A Point-free Approach to Bidirectional Transformation

Hugo Pacheco

DI-CCTC, Universidade do Minho, Braga, Portugal

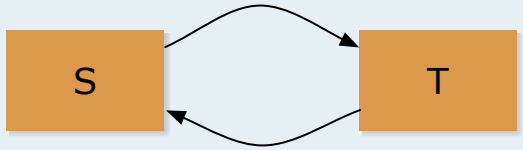
IPL Meeting

Tokyo - December 14th 2010

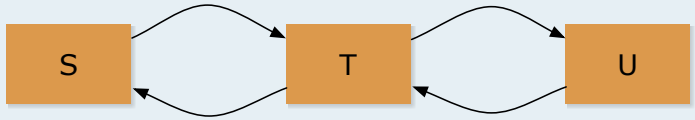
Bidirectional Transformations

Bidirectional languages

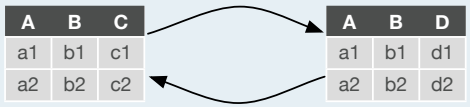
- derive two unidirectional transformations from a specification

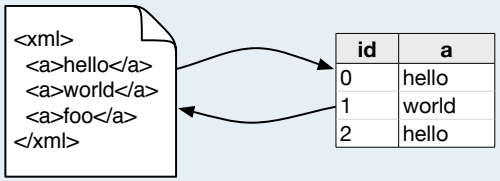


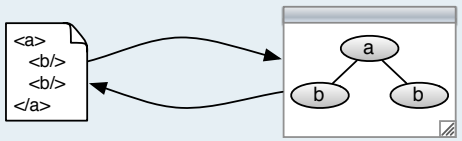
- clean semantics: consistency properties
- compositional

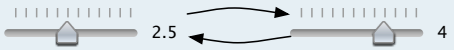


Bidirectional languages exist for...



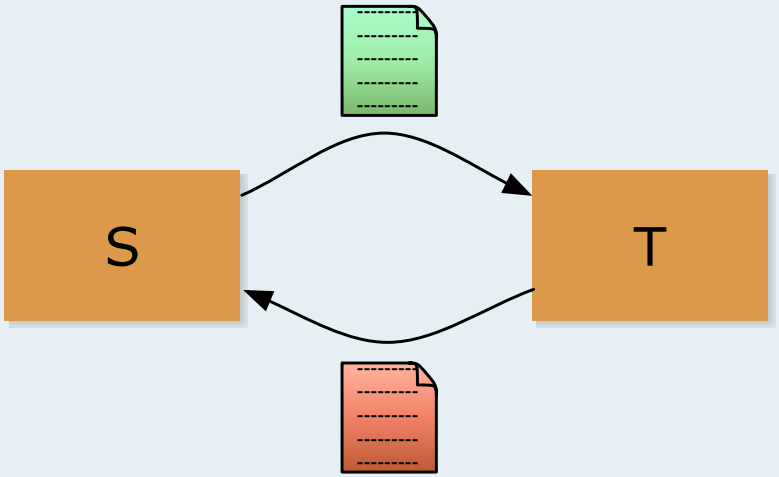






Motivation - Specification

- bidirectional behaviour is defined by sophisticated procedures



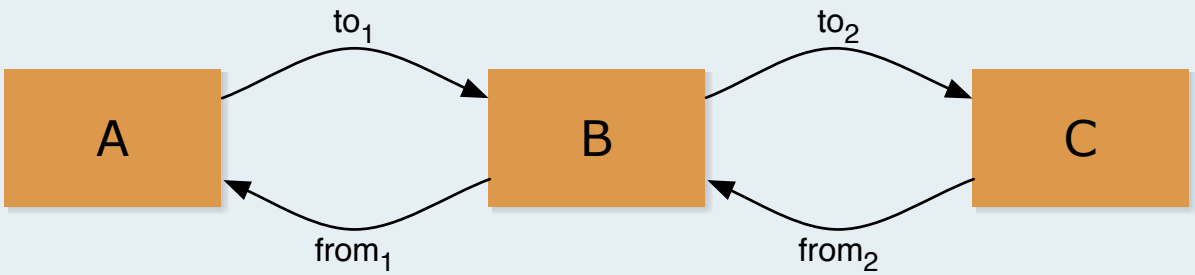
- hard to prove properties about the bidirectional transformations

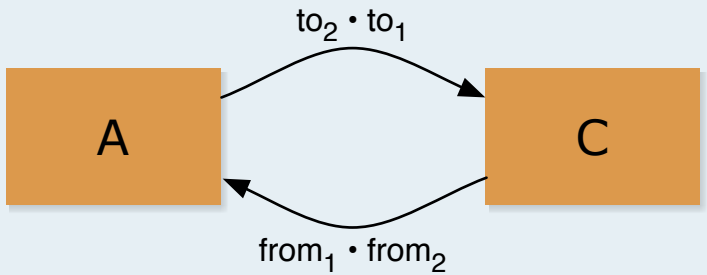
Goal

- a good framework to prove properties?

Motivation - Calculation

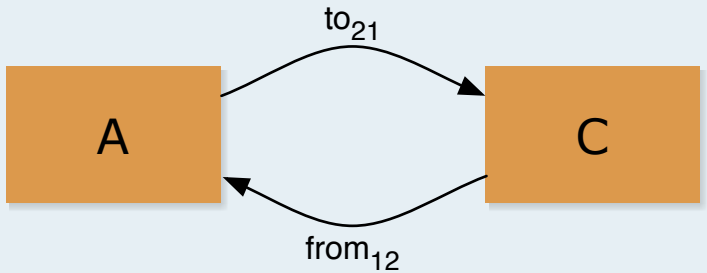
- compositionality = cluttering





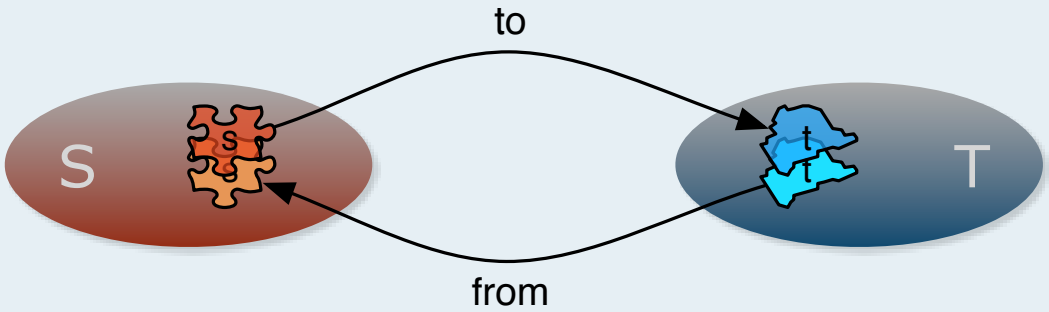
Goal

- how to calculate an optimised transformation?



Motivation - Totality

- non-total transformations



- partial laws

$$to \circ from = id$$

$$to \circ from \sqsubseteq id$$

- allow more expressive languages
- but may disallow desired updates

Goal

- need to control the partiality: which updates are valid?

A point-free design

- An application domain (Trees)

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$
$$\textbf{data}\ [a] = [] \mid a : [a]$$

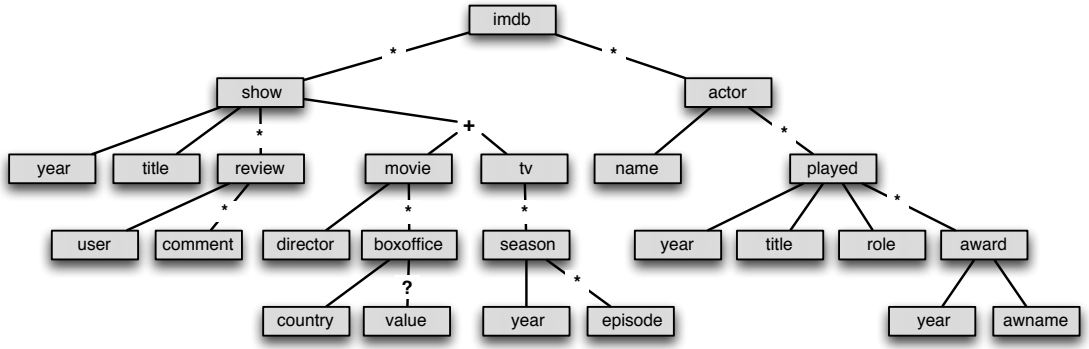
- A set for combinators

$$\begin{aligned} \mathit{id} &: A \to A \\ \circ &: (B \to C) \to (A \to B) \to (A \to C) \\ \pi_1 &: A \times B \to A \\ i_1 &: A \to A + B \\ \triangle &: (A \to B) \to (A \to C) \to (A \to B \times C) \end{aligned}$$

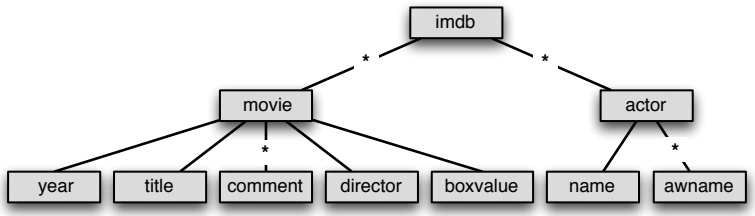
- A set of calculation/simplification laws

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\text{-Assoc}$$
$$\pi_1 \circ (f \triangle g) = f \wedge \pi_2 \circ (f \triangle g) = g \qquad \times\text{-Cancel}$$

An application scenario: XML querying



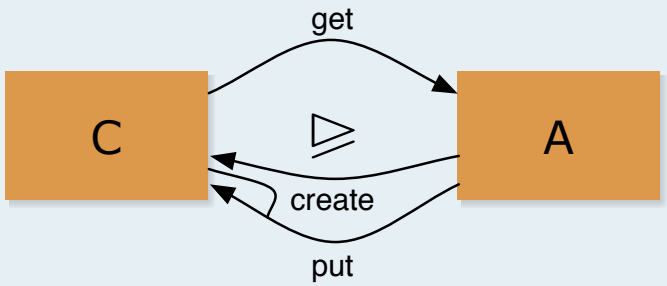
$$
\begin{aligned}
\mathit{imdb} &= \mathit{shows} \times \mathit{actors} \\
\mathit{shows} &= \mathit{map}\ ((\mathit{id} \times \mathit{reviews}) \times \mathit{id}) \circ \mathit{filter\_l} \\
&\quad \circ\ \mathit{map}\ \mathit{distr} \circ \mathit{map}\ (\mathit{id} \times (\mathit{movie} + \mathit{tv})) \\
\mathit{reviews} &= \mathit{length} \circ \mathit{concat} \circ \mathit{map}\ \pi_{\mathit{comments}} \\
\mathit{movie} &= \mathit{id} \times \mathit{boxoffices} \\
\mathit{boxoffices} &= \mathit{sum} \circ \mathit{filter\_r} \circ \mathit{map}\ \pi_{\mathit{value}} \\
\mathit{tv} &= \mathit{concat} \circ \mathit{map}\ \pi_{\mathit{episodes}} \\
\mathit{actors} &= \mathit{map}\ (\mathit{id} \times \mathit{awards}) \\
\mathit{awards} &= \mathit{map}\ \pi_{\mathit{awname}} \circ \mathit{concat} \circ \mathit{map}\ \pi_{\mathit{awards}}
\end{aligned}
$$



Data abstraction

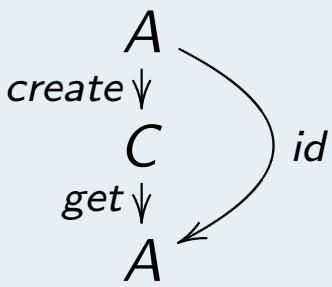
Lenses

$$
\begin{aligned}
get &: C \to A \\
create &: A \to C \\
put &: A \times C \to C
\end{aligned}
$$



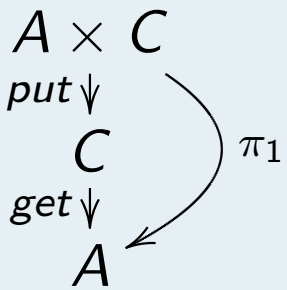
Properties for well-behaved lenses

- CreateGet



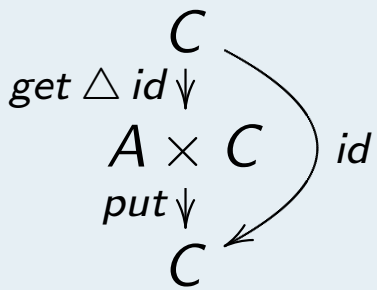
$get \circ create = id$

- PutGet



$get \circ put = \pi_1$

- GetPut



$put \circ (get \triangle id) = id$

A lens point-free design

- An application domain (Lenses over trees)

$$\begin{aligned}\textbf{data}\ c \trianglerighteq a = Lens\ \{ & get \quad :: c \to a \\ , & create :: a \to c \\ , & put \quad :: (a, c) \to c\}\end{aligned}$$

- A set for combinators

$$\begin{aligned} id &: A \trianglerighteq A \\ \circ &: (B \trianglerighteq C) \to (A \trianglerighteq B) \to (A \trianglerighteq C) \\ \pi_1 &: A \times B \trianglerighteq A \\ \times &: (A \trianglerighteq C) \to (B \trianglerighteq D) \to (A \times B \trianglerighteq C \times D) \end{aligned}$$

- A set of calculation/simplification laws

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\text{-Assoc}$$

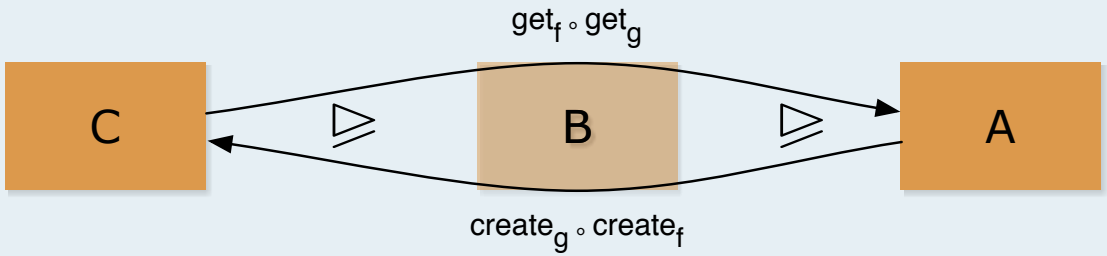
$$\pi_1 \circ (f \times g) = f \circ \pi_1 \wedge \pi_2 \circ (f \times g) = g \circ \pi_2 \qquad \times\text{-Cancel}$$

Composition as a lens

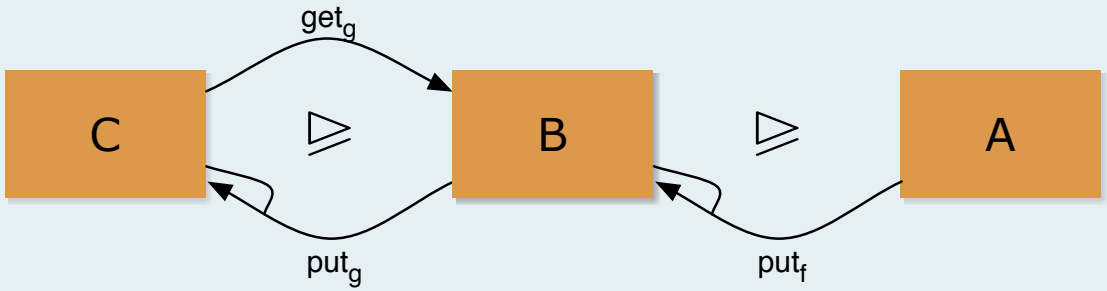
Lens composition

$$\forall f : B \trianglerighteq A,\ g : C \trianglerighteq B.\ \ f \circ g : C \trianglerighteq A$$

$get = get_f \circ get_g$ $\qquad$ $create = create_g \circ create_f$



$$put = put_g \circ (put_f \circ (id \times get_g) \triangle \pi_2) : A \times C \to C$$



$$id \circ f = f = f \circ id \qquad \textsc{Id-Nat}$$

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\textsc{-Assoc}$$

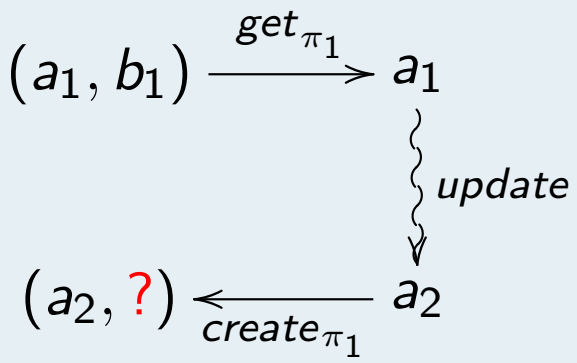
Products - Projection lens

Drop element

$$\forall f : A \to B.\ {\pi_1}^f : A \times B \trianglerighteq A$$

$$\begin{aligned} get &: A \times B \to A \\ get &= \pi_1 \\ create &: A \to A \times B \\ create &= id \triangle f \\ put &: A \times (A \times B) \to A \times B \\ put &= id \times \pi_2 \end{aligned}$$

- Choice of create



Properties

$$\begin{aligned} get \circ create &= \pi_1 \circ (id \triangle f) &&= id \\ get \circ put &= \pi_1 \circ (id \times \pi_2) &&= \pi_1 \\ put \circ (get \triangle id) &= (id \times \pi_2) \circ (\pi_1 \triangle id) &&= \pi_1 \triangle \pi_2 = id \end{aligned}$$

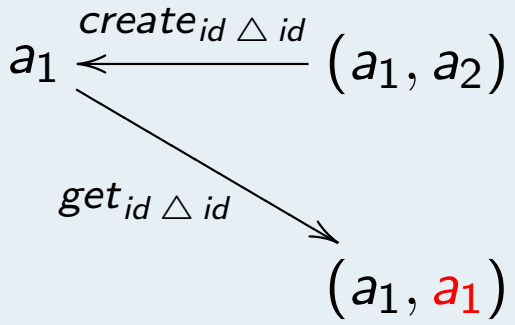
Products - Splitting

Duplicate element

$$\forall f : A \trianglerighteq B, g : A \trianglerighteq C.\ f \triangle g : A \trianglerighteq B \times C$$

$$\begin{aligned} get &: A \to B \times C \\ get &= get_f \triangle get_g \\ create &: B \times C \to A \\ create &= create_f \circ \pi_1 \ \vee \\ &\quad\ create_g \circ \pi_2 \ \vee \ \ldots \\ put &= create \circ \pi_1 \end{aligned}$$

- Bad create



- $f \triangle g$ is not a robust abstraction $\Leftarrow$ duplication

Partial lens & properties

$$get \circ create \sqsubseteq id \qquad get \circ put \sqsubseteq id$$
$$put \circ (get \triangle id) = id$$

Products - More lenses

Product

$$\forall f : A \trianglerighteq C, g : B \trianglerighteq D.\ f \times g : A \times B \trianglerighteq C \times D$$

$$swap : A \times B \trianglerighteq B \times A$$

$$assoc : A \times (B \times C) \trianglerighteq (A \times B) \times C$$

Some laws

$$id \times id = id \qquad \times\text{-Functor-Id}$$

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-Functor-Comp}$$

$${\pi_1}^h \circ (f \times g) = f \circ {\pi_1}^{create_g \circ h \circ get_f} \qquad \pi_1\text{-Nat}$$

$$swap \circ (f \times g) = (g \times f) \circ swap \qquad swap\text{-Nat}$$

$${\pi_1}^f \circ swap = {\pi_2}^f \qquad swap\text{-Cancel}$$

Sums - Either lens

Injections

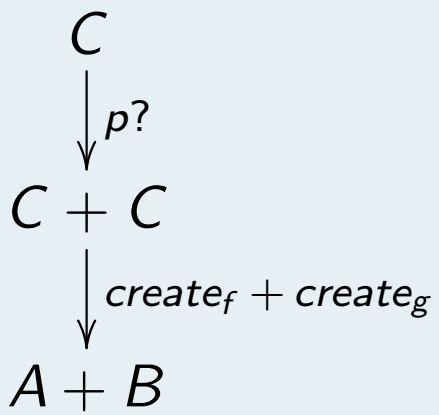
- $i_1 : A \to A + B$ and $i_2 : B \to A + B$ are not perfect abstractions $\Leftarrow$ insert conditional information

"Conditional" choice

$$\forall p : C \to 2, f : A \trianglerighteq C, g : B \trianglerighteq C.\ (f \nabla g)^p : A + B \trianglerighteq C$$

$$\begin{aligned} get &: A + B \to C \\ get &= get_f \nabla get_g \\ create &: C \to A + B \\ create &= (create_f + create_g) \circ p\,? \\ put &: C \times (A + B) \to A + B \\ put &= (put_f + put_g) \circ distr \end{aligned}$$

- Choice of create



Sums - More lenses

Sum combinator

$$\forall f : A \trianglerighteq C, g : B \trianglerighteq D.\ \circ (f + g)^{h,i} : A + B \trianglerighteq C + D$$

- Choice of put

$$\begin{array}{c} (C + D) \times (A + B) \\ \downarrow dists \\ C \times A \ + \ C \times B \ + \ D \times A \ + \ D \times B \\ \downarrow put_f \quad \downarrow \ldots \quad \downarrow \ldots \quad \downarrow put_g \\ \ldots \quad \ldots \quad \ldots \quad \ldots \end{array}$$

Some laws

$$f \circ (g \nabla h)^p = (f \circ g \nabla f \circ h)^{p \circ create_f} \qquad +\text{-Fusion}$$

$$(f \nabla g)^p \circ (h + i)^{j,k} = (f \circ h \nabla g \circ i)^p \qquad +\text{-Absor}$$

$$(id + id)^{f,g} = id \qquad +\text{-Functor-Id}$$

$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad +\text{-Functor-Comp}$$

$$(f + g)^{j,k} \circ (h + i)^{l,m} = (f \circ h + g \circ i)^{o,p} \Leftrightarrow \ldots \qquad +\text{-Comp}$$

Recursive lenses

Some recursive lenses

$$length : [A] \trianglerighteq \mathbb{N} \qquad map : (A \trianglerighteq B) \to [A] \trianglerighteq [B]$$
$$get\ [\,] = 0 \qquad get\ f\ [\,] = [\,]$$
$$get\ (x : xs) = (get\ xs) + 1 \qquad get\ f\ (x : xs) = get_f\ x : get\ xs$$

- How can we bidirectionalise them?

Recursive point-free combinators

$$in_F : F\ \mu F \trianglerighteq \mu F \qquad \forall f : F\ A \trianglerighteq A.\ (\!|f|\!)_F : \mu F \trianglerighteq A$$
$$out_F : \mu F \trianglerighteq F\ \mu F \qquad \forall f : A \trianglerighteq F\ A.\ [\!(f)\!]_F : A \trianglerighteq \mu F$$
$$[A] \simeq \mu L_A \qquad L_A\ [A] = 1 + A \times [A]$$

"Lensified" examples

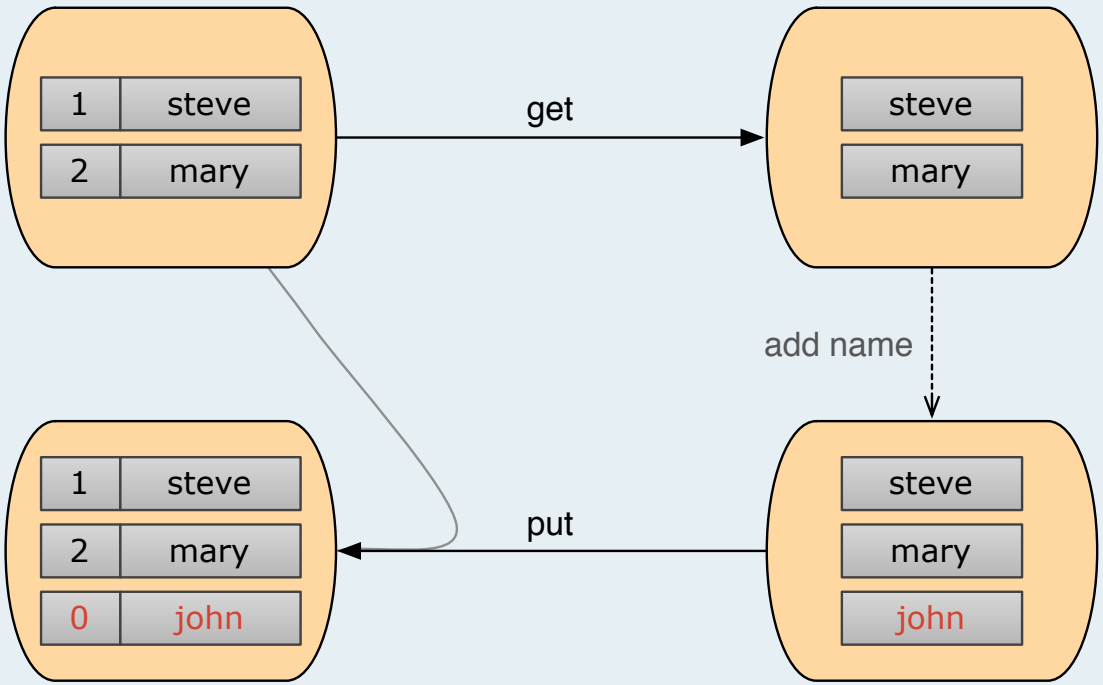
$$length^A : [A] \trianglerighteq \mathbb{N} \qquad map : (A \trianglerighteq B) \to [A] \trianglerighteq [B]$$
$$length^a = (\!|in_N \circ (id + \pi_2{}^{const\ a})|\!)_L \qquad map\ f = (\!|in_L \circ (id + f \times id)|\!)_L$$

- Good: reuse existing point-free combinators! not primitives!

Recursive lenses - Example

$$put_{map\ f} :: ([b], [a]) \to [a]$$
$$put_{map\ f}\ ([\,], as) = [\,]$$
$$put_{map\ f}\ (b : bs, [\,]) = create_f\ b : put_{map\ f}\ (bs, [\,])$$
$$put_{map\ f}\ (b : bs, a : as) = put_f\ (b, a) : put_{map\ f}\ (bs, as)$$

- $map\ \pi_2{}^{const\ 0} : [ID \times Name] \trianglerighteq [Name]$



Recursive lenses - Laws

Some recursive laws

$$in_F \circ out_F = id \ \wedge \ out_F \circ in_F = id \qquad \textit{in-out}\text{-Iso}$$

$$(\![in_F]\!)_F = id \qquad (\![\cdot]\!)\text{-Reflex}$$

$$(\![f]\!)_F \circ in_F = f \circ F\ (\![f]\!)_F \qquad (\![\cdot]\!)\text{-Cancel}$$

$$f \circ (\![g]\!)_F = (\![h]\!)_F \Leftarrow f \circ g = h \circ F\ f \qquad (\![\cdot]\!)\text{-Fusion}$$

Some derived laws for lists

$$map\ id = id \qquad map\text{-Id}$$

$$map\ f \circ map\ g = map\ (f \circ g) \qquad map\text{-Fusion}$$

$$filter\_l \circ map\ (f + g)^{h,i} = map\ f \circ filter\_l \qquad filter\_l\text{-Map}$$

$$length^v \circ map\ f = length^{create_f\ v} \qquad length\text{-Map}$$

Summary (Lens language)

Grammar for lens combinators

$$
\begin{aligned}
\mathit{Lens} &::= \mathit{id} \mid \mathit{Lens} \circ \mathit{Lens} \mid !^{f} \mid \mathit{Prod} \mid \mathit{Sum} \mid \mathit{Iso} \mid \mathit{Rec} \\
\mathit{Prod} &::= {\pi_1}^{f} \mid {\pi_2}^{f} \mid \mathit{Lens} \times \mathit{Lens} \\
\mathit{Sum} &::= (\mathit{Lens} \nabla \mathit{Lens})^{p} \mid (\mathit{Lens} + \mathit{Lens})^{f,g} \\
\mathit{Iso} &::= \mathit{assoc} \mid \mathit{assoc}^{-1} \mid \mathit{coassoc} \mid \mathit{coassoc}^{-1} \\
&\quad \mid \mathit{swap} \mid \mathit{coswap} \mid \mathit{distl} \mid \mathit{distr} \\
\mathit{Rec} &::= \mathit{in}_F \mid \mathit{out}_F \mid F\ \cdot \mid (\!|\cdot|\!)_F \mid [\!(\cdot)\!]_F
\end{aligned}
$$

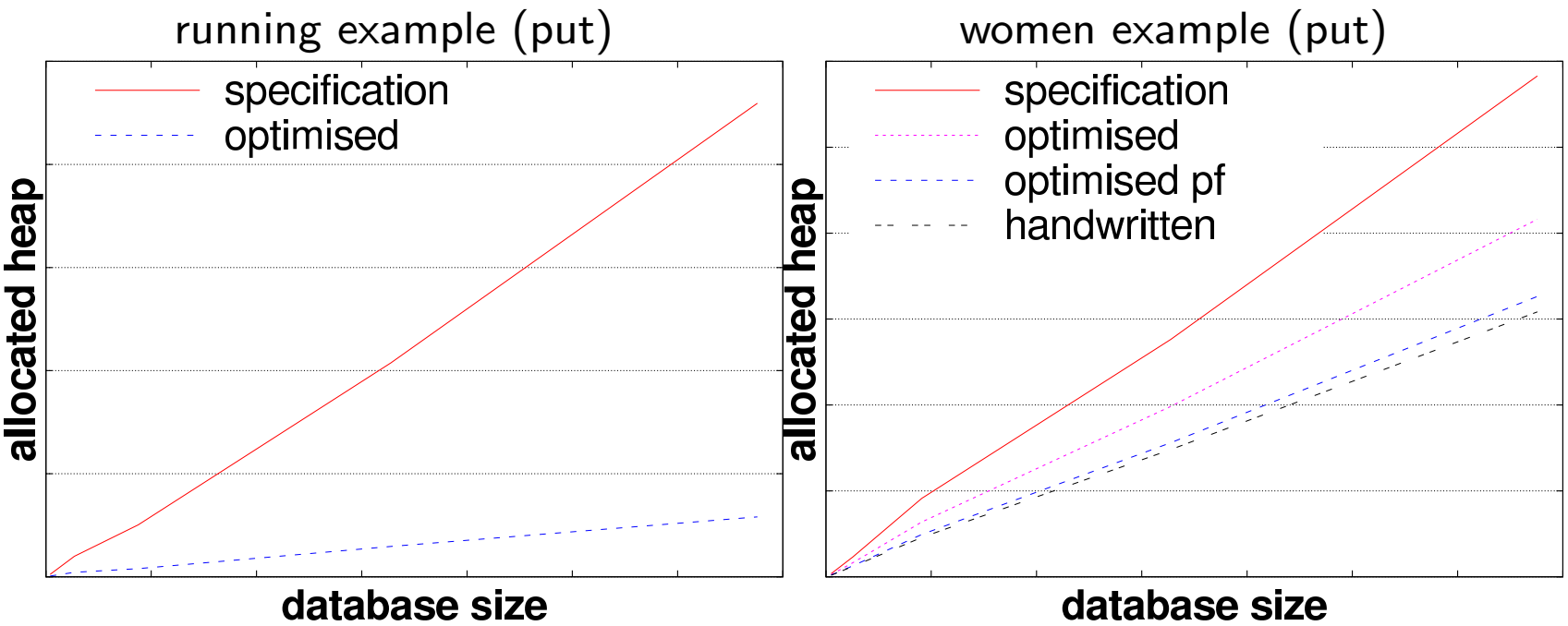
Notable exceptions

$$
\begin{aligned}
\mathit{NonLens} ::= {} & i_1 : A \to A + B \mid i_2 : B \to A + B \\
& \mid \underline{\cdot} : 1 \to B \qquad \mid ? : (A \to 2) \to (A \to A + A) \\
& \mid \cdot \triangle \cdot : (A \to B) \to (A \to C) \to (A \to B \times C)
\end{aligned}
$$

Optimisation

- calculational laws $\Rightarrow$ automated optimisation tool
- what about an handwritten definition?
- simpler example

$$\textbf{type}\ Person = (Name, Gender)\ \textbf{data}\ Gender = M \mid F$$
$$women\ :\ [Person] \trianglerighteq \mathbb{N}$$
$$women = length \circ filter\_r \circ map\ (out_G \circ \pi_{Gender})$$



References

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce and A. Schmitt
Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem.
ACM Transactions on Programming Languages and Systems, 2007.

Hugo Pacheco and Alcino Cunha
Generic Point-free Lenses.
Mathematics of Program Construction, 2010.

Hugo Pacheco and Alcino Cunha
Calculating with Lenses: Optimising Bidirectional Transformations
Partial Evaluation and Program Manipulation, 2011.

Alcino Cunha and Hugo Pacheco
Algebraic Specialization of Generic Functions for Recursive Types.
Mathematically Structured Functional Programming, 2008.

Demos: Haskell++

- `http://hackage.haskell.org` ⇒ `pointless-lenses`
- `http://hackage.haskell.org` ⇒ `pointless-rewrite`

Conclusions

Pros

+ Bidirectional language from standard point-free combinators
+ Clear bidirectionalisation: straightforward proofs
+ Support for recursive lenses (w/ termination conditions)
+ Lens bidirectional calculus: reasoning and optimisation of complex transformations
+ (Some) choice of backward transformation

Cons

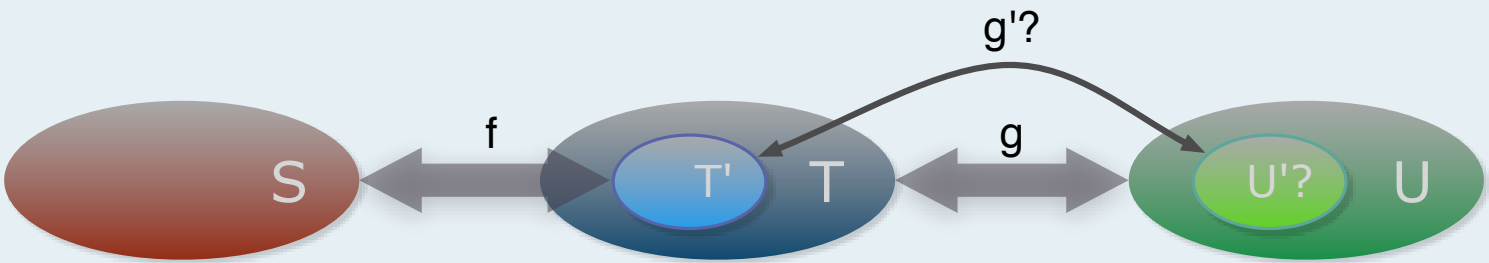
– Expressiveness (perfect abstractions for the sake of totality)
– Opaque isomorphism combinators
– How much choice?

Currently under work...

Motivation - Expressiveness / Partiality

- no full products and sums - too restrictive
- partial lenses ($\triangle$, i_1, i_2, $\underline{\cdot}$, ?)

- "ill-behaved" composition



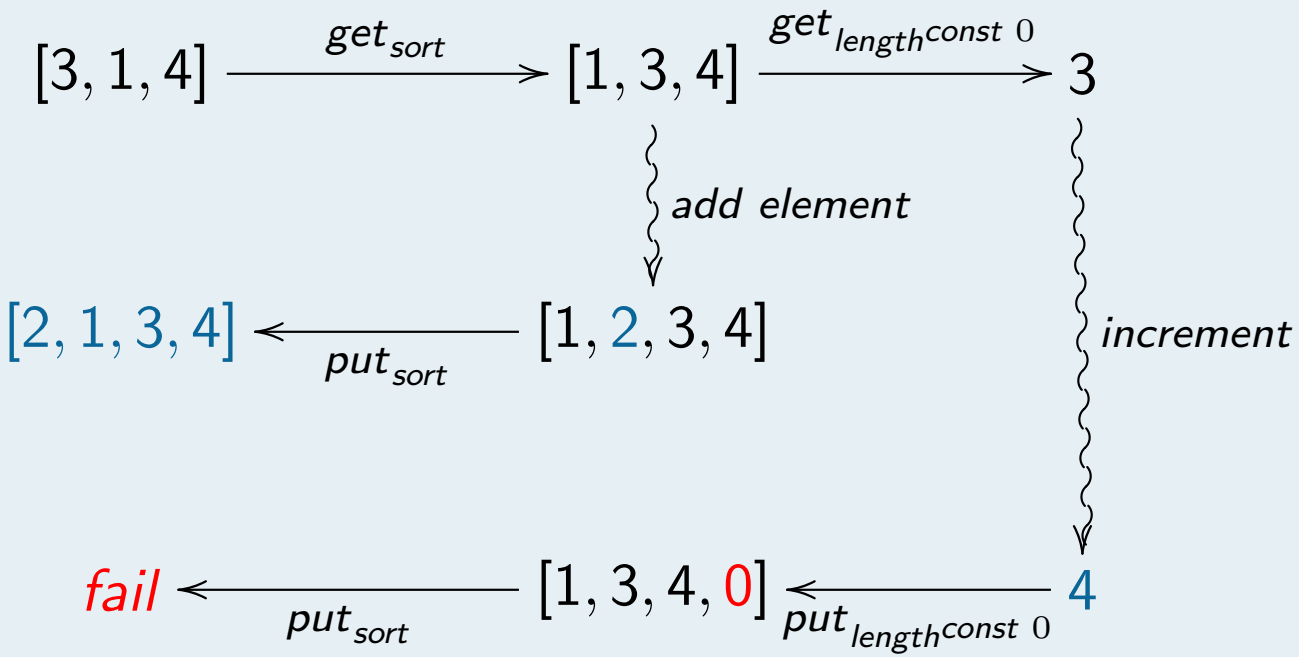
Goal

- how to compute the correct restrictions on the backward transformation and the lens domains?

A partial lens example

- insertion sort:

$$sort : [A] \trianglerighteq [A]_{\phi_{sort}=\{\ [a_1,a_2] \in [A] \mid a_1 \leqslant a_2\ \}}$$
$$sort = (\!|nil \nabla insert|\!)_L$$
$$insert : A \times [A]_{\phi_{sort}} \trianglerighteq [A]_{\phi_{sort}}$$



Idea - Go relational...

Relational lenses

- backward transformation as relational converse:

$$id \triangle id : A \trianglerighteq^{R} A \times A$$

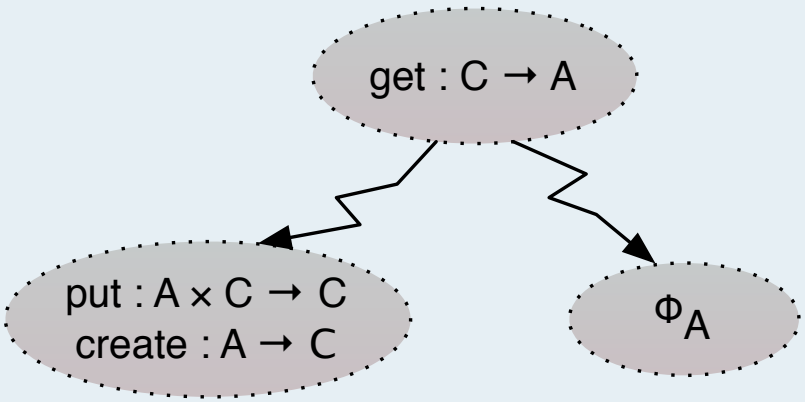
$$create_{id \triangle id} = (id \triangle id)^{\circ}$$

- functional refinement (no duplication):

$$(\pi_1 \triangle \pi_2) : A \times B \trianglerighteq A \times B$$

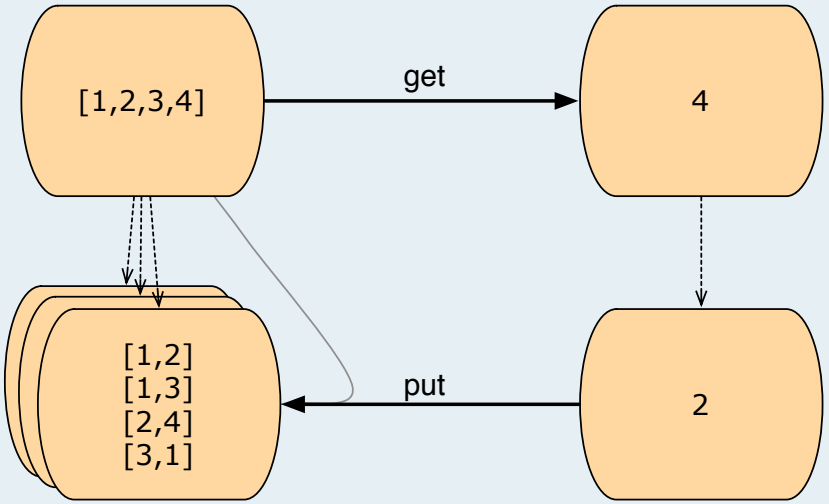
$$id \triangle id : A \trianglerighteq (A \times A)_{\phi_{dup} = \{ (a_1, a_2) \in A \times A \mid a_1 = a_2 \}}$$

Total lenses (again)



Motivation - Determinism

- non-determinism: multiple *put*s



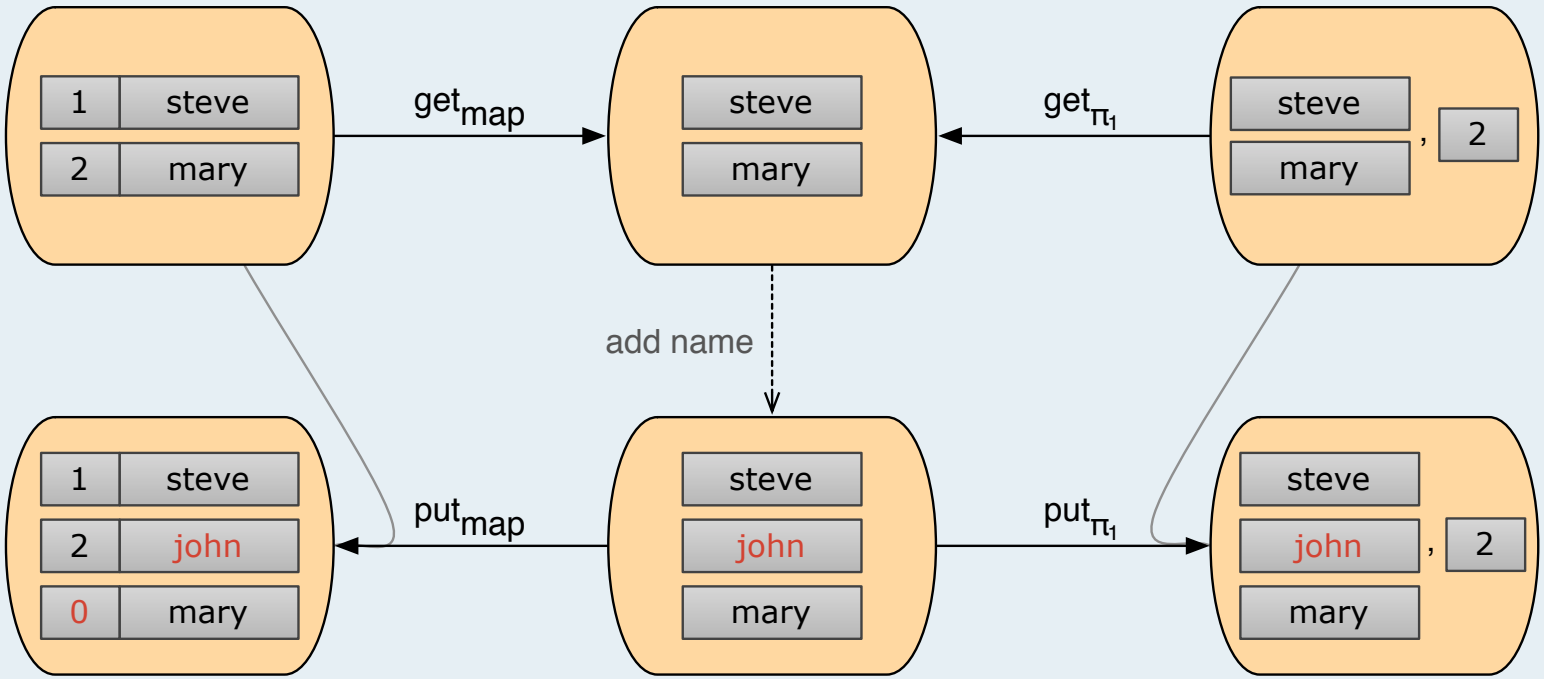
- is there a best one?

Goal

- how to allow users to easily control *put* without disrupting the lens properties?

An unsatisfactory example

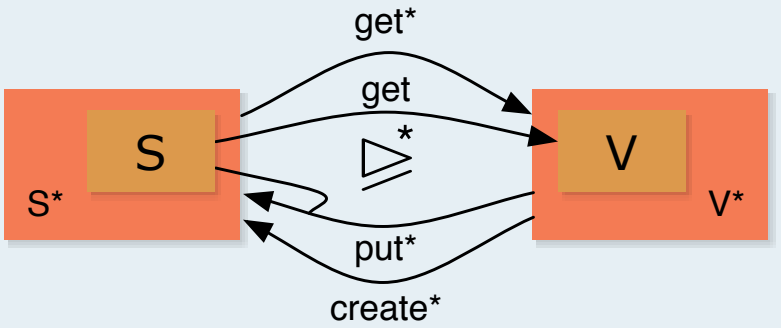
- alignment: *map* operates positionally
- source "side-effects": π_1 recovers part of the original source



Idea - Open operation-based lenses

- extend the view (and source) with editing tags

$$get^* \circ create^* = id$$
$$get^* \circ put^* = \pi_1$$
$$put^* \circ (get \triangle id) = id$$



- users can annotate the view (whichever tags)

$$diff : V \times V \to V^*$$

- users can modify the source during put^* (if the view has tags)

$$reflect_{(\![\cdot]\!)} : A^* \to \mu F \to \mu F \quad \text{-- before each recursive step}$$
$$reflect_{\pi_1} : A^* \to B \to B \quad \text{-- before put (global)}$$

Remastering the example

$$reflect_{map} :: [Name]^+ \to [(Id, Name)] \to [(Id, Name)]$$
$$reflect_{map}\ (n{:}^+ns)\ cs = (0, n) : cs$$
$$reflect_{map}\ ns\ cs = cs$$

$$reflect_{\pi_1} :: [Name]^+ \to Int \to Int$$
$$reflect_{\pi_1}\ (a{:}^+as)\ i = reflect_{\pi_1}\ as\ (succ\ i)$$
$$reflect_{\pi_1}\ (a : as)\ i = reflect_{\pi_1}\ as\ i$$