic data types

1. decompose: shape + data



2. $a \Delta b$ = partial function

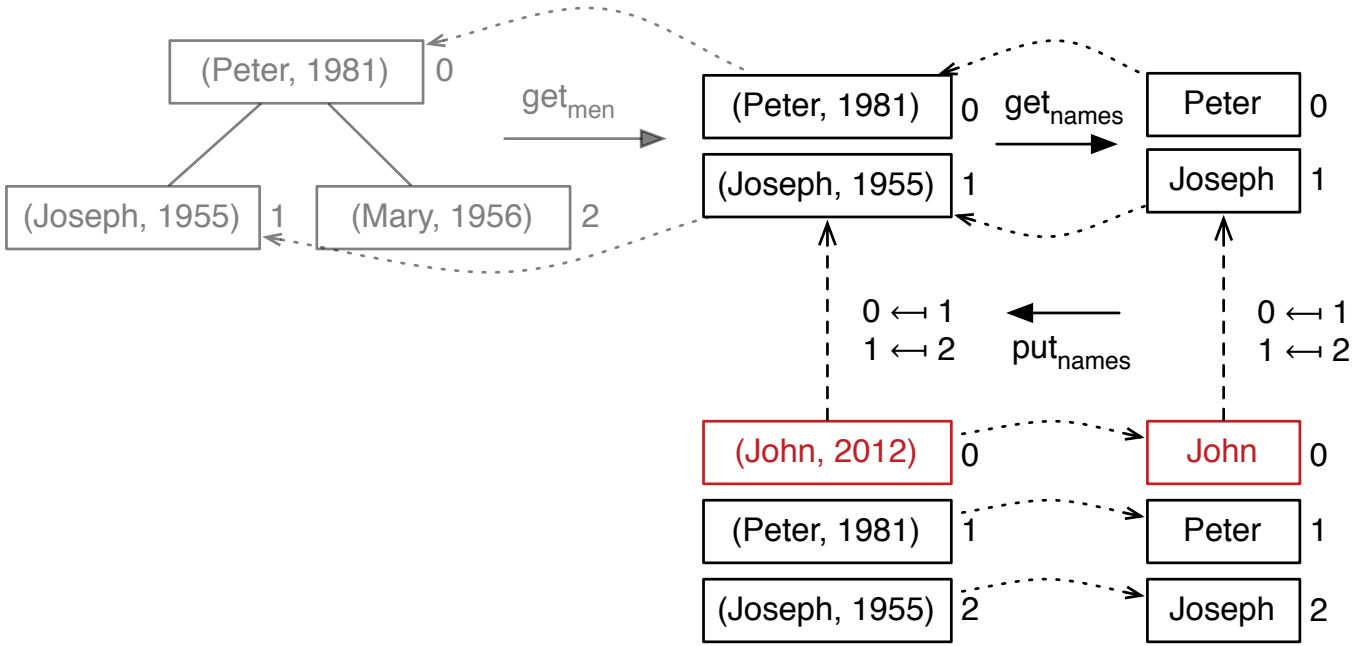


$$\textbf{data}\ [a] = [\,] \mid a : [a]$$

- we define a language of point-free delta lenses
- we tailor our previous lens language to consider deltas

Delta-based Example (Mapping)

- for mapping lenses, that preserve the shape

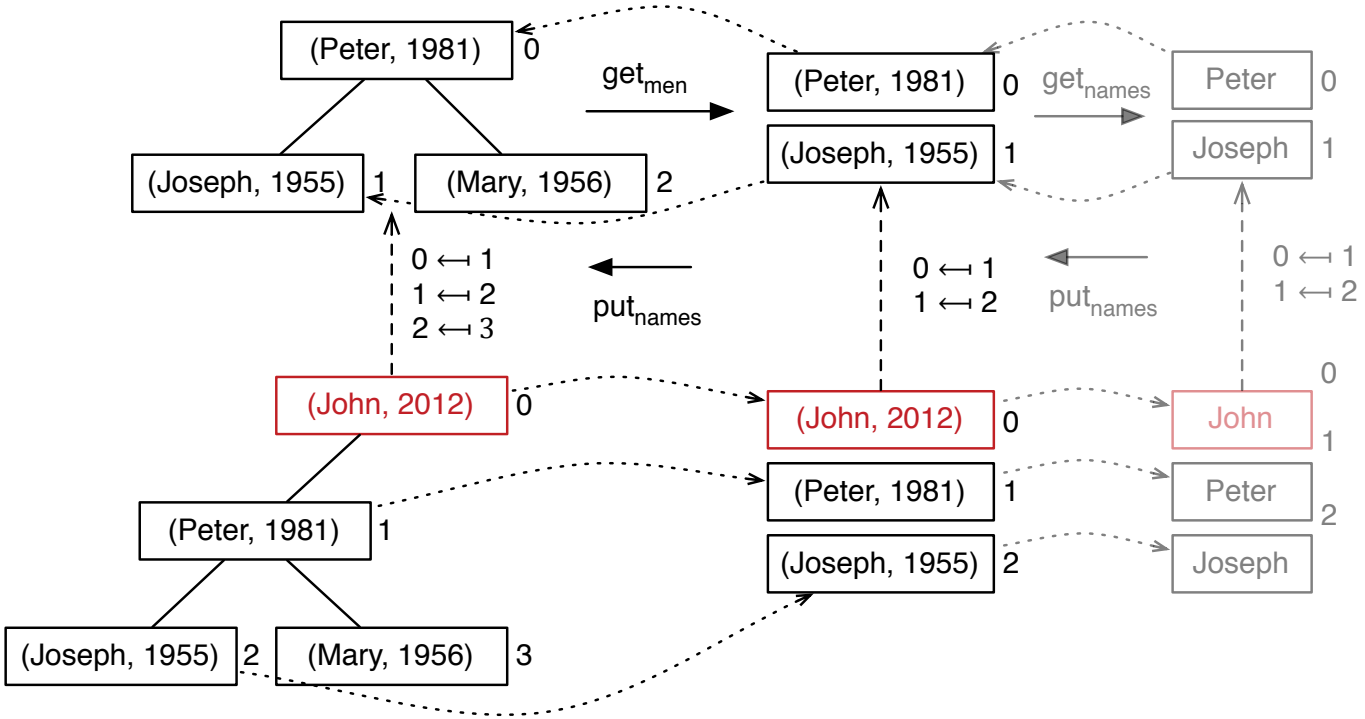


$$names : [Person] \trianglerighteq [Name]$$

- data alignment

Delta-based Example (Reshaping)

- for reshaping lenses, that preserve the data

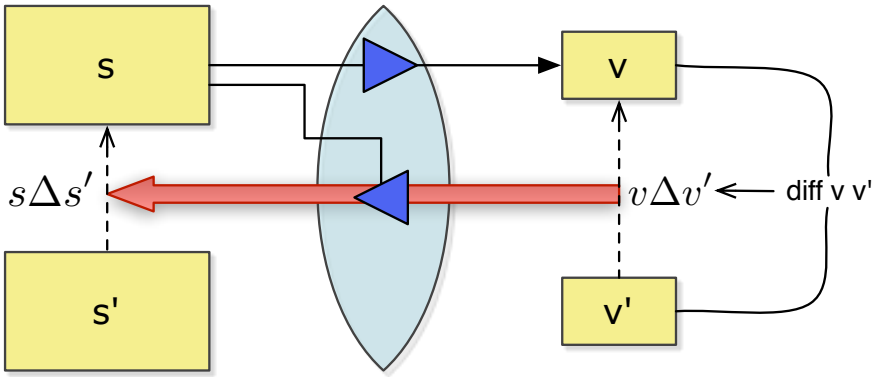


$$men : Tree\ Person \trianglerighteq [Person]$$

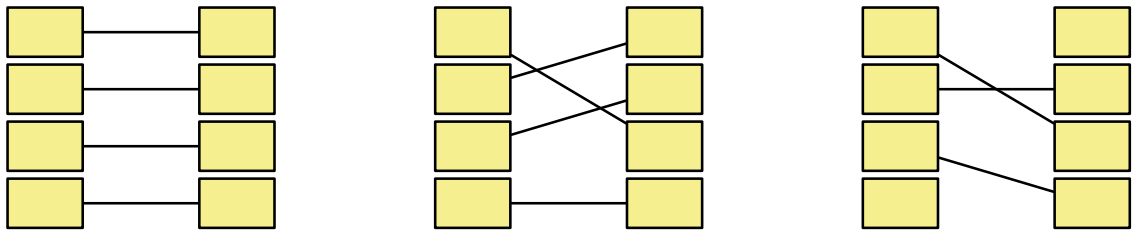
- shape alignment
  1. identify shape updates: insertions/deletions
  2. propagate shape updates: insertions/deletions

Delta Lens Configurability

- a delta can be calculated from the original and updated states



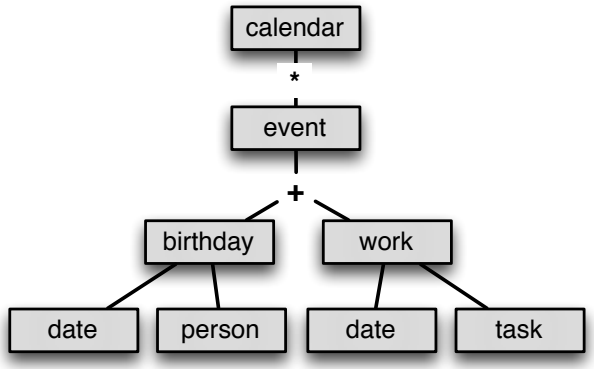
- user can choose an arbitrary heuristic



3 Genericity

Motivation - Conciseness

- bidirectional transformations are typically built to match a specific structure, via multiple steps
- collect all event dates in a calendar data format

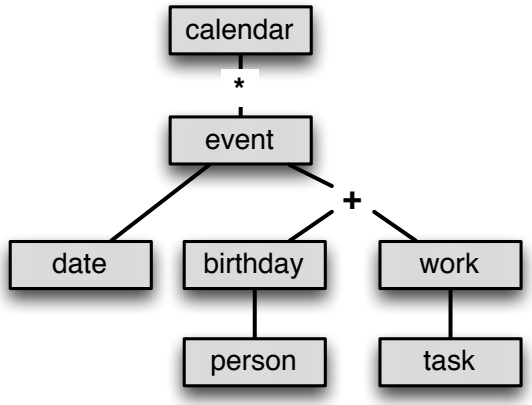


$$calendarDts : Calendar \rhd Date$$
$$calendarDts = map\ eventDts$$
$$eventDts = birthdayDts \nabla_{\bullet} workDts$$
$$birthdayDts = \pi_1^{const\ \texttt{"unknown"}}$$
$$workDts = \pi_1^{const\ \texttt{""}}$$

- impractical!
- for a real calendar format (iCal, XML, etc), it would be much more boring

Motivation - Reusability

- for the same high-level transformation, different BXs for different structures (collect all dates)

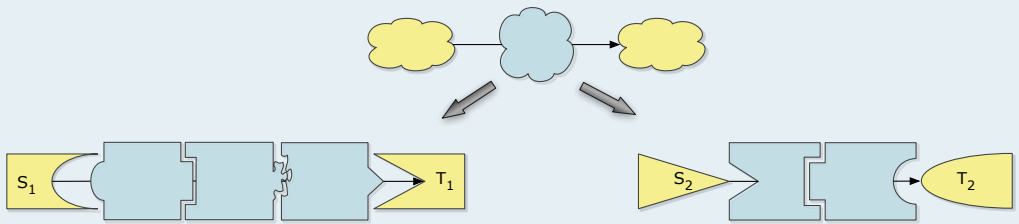


$$calendarDts' : Calendar' \trianglerighteq Date$$
$$calendarDts' = map\ eventsDts'$$
$$eventDts' = \pi_1^{const\ (Right\ (Work\ ""))}$$

- does not support evolution!

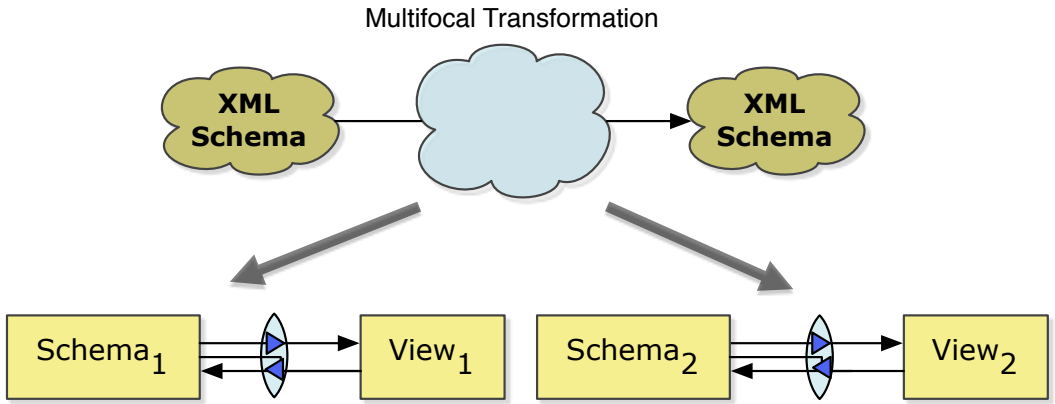
Question

- how to define a transformation in a concise and reusable way?



The *Multifocal* Language

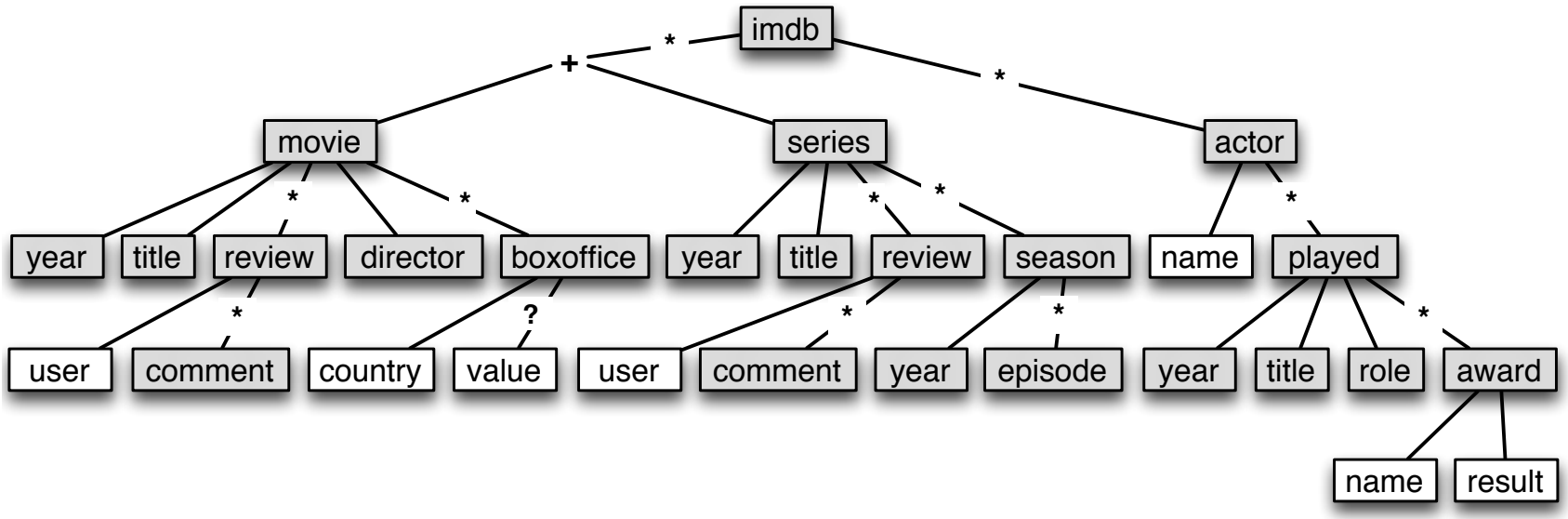
- we propose the *Multifocal* XML transformation language



- Two-level:
  1. schema-level transformations as views between XML Schemas
  2. model-level transformations as lenses between XML documents
- Strategic: concise specification style (e.g. traversals)
- Bidirectional: underlying document transformations as lenses

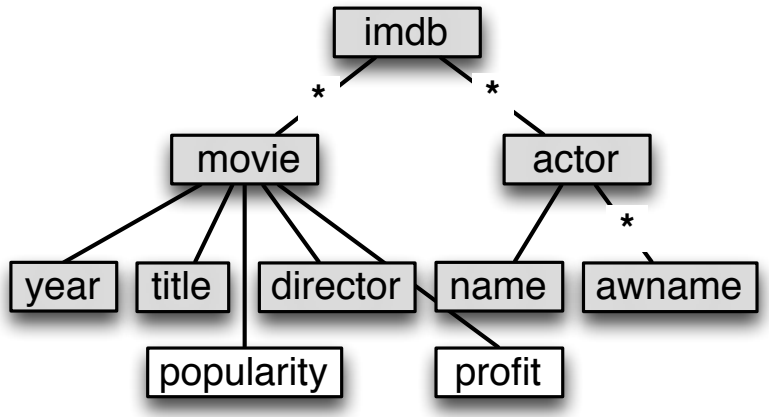
Multifocal Example: XML Views

- source XML Schema modeling a movie database



Multifocal Example: XML Views

- informal XML Schema transformation
  1. delete series
  2. for each movie:
     - count its popularity (total number of review comments)
     - estimate its profit (sum of the boxoffice values)
  3. for each actor, select its name and a list of award names
- view XML Schema



Multifocal Example: XML Views

```
<imdb>
 <movie>
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  <review user="emma">
   <comment>Gorgeous!</comment></review>
  <director>Quentin Tarantino</director>
  <boxoffice country="USA" value="22089322"/>
  <boxoffice country="Japan" value="3521628"/>
 </movie>
 <series><year>2011</year>
  <title>Game of Thrones</title>
  <season><year>2011</year>
   <episode>Winter is Coming</episode>
 </season></series>
 <actor name="Umma Thurman">
  <played><year>2003</year>
   <title>Kill Bill: Vol. 1</title>
   <role>The Bride</role>
   <award name="Saturn" result="Won"/>
 </played></actor>
</imdb>
```

Multifocal Example: XML Views

```
<imdb>
 <movie popularity="1" profit="25610950">
  <year>2003</year>
  <title>Kill Bill: Vol. 1</title>
  <director>Quentin Tarantino</director>
 </movie>
 <actor name="Umma Thurman">
  <awname>Saturn</awname>
 </actor>
</imdb>
```

Multifocal Example: XML Views

```
<imdb>
 <movie> ... </movie>
 <movie popularity="2" profit="15">
  <year>2012</year>
  <title>Sherlock Holmes: Game of Shadows</title>
  <director>Guy Ritchie</director>
 </movie>
 <actor name="Uma Thurman">
  <awname>Saturn Best Actress</awname>
 </actor>
</imdb>
```

Multifocal Example: XML Views

```
<imdb>
 <movie> ... </movie>
 <series> ... </series>
 <movie><year>2012</year>
  <title>Sherlock Holmes: Game of Shadows</title>
  <review user="" comment=""/>
  <review user="" comment=""/>
  <director>Guy Ritchie</director>
  <boxoffice country="" value="15"/>
 </movie>
 <actor name="Uma Thurman">
  <played><year>2003</year>
   <title>Kill Bill: Vol. 1</title>
   <role>The Bride</role>
   <award name="Saturn Best Actress" result="Won"/>
 </played></actor>
</imdb>
```

Multifocal Language: Basic Combinators

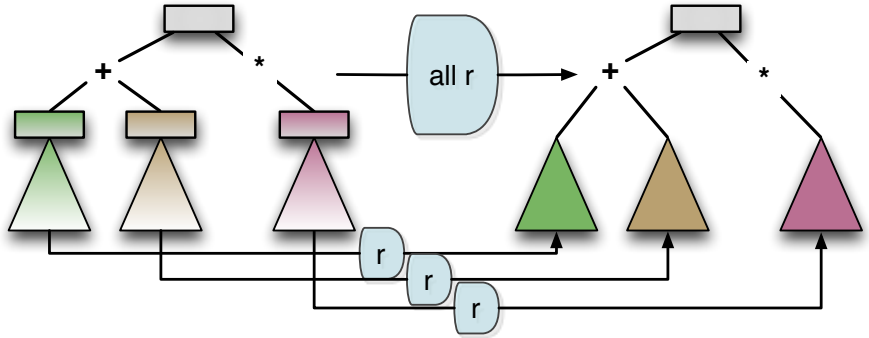
- generic style = concise specification
- strategic combinators

$$Strat = Schema \rightarrow Maybe\ (Schema, Lens)$$

- construct flexible strategies in a compositional way
- basic combinators (in what order? how often?):
  - identity $\texttt{nop} : Strat \rightarrow Strat$
  - sequentially $(>>) : Strat \rightarrow Strat \rightarrow Strat$
  - alternatively $(||) : Strat \rightarrow Strat \rightarrow Strat$
  - repetitively $\texttt{many} : Strat \rightarrow Strat$
  - optionally $\texttt{try} : Strat \rightarrow Strat$

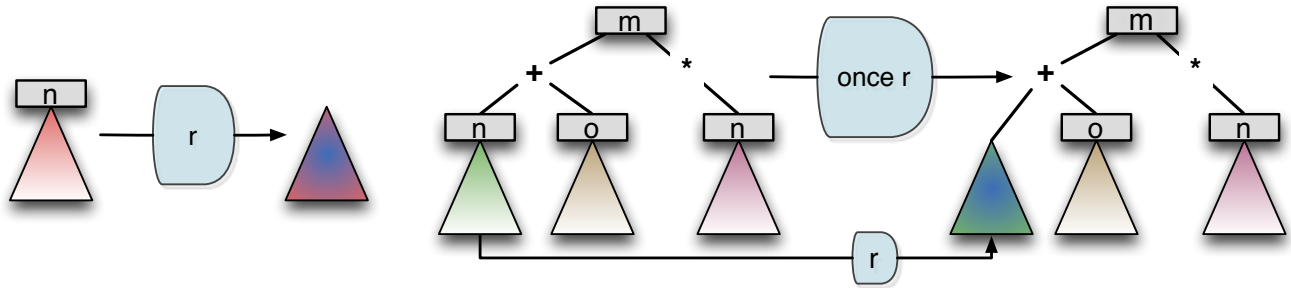
Multifocal Language: Traversal Combinators

- traversal combinators (at what depth?)
  - apply a strategy to all children



  - apply a strategy to all descendants

    `everywhere` : *Strat* → *Strat*
  - apply a strategy once at an arbitrary depth

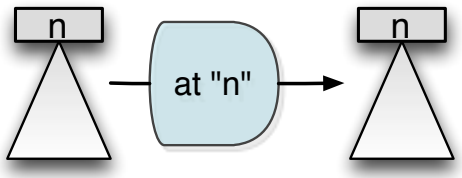


  - apply a strategy many times at an arbitrary depth

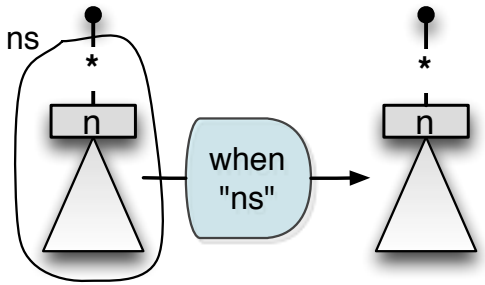
    `outermost` : *Strat* → *Strat*

Multifocal Language: Local Combinators

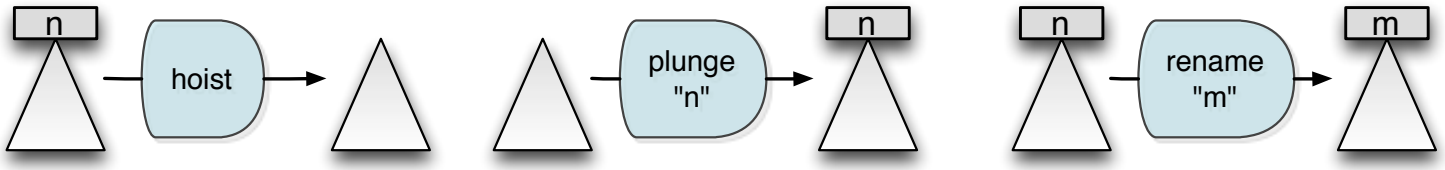
- control the application of certain strategies
- local combinators (under which conditions?)
  - at a particular element



  - at a particular location



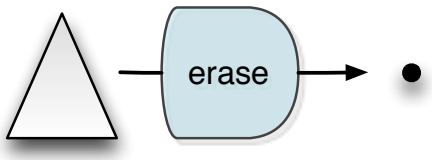
- XML name-based combinators



Multifocal Language: Abstraction Combinators

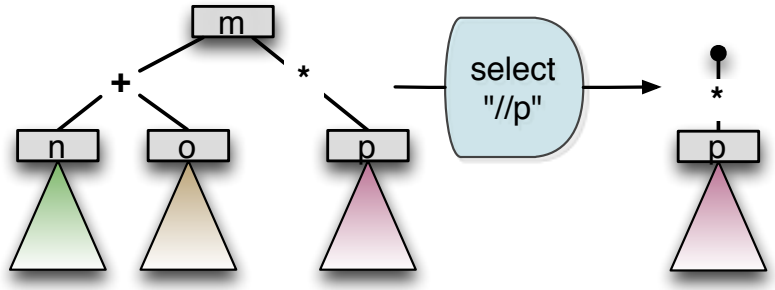
- language for defining XML views
- abstraction combinators (what to delete?)

- erase the current tree (explicit)



- empty tree

- apply an XPath query (implicit)



1. specialize the XPath expression ($/m\,/\,p$) for the source schema
2. convert it to a lens into the query's result type

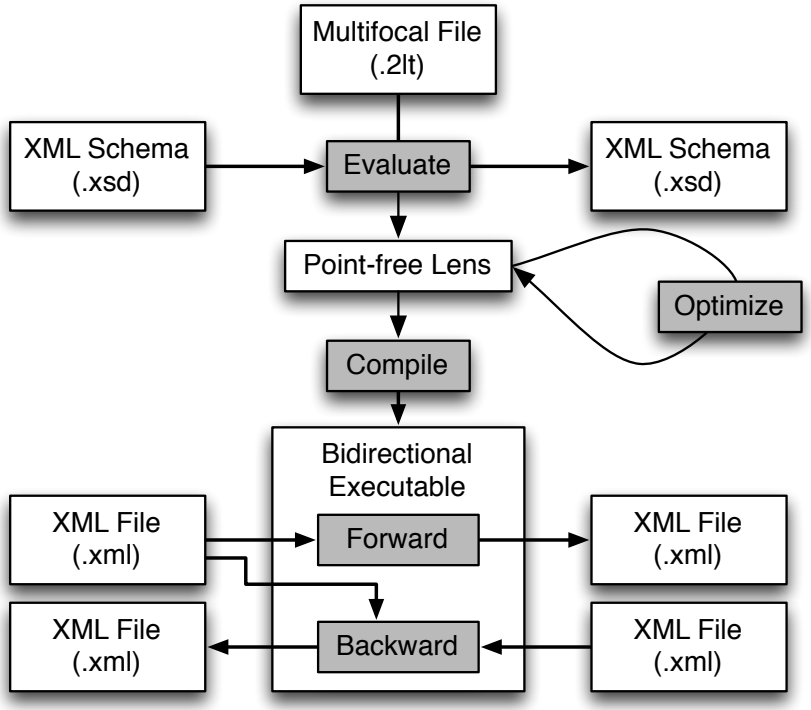
Multifocal Example: Strategic XML Views

1. delete series
2. for each movie:
   - count its popularity (total number of review comments)
   - estimate its profit (sum of the boxoffice values)
3. for each actor, select its name and a list of award names

1. ```
   everywhere (try (at "series" erase))
   ```
2. ```
   >> everywhere (try (at "movie" (
    outermost (when "reviews" (
     select "count(//comment)" >> plunge "@popularity"))
    >> outermost (when "boxoffices" (
     select "sum(//@value)" >> plunge "@profit")))))
   ```
3. ```
   >> everywhere (try (at "actor" (
    outermost (at "played" (
     select "award/@name" >> all (rename "awname"))))))
   ```
   ```

The *Multifocal* Framework

- we implement the *Multifocal* framework



- three stages:
  1. evaluate: XML Schema $\Rightarrow$ XML Schema + lens
  2. optimize (optional): lens $\Rightarrow$ optimized lens
  3. compile: (optimized) lens $\Rightarrow$ executable

Wrapping Up

1. efficiency
   - point-free lens language
   - algebraic calculus of lenses (fusion, tupling)
2. configurability
   - point-free delta lens language
   - user-provided alignment heuristics
3. genericity
   - *Multifocal* language and framework
   - a *Multifocal* XML schema transformation yields BXs in our lens language that can be optimized
   - reusable: multiple schemas
   - concise: strategic combinators

Libraries and Tools

- implemented in Haskell++
- available online

```
http://hackage.haskell.org/package/pointless-lenses
```

```
cabal install <package>
```

pointless-lenses (point-free lens library)
pointless-rewrite (point-free optimization library)
pointless-2lt (strategic two-level lens library)
multifocal (*Multifocal* system)

Further Reading


Hugo Pacheco and Alcino Cunha
Generic Point-free Lenses
MPC 2010.


Hugo Pacheco and Alcino Cunha
Calculating with lenses: optimising bidirectional transformations
PEPM 2011.


Hugo Pacheco, Alcino Cunha and Zhenjiang Hu
Delta Lenses over Inductive Types
BX 2012.


Hugo Pacheco and Alcino Cunha
Multifocal: A Strategic Bidirectional Transformation Language for XML Schemas
ICMT 2012.


Alcino Cunha and Hugo Pacheco
Algebraic Specialization of Generic Functions for Recursive Types
Electronic Notes in Theoretical Computer Science, 2011.