A Combinatorial Language for Put-based Bidirectional Programming

Hugo Pacheco

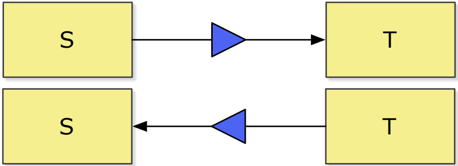
National Institute of Informatics, Tokyo, Japan

IPL Meeting

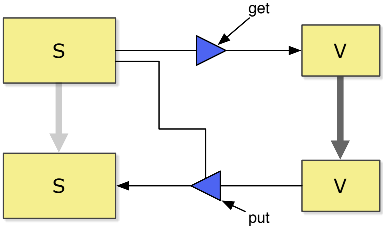
Tokyo - July 2nd, 2013

Bidirectional Transformations (BXs)

"A mechanism for maintaining the consistency of two (or more) related sources of information."



- lenses are one of the most popular BX frameworks



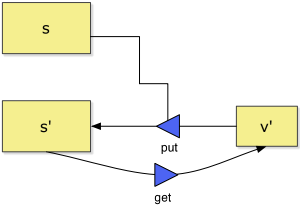
Framework

$$\mathbf{data}\ s \Rightarrow v = Lens\ \{ get :: s \to v \\ \quad , put :: s \to v \to s \}$$

Lens laws

- PUTGET law

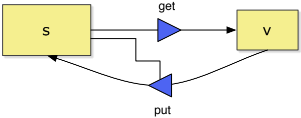
  put must translate view updates exactly.



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

- GETPUT law

  put must preserve empty view updates.

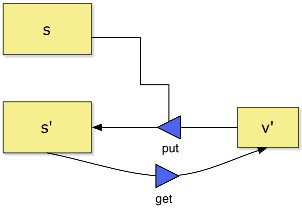


$$put\ s\ (get\ s) = s$$

Partial lens laws

- $\textsc{PutGet}$ law

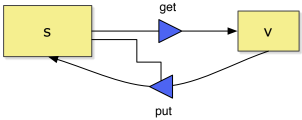
  put must translate view updates exactly. get defined for updated sources.



$$s' \in put\ s\ v' \Rightarrow v' = get\ s'$$

- $\textsc{GetPut}$ law

  put must preserve empty view updates. put defined for empty view updates.

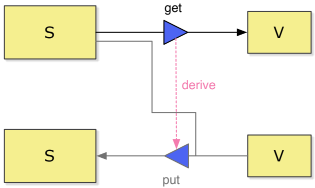


$$v \in get\ s \Rightarrow s = put\ s\ v$$

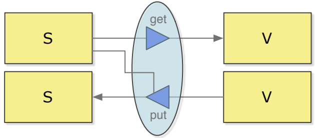
Get-based lens programming

- BX applications vary on the bidirectionalization approach
- write a single program that denotes both transformations

- bidirectionalization: write get in a familiar (unidirectional) programming language and derive a suitable put through particular techniques



- bidirectional programming languages: programs can be interpreted both as a *get* function and a *put* function



Get-based lens programming

- common trait: write *get* and derive *put* automatically
- easy and maintainable
- but requires a careful tradeoff: expressiveness vs updatability
- *get*-based domain-specific lens languages:
  - *put* total (– expressiveness)

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.

H. Pacheco and A. Cunha
Generic Point-free Lenses
Mathematics of Program Construction, 2010.

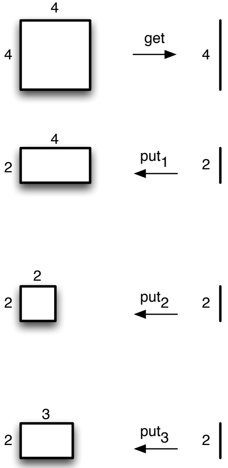
  - *put* partial (– updatability)

D. Liu, Z. Hu, and M. Takeichi
Bidirectional interpretation of XQuery
Partial Evaluation and Program Manipulation, 2007.

Z. Hu, S.-C. Mu, and M. Takeichi
A programmable editor for developing structured documents based on bidirectional transformations
Higher Order and Symbolic Computation, 2008.

Motivation - Ambiguous *put*

- unavoidable ambiguity: it is well-known that there are many possible well-behaved *put*s for a *get*



$$\mathit{height} : (\mathit{Int}, \mathit{Int}) \to \mathit{Int}$$
$$\mathit{height}\ (w, h) = h$$

-- keep original width

$$\mathit{putheight}_1 : (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \to \mathit{Int}$$
$$\mathit{putheight}_1\ (w, h)\ h' = \quad \mathbf{let}\ w' = w\ \mathbf{in}\ (w', h')$$

-- keep the width/height ratio

$$\mathit{putheight}_2 : (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \to \mathit{Int}$$
$$\mathit{putheight}_2\ (w, h)\ h' = \quad \mathbf{let}\ w' = h' * (w \mathbin{/} h)\ \mathbf{in}\ (w', h')$$

-- default width

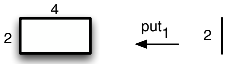
$$\mathit{putheight}_3 : (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \to \mathit{Int}$$
$$\mathit{putheight}_3\ (w, h)\ h' = \quad \mathbf{let}\ w' = \mathbf{if}\ h' \equiv h\ \mathbf{then}\ w\ \mathbf{else}\ 3\ \mathbf{in}\ (w', h')$$

Motivation - An unpractical assumption

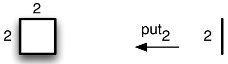
- *get*-based programming has an implicit assumption that

  > *it is sufficient to derive a suitable put that can be combined with get to form a well-behaved lens.*

- but the most suitable *put* does not exist!
- for $get = height \ldots$
  - shall put_{height} preserve the width? (rectangle)



  - shall put_{height} update the width? (square)



- each BX approach will provide its own (typically conservative) solution! $\Rightarrow$ boom of BX approaches over the last 10 years

Motivation - A promising result

Lemma

Given a put function, there exists at most one get function such that GetPut *and* PutGet *hold.*

Theorem (Uniqueness of *get* for well-behaved (partial) *put*)

Assume a put function such that:

1. $(flip\ put)\ v$ is idempotent, i.e., $put\ (put\ s\ v)\ v = put\ s\ v$
2. $put\ s$ is injective

Then (a) there is exactly one get function such that the resulting lens is well-behaved and (b) $get\ s = v \Leftrightarrow s = put\ s\ v$

S. Fischer, Z. Hu and H. Pacheco
"Putback" is the Essence of Bidirectional Programming
GRACE-TR 2012-08, GRACE Center, National Institute of Informatics, December 2012.

Put-based bidirectional programming

- *get*-based = maintainability at the cost of expressiveness or updatability
- write a *get* program from S to V

$$S \xrightarrow{f} U \xrightarrow{g} V$$

- however, writing $put : S \to V \to S$ is much more difficult than writing $get : S \to V$
- idea: language of injective "*put s*" combinators from V to S

$$S \xleftarrow{f} U \xleftarrow{g} V$$

- *put*-based = fully describe a BX!

Framework

$$\mathbf{data}\ s \Leftarrow v = Putlens\ \{ put :: Maybe\ s \to v \to s \\ , get :: s \to v \}$$

A point-free put-based bidirectional language

- functional languages: data domain of algebraic data types
- algebraic data types = trees = sums of products

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

$$\begin{tikzcd} {[A]} \arrow[d, shift right, "\mathit{out}"'] \\ 1 + A \times [A] \arrow[u, shift right, "\mathit{in}"'] \end{tikzcd} \qquad \begin{tikzcd} \mathit{Maybe}\ A \arrow[d, shift right, "\mathit{out}"'] \\ 1 + A \arrow[u, shift right, "\mathit{in}"'] \end{tikzcd}$$

- we will build a point-free *put* language that reverses...

H. Pacheco and A. Cunha
Generic Point-free Lenses
Mathematics of Program Construction, 2010.

... and is inspired in the injective language from...

S.-C. Mu, Z. Hu, and M. Takeichi
An injective language for reversible computation
Mathematics of Program Construction, 2004.

... but is far more expressive!

- elegant formalism to introduce computational effects in functional languages

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \rightarrow m\ a \\
\quad (\gg\!=) \ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\
\quad \mathit{fail} \quad\ :: m\ a
\end{array}
$$

$$\mathit{return}\ x \gg\!= f = f\ x$$

$$m \gg\!= \mathit{return} = m$$

$$(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$$

$$\mathit{fail} \gg\!= (\lambda x \rightarrow m) = \mathit{fail}$$

- imperative-style **do** notation

$$
\begin{array}{l}
\textbf{do}\ x \leftarrow mx \\
\quad\ y \leftarrow my \\
\quad\ \mathit{return}\ (f\ x\ y)
\end{array}
$$

- identity monad (Simple function application)

  instance *Monad Identity* **where** ...

  $runIdentity :: Identity\ a \rightarrow a$

- reader monad (Read values from a shared environment)

  instance *Monad* (*Reader r*) **where** ...

  $ask :: Reader\ r\ r$
  $withReader :: (r \rightarrow r') \rightarrow Reader\ r'\ a \rightarrow Reader\ r\ a$
  $runReader :: Reader\ r\ a \rightarrow r \rightarrow a$

- state monad (Read/write values from/to a shared state)

  instance *Monad* (*State s*) **where** ...

  $getState :: State\ s\ s$
  $putState :: s \rightarrow State\ s\ ()$
  $runState :: State\ s\ a \rightarrow s \rightarrow (a, s)$

Monadic put-based framework

- we augment put functions with an arbitrary monad
- users can instantiate the monad with suitable computational effects in order to refine *put* behavior
- forward *get* functions remain purely functional
- does not affect well-behavedness

Framework

$$\textbf{data}\ s \Leftarrow_m v = \mathit{Putlens}\ \{\ \mathit{put} :: \mathit{Maybe}\ s \rightarrow v \rightarrow m\ s \\ ,\ \mathit{get} :: s \rightarrow v\ \}$$

$$s' \in \mathit{put}\ s\ v' \Rightarrow \mathit{get}\ s' = v' \qquad \textsc{PutGet}_{\Leftarrow}$$

$$v \in \mathit{get}\ s \Rightarrow \mathit{return}\ s = \mathit{put}\ s\ v \qquad \textsc{GetPut}_{\Leftarrow}$$

Monadic put-based framework

- we augment put functions with an arbitrary monad
- users can instantiate the monad with suitable computational effects in order to refine *put* behavior
- forward *get* functions remain purely functional
- does not affect well-behavedness

Framework

$$\mathbf{data}\ s \Leftarrow_m v = \mathit{Putlens}\ \{\mathit{put} :: \mathit{Maybe}\ s \to v \to m\ s\ ,\mathit{get} :: s \to v\}$$

$$s' \in \mathit{put}\ s\ v' \ \Rightarrow\ \mathit{get}\ s' = v' \qquad \textsc{PutGet}_{\Leftarrow}$$

$$\cancel{v \in \mathit{get}\ s \ \Rightarrow\ \mathit{return}\ s = \mathit{put}\ s\ v} \qquad \textsc{GetPut}_{\Leftarrow}$$

Monadic put-based framework

- we augment put functions with an arbitrary monad
- users can instantiate the monad with suitable computational effects in order to refine *put* behavior
- forward *get* functions remain purely functional
- does not affect well-behavedness

Framework

$$\mathbf{data}\ s \Leftarrow_m v = \mathit{Putlens}\ \{\mathit{put} :: \mathit{Maybe}\ s \to v \to m\ s$$
$$,\mathit{get} :: s \to v\}$$

$$s' \in \mathit{put}\ s\ v' \Rightarrow \mathit{get}\ s' = v' \qquad \textsc{PutGet}_{\Leftarrow}$$
$$v \in \mathit{get}\ s \wedge m = \mathit{put}\ s\ v \Rightarrow \mathit{assert}\ (\equiv s)\ m = m \qquad \textsc{GetPut}_{\Leftarrow}$$

$$\mathit{assert} :: \mathit{Monad}\ m \Rightarrow (a \to \mathit{Bool}) \to m\ a \to m\ a$$

Identity and Composition

$$\mathrm{id} \in V \Leftarrow_{\mu} V$$

$$\begin{array}{l} \mathrm{id} :: v \Leftarrow_{m} v \\ \mathrm{id}\ s\ v' = \mathit{return}\ v' \end{array}$$

$$\frac{f \in S \Leftarrow_{\mu} U \quad g \in U \Leftarrow_{\mu} V}{f \circ\!\!< g \in S \Leftarrow_{\mu} V}$$

$$\begin{array}{l} (\circ\!\!<) :: (s \Leftarrow_{m} u) \to (u \Leftarrow_{m} v) \to (s \Leftarrow_{m} v) \\ (f \circ\!\!< g)\ \mathit{Nothing}\ v' = \mathbf{do}\ u' \leftarrow g\ \mathit{Nothing}\ v' \\ \qquad\qquad\qquad\qquad\qquad\quad\ f\ \mathit{Nothing}\ u' \\ (f \circ\!\!< g)\ (\mathit{Just}\ s)\ v' = \mathbf{do}\ u' \leftarrow g\ (\mathit{Just}\ (\mathit{get}\ f\ s))\ v' \\ \qquad\qquad\qquad\qquad\qquad\quad\ f\ (\mathit{Just}\ s)\ u' \end{array}$$

- implementation is well-behaved but partial
- semantic set-theoretic types: well-typed lenses are total

Filtering and bottom

$$\Phi\ V_1 \ \in\ (V_1 \Leftarrow_\mu V_1)$$

$$\begin{aligned} &\Phi :: (v \to \mathit{Bool}) \to (v \Leftarrow_m v) \\ &\Phi\ p\ s\ v' = \textbf{if}\ p\ v'\ \textbf{then}\ \mathit{return}\ v' \\ &\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{fail} \end{aligned}$$

$$\mathrm{bot} \ \in\ (\emptyset \Leftarrow_\mu \emptyset)$$

$$\begin{aligned} &\mathrm{bot} :: s \Leftarrow_m v \\ &\mathrm{bot}\ s\ v' = \mathit{fail} \end{aligned}$$

- partial *put*: only certain views are permitted

Effectful put computations

$$\frac{f \;\in\; \mathit{Maybe}\; S \rightarrow V \rightarrow \mu\; 1 \quad g \;\in\; S \Leftarrow_{\mu} V}{\text{effect}\; f\; g \;\in\; S \Leftarrow_{\mu} V}$$

$$\begin{aligned}
&\text{effect} :: (\mathit{Maybe}\; s \rightarrow v \rightarrow m\; ()) \rightarrow (s \Leftarrow_{m} v) \rightarrow (s \Leftarrow_{m} v)\\
&\text{effect}\; f\; g\; s\; v' = \mathbf{do}\; f\; s\; v'\\
&\qquad\qquad\qquad\qquad\;\; g\; s\; v'
\end{aligned}$$

- run some monadic computation before executing a putlens
- does not affect well-behavedness

Add first element to the source

$$\frac{P \subseteq S_1 \times V \qquad f \in \mathit{Maybe}\ P \to V \to \mu\ S_1 \qquad f\ (\mathit{Just}\ (s_1, v))\ v = \mathit{return}\ s_1}{\text{addfst}\ f \in P \Leftarrow_\mu V}$$

$$\begin{array}{l}
\text{addfst} :: (\mathit{Maybe}\ (s_1, v) \to v \to m\ s_1) \to ((s_1, v) \Leftarrow_m v) \\
\text{addfst}\ f = \mathit{checkGetPut}\ \mathit{put}'\ \textbf{where} \\
\quad \mathit{put}'\ s\ v' = \textbf{do}\ {s_1}' \leftarrow f\ s\ v' \\
\qquad\qquad\qquad\quad \mathit{return}\ ({s_1}', v')
\end{array}$$

- dynamic: repair source creation function to satisfy $\textsc{GetPut}$
- static: possible dependency between view and source values

Keep first element in the source

$$\frac{f \;\in\; V \to \mu\, S_1}{\text{keepfstOr}\ f \;\in\; S_1 \;\times\; V \Leftarrow_{\mu} V}$$

$$\begin{aligned}
&\text{keepfstOr} :: (v \to m\ s_1) \to ((s_1, v) \Leftarrow_m v)\\
&\text{keepfstOr}\ f = \text{addfst}\ f'\ \textbf{where}\ f'\ \mathit{Nothing}\ v' = f\ v'\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad f'\ (\mathit{Just}\ (s_1, v))\ v' = \mathit{return}\ s_1
\end{aligned}$$

$$\text{keepfst} = \text{keepfstOr}\ (\lambda s\ v' \to \mathit{fail})$$

Copy the view element

$$\text{copy} \;\in\; \{(v_1, v_2) \mid v_1 \;\in\; V \wedge v_2 \;\in\; V \wedge v_1 = v_2\} \Leftarrow_{\mu} V$$

$$\begin{aligned}
&\text{copy} :: (v, v) \Leftarrow_m v\\
&\text{copy} = \text{addfst}\ (\lambda s\ v' \to \mathit{return}\ v')
\end{aligned}$$

Drop first element in the view

$$\frac{f \in V \to V_1}{\text{remfst } f \in V \Leftarrow_\mu \{(v_1, v) \mid v_1 \in V_1 \wedge v \in V \wedge v_1 = f\ v\}}$$

$$\begin{array}{l} \text{remfst} :: (v \to v_1) \to (v \Leftarrow_m (v_1, v)) \\ \text{remfst } f\ s\ ({v_1}', v') = \textbf{if } f\ v' \equiv {v_1}' \textbf{ then } \mathit{return}\ v' \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{fail} \end{array}$$

- partial *put*: equality test to guarantee injectivity
- for every pair (v_1, v), v_1 can be reconstructed from $f\ v$

Products - Parallel *put* application

Apply two putlenses to both sides of a pair

$$\frac{f \in S_1 \Leftarrow_\mu V_1 \qquad g \in S_2 \Leftarrow_\mu V_2}{f \otimes g \in S_1 \times S_2 \Leftarrow_\mu V_1 \times V_2}$$

$$
\begin{array}{l}
(\otimes) :: (s_1 \Leftarrow_m v_1) \to (s_2 \Leftarrow_m v_2) \to ((s_1, s_2) \Leftarrow_m (v_1, v_2)) \\
(f \otimes g)\ \mathit{Nothing}\ ({v_1}', {v_2}') = \mathbf{do} \\
\qquad\qquad\qquad\qquad {s_1}' \leftarrow f\ \mathit{Nothing}\ {v_1}' \\
\qquad\qquad\qquad\qquad {s_2}' \leftarrow g\ \mathit{Nothing}\ {v_2}' \\
\qquad\qquad\qquad\qquad \mathit{return}\ ({s_1}', {s_2}') \\
(f \otimes g)\ (\mathit{Just}\ (s_1, s_2))\ ({v_1}', {v_2}') = \mathbf{do} \\
\qquad\qquad\qquad\qquad {s_1}' \leftarrow f\ (\mathit{Just}\ s_1)\ {v_1}' \\
\qquad\qquad\qquad\qquad {s_2}' \leftarrow g\ (\mathit{Just}\ s_2)\ {v_2}' \\
\qquad\qquad\qquad\qquad \mathit{return}\ ({s_1}', {s_2}')
\end{array}
$$

Inject a tag in the view (user-specified predicate)

$$\frac{\begin{array}{c} p \in \mathit{Maybe}\,(V_1 + V_2) \to V_1 \cup V_2 \to \mu\ \mathit{Bool} \\ p\,(\mathit{Just}\,(\mathit{Left}\ v))\ v = \mathit{return}\ \mathit{True} \\ p\,(\mathit{Just}\,(\mathit{Right}\ v))\ v = \mathit{return}\ \mathit{False} \end{array}}{\mathsf{inj}\ p \in V_1 + V_2 \Leftarrow_\mu V_1 \cup V_2}$$

$$\begin{array}{l} \mathsf{inj}\ p :: (\mathit{Maybe}\,(\mathit{Either}\ v\ v) \to v \to m\ \mathit{Bool}) \\ \qquad \to (\mathit{Either}\ v\ v \Leftarrow_m v) \\ \mathsf{inj}\ p = \mathit{checkGetPut}\ \mathit{put}'\ \mathbf{where} \\ \quad \mathit{put}'\ s\ v' = \mathbf{do}\ b \leftarrow p\ s\ v' \\ \qquad\qquad\qquad\qquad \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\,(\mathit{Left}\ v') \\ \qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{return}\,(\mathit{Right}\ v') \end{array}$$

Inject a tag in the view (retrieved from the source)

$$\frac{p \ \in \ V \to \mu\ Bool}{\mathsf{injsOr} \ \in \ V + V \Leftarrow_\mu V}$$

$$\begin{array}{l}
\mathsf{injsOr} :: (v \to m\ Bool) \to (Either\ v\ v \Leftarrow_m v) \\
\mathsf{injsOr}\ p = \mathsf{inj}\ p' \\
\quad \mathbf{where}\ p'\ Nothing\ v' = p\ v' \\
\qquad p'\ (Just\ (Left\ s))\ v' = return\ True \\
\qquad p'\ (Just\ (Right\ s))\ v' = return\ False
\end{array}$$

Inject left/right tags

$$\mathsf{injl} \ \in \ V + \emptyset \Leftarrow_\mu V$$

$$\mathsf{injl} :: Either\ v\ v_2 \Leftarrow_m v$$

$$\mathsf{injr} \ \in \ \emptyset + V \Leftarrow_\mu V$$

$$\mathsf{injr} :: Either\ v_1\ v \Leftarrow_m v$$

Ignore tags in the view

$$\frac{f \;\in\; S_1 \Leftarrow_\mu V_1 \quad g \;\in\; S_2 \Leftarrow_\mu V_2 \quad S_1 \cap S_2 = \emptyset}{f \nabla g \;\in\; S_1 \cup S_2 \Leftarrow_\mu V_1 + V_2}$$

$$\begin{array}{l}
(\,\nabla\,) :: (s \Leftarrow_m v_1) \to (s \Leftarrow_m v_2) \to (s \Leftarrow_m \mathit{Either}\ v_1\ v_2) \\
(f \nabla g)\ s\ (\mathit{Just}\ (\mathit{Left}\ {v_1}'))\ \ = \mathit{assert}\ (\mathit{disjoint}\ f\ g)\ (f\ {v_1}') \\
(f \nabla g)\ s\ (\mathit{Just}\ (\mathit{Right}\ {v_2}')) = \mathit{assert}\ (\mathit{disjoint}\ g\ f)\ (g\ {v_2}') \\
\mathit{disjoint}\ x\ y\ s = \mathit{isJust}\ (\mathrm{get}\ x\ s) \wedge \mathit{isNothing}\ (\mathrm{get}\ y\ s)
\end{array}$$

- constraint: the domains of get_f and get_g must be disjoint to guarantee injectivity (we *get* through the same path as we have *put*)
- extension ("observable" *get* domains)

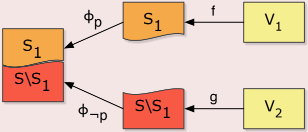
$$\begin{array}{l}
\textbf{data}\ s \Leftarrow_m v = \mathit{PutLens}\ \{\,\mathit{put} : \mathit{Maybe}\ s \to v \to m\ s \\
\qquad\qquad\qquad\qquad\quad ,\ \mathit{get} : s \to \mathit{Maybe}\ v\,\}
\end{array}$$

Ignore tags in the view (source-based branching)

$$\frac{S_1 \subseteq S \quad f \in S_1 \Leftarrow_\mu V_1 \quad g \in S \setminus S_1 \Leftarrow_\mu V_2}{f \nabla_{S_1} g \in S \Leftarrow_\mu V_1 + V_2}$$

$$\nabla_{.} :: (s \to \mathit{Bool}) \to (s \Leftarrow_m v_1) \to (s \Leftarrow_m v_2) \to (s \Leftarrow_m \mathit{Either}\ v_1\ v_2)$$
$$f \nabla_p g = (\Phi\ p \circ\!\!< f) \nabla (\Phi\ (\mathit{not} \circ p) \circ\!\!< g)$$

$$f \mathbin{{}_\bullet\nabla} g \qquad (S_1 = \mathit{dom}\ (\mathit{get}\ f))$$
$$f \mathbin{\nabla_\bullet} g \qquad (S_1 = \mathit{not} \circ \mathit{dom}\ (\mathit{get}\ g))$$



"Uninject" left/right tags

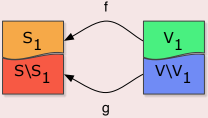
$$\mathsf{uninjl} \in V \Leftarrow_\mu V + \emptyset$$
$$\mathsf{uninjl} :: v \Leftarrow_m \mathit{Either}\ v\ v_2$$

$$\mathsf{uninjr} \in V \Leftarrow_\mu \emptyset + V$$
$$\mathsf{uninjr} :: v \Leftarrow_m \mathit{Either}\ v_1\ v$$

if-then-else view conditional

$$\frac{V_1 \subseteq V \quad f \in S_1 \Leftarrow_\mu V_1 \quad g \in S \setminus S_1 \Leftarrow_\mu V \setminus V_1}{\text{ifVthenelse } V_1\ f\ g \in S \Leftarrow_\mu V}$$

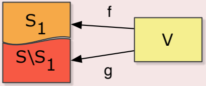
$$\begin{aligned}\text{ifVthenelse} :: &\ (v \to Bool) \to (s \Leftarrow_m v) \\ &\to (s \Leftarrow_m v) \to (s \Leftarrow_m v)\end{aligned}$$



if-then-else source conditional

$$\frac{S_1 \subseteq S \quad f \in S_1 \Leftarrow_\mu V \quad g \in S \setminus S_1 \Leftarrow_\mu V}{\text{ifSthenelse } S_1\ f\ g \in S \Leftarrow_\mu V}$$

$$\begin{aligned}\text{ifSthenelse} :: &\ (s \to Bool) \to (s \Leftarrow_m v) \\ &\to (s \Leftarrow_m v) \to (s \Leftarrow_m v)\end{aligned}$$



Sums - Disjoint *put* application

Applies two putlenses to distinct sides of a sum

$$\frac{f \ \in \ S_1 \Leftarrow_\mu V_1 \quad g \ \in \ S_2 \Leftarrow_\mu V_2}{f \oplus g \ \in \ S_1 + S_2 \Leftarrow_\mu V_1 + V_2}$$

$$
\begin{array}{l}
(\oplus) :: (s_1 \Leftarrow_m v_1) \rightarrow (s_2 \Leftarrow_m v_2) \rightarrow (Either\ s_1\ s_2 \Leftarrow_m Either\ v_1\ v_2) \\
(f \oplus g)\ (Just\ (Left\ s_1))\ (Left\ {v_1}') = \mathbf{do} \\
\qquad\qquad \{ {s_1}' \leftarrow f\ (Just\ s_1)\ {v_1}'; return\ (Left\ {s_1}') \} \\
(f \oplus g)\ s\ (Left\ {v_1}') = \mathbf{do} \\
\qquad\qquad \{ {s_1}' \leftarrow f\ Nothing\ {v_1}'; return\ (Left\ {s_1}') \} \\
(f \oplus g)\ (Just\ (Right\ s_2))\ (Right\ {v_2}') = \mathbf{do} \\
\qquad\qquad \{ {s_2}' \leftarrow f\ (Just\ s_2)\ {v_2}'; return\ (Right\ {s_2}') \} \\
(f \oplus g)\ s\ (Right\ {v_2}') = \mathbf{do} \\
\qquad\qquad \{ {s_2}' \leftarrow f\ Nothing\ {v_2}'; return\ (Right\ {s_2}') \}
\end{array}
$$

Algebraic data types

$$
\begin{array}{lll}
\mathsf{in}_{[A]} \in [A] \Leftarrow_\mu 1 + A \times [A] & & \mathsf{out}_{[A]} \in 1 + A \times [A] \Leftarrow_\mu [A] \\
\mathsf{nil} \in [A] \Leftarrow_\mu 1 & & \mathsf{unnil} \in 1 \Leftarrow_\mu [A] \\
\mathsf{cons} \in [A] \Leftarrow_\mu A \times [A] & & \mathsf{uncons} \in A \times [A] \Leftarrow_\mu [A]
\end{array}
$$

Products

$$
\begin{array}{lcl}
\mathsf{swap} & \in & B \times A \Leftarrow_\mu A \times B \\
\mathsf{assocl} & \in & (A \times B) \times C \Leftarrow_\mu A \times (B \times C) \\
\mathsf{assocr} & \in & A \times (B \times C) \Leftarrow_\mu (A \times B) \times C
\end{array}
$$

Sums

$$
\begin{array}{lcl}
\mathsf{coswap} & \in & B + A \Leftarrow_\mu A + B \\
\mathsf{coassocl} & \in & (A + B) + C \Leftarrow_\mu A + (B + C) \\
\mathsf{coassocr} & \in & A + (B + C) \Leftarrow_\mu (A + B) + C
\end{array}
$$

Distributivity

$$
\begin{array}{lcl}
\mathsf{distl} & \in & ((A \times C) + (B \times C) \Leftarrow_\mu (A + B) \times C \\
\mathsf{distr} & \in & (A \times B) + (A \times C) \Leftarrow_\mu A \times (B + C)
\end{array}
$$

A point-free put-based bidirectional language (Summary)

Language of point-free putlens combinators

$$
\begin{array}{lcll}
\mathit{Put} & ::= & \mathsf{id} \mid \mathit{Put} \circ\!< \mathit{Put} & \text{-- basic combinators} \\
 & \mid & \Phi\ p \mid \mathsf{bot}\ p & \text{-- partial combinators} \\
 & \mid & \mathsf{effect}\ f\ \mathit{Put} & \text{-- monadic effects} \\
 & \mid & \mathit{Prod} \mid \mathit{Sum} \mid \mathit{Cond} \mid \mathit{Iso} \mid \mathit{Rec} & \\
\mathit{Prod} & ::= & \mathsf{addfst}\ f \mid \mathsf{addsnd}\ f \mid \mathsf{keepfstOr} \mid \mathsf{keepsndOr} \mid \mathsf{copy} & \text{-- create pairs} \\
 & \mid & \mathsf{remfst}\ f \mid \mathsf{remsnd}\ f & \text{-- destroy pairs} \\
 & \mid & \mathit{Put} \otimes \mathit{Put} & \text{-- product} \\
\mathit{Sum} & ::= & \mathsf{inj}\ p \mid \mathsf{injsOr} \mid \mathsf{injl} \mid \mathsf{injr} & \text{-- create sums} \\
 & \mid & \mathit{Put} \nabla \mathit{Put} \mid \mathit{Put} \nabla_p \mathit{Put} \mid \mathit{Put} \bullet\!\nabla \mathit{Put} \mid \mathit{Put} \bullet\!\nabla \mathit{Put} & \text{-- destroy sums} \\
 & \mid & \mathsf{uninjl} \mid \mathsf{uninjr} & \text{-- destroy sums} \\
 & \mid & \mathit{Put} + \mathit{Put} & \text{-- sum} \\
\mathit{Cond} & ::= & \mathsf{ifthenelse} \mid \mathsf{ifVthenelse} \mid \mathsf{ifSthenelse} & \text{-- conditional put app.} \\
\mathit{Iso} & ::= & \mathsf{swap} \mid \mathsf{assocl} \mid \mathsf{assocr} & \text{-- rearrange pairs} \\
 & \mid & \mathsf{coswap} \mid \mathsf{coassocl} \mid \mathsf{coassocr} & \text{-- rearrange sums} \\
 & \mid & \mathsf{distl} \mid \mathsf{distr} & \text{-- distr. sums over pairs} \\
\mathit{Rec} & ::= & \mathsf{in} \mid \mathsf{out} & \text{-- algebraic data types}
\end{array}
$$

Example (list embedding)

- *put* function

$$\begin{aligned}&embedAt :: Int \to [a] \to a \to [a]\\&embedAt\ 0\ (x : xs)\ y = y : xs\\&embedAt\ i\ (x : xs)\ y\ = x :\\&\quad embedAt\ (i-1)\ xs\ y\end{aligned}$$

- *get* function

$$\begin{aligned}&elementAt : Int \to [a] \to a\\&elementAt\ 0\ (x : xs) = x\\&elementAt\ i\ (x : xs) =\\&\quad elementAt\ (i-1)\ xs\end{aligned}$$

$$\begin{aligned}&\mathrm{embedAt} :: Int \to ([a] \Leftarrow_{Identity} a)\\&\mathrm{embedAt}\ 0 = \mathrm{unhead}\\&\mathrm{embedAt}\ n = \mathrm{untail} \circ\!\!< \mathrm{embedAt}\ (n-1)\end{aligned}$$

$$\begin{aligned}&\mathrm{unhead} = \mathrm{cons} \circ\!\!< \mathrm{keepsnd}\\&\mathrm{untail}\ \ = \mathrm{cons} \circ\!\!< \mathrm{keepfst}\end{aligned}$$

```
get (embedAt 2) "abcd" = Just 'c'
put (embedAt 2) (Just "abcd") 'x' = Identity "abxd"
put (embedAt 2) (Just "a") 'x' = **undefined
```

Example (list embedding V2)

- *put* function

$$
\begin{aligned}
&\mathit{embedAt} :: \mathit{Int} \to [a] \to a \to [a] \\
&\mathit{embedAt}\ 0\ (x : xs)\ y = y : xs \\
&\mathit{embedAt}\ i\ (x : xs)\ y\ = x : \\
&\quad \mathit{embedAt}\ (i - 1)\ xs\ y
\end{aligned}
$$

- *get* function

$$
\begin{aligned}
&\mathit{elementAt} :: \mathit{Int} \to [a] \to a \\
&\mathit{elementAt}\ 0\ (x : xs) = x \\
&\mathit{elementAt}\ i\ (x : xs) = \\
&\quad \mathit{elementAt}\ (i - 1)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{embedAt}' :: \mathit{Int} \to ([a] \Leftarrow_{\mathit{Identity}} a) \\
&\mathrm{embedAt}'\ 0 = \mathrm{unhead}' \\
&\mathrm{embedAt}'\ n = \mathrm{untail}' \circ\!\!< \mathrm{embedAt}'\ (n - 1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{unhead}' = \mathrm{cons} \circ\!\!< \mathrm{keepsndOr}\ (\lambda v \to \mathit{return}\ [\,]) \\
&\mathrm{untail}' \ \ = \mathrm{cons} \circ\!\!< \mathrm{keepfstOr}\ (\lambda(v : vs) \to \mathit{return}\ v)
\end{aligned}
$$

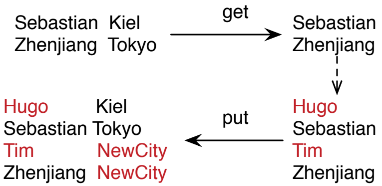
```
get (embedAt’ 2) "a" = Nothing
put (embedAt’ 2) (Just "a") ’x’ = Identity "axx"
```

Example (DB projection)

- *get* function

$$\textbf{type}\ \mathit{Person} = (\mathit{Name}, \mathit{City})$$
$$\mathit{name} :: \mathit{Person} \to \mathit{Name}$$
$$\mathit{city} :: \mathit{Person} \to \mathit{City}$$

$$\mathit{peopleNames} :: [\mathit{Person}] \to [\mathit{Name}]$$
$$\mathit{peopleNames} = \mathit{map}\ \mathit{name}$$



- *put*-based lens

$$\mathrm{map} :: (b \Leftarrow_m a) \to ([b] \Leftarrow_m [a])$$
$$\mathrm{map}\ f = \mathrm{ifVthenelse}\ \mathit{null}\ (\mathrm{nil} \circ\!\!< \mathrm{unnil})\ (\mathrm{cons} \circ\!\!< (f \otimes \mathrm{map}\ f) \circ\!\!< \mathrm{uncons})$$

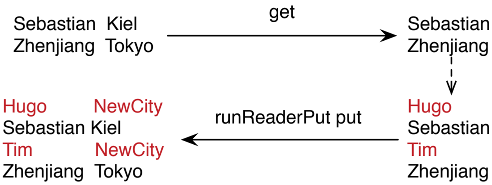
$$\mathrm{peopleNames} :: [\mathit{Person}] \Leftarrow_{\mathit{Identity}} [\mathit{Name}]$$
$$\mathrm{peopleNames} = \mathrm{map}\ (\mathrm{addsnd}\ \mathit{cityOf})$$
$$\quad \textbf{where}\ \mathit{cityOf}\ (\mathit{Just}\ s)\ v = \mathit{return}\ s$$
$$\qquad\qquad \mathit{cityOf}\ \mathit{Nothing}\ v = \mathit{return}\ \texttt{"NewCity"}$$

Example (DB projection with environment)

- *put*-based lens

$$
\begin{aligned}
&\mathrm{peopleNames} : [\mathit{Person}] \Leftarrow_{\mathit{Reader}\ [\mathit{Person}]} [\mathit{Name}] \\
&\mathrm{peopleNames} = \mathrm{map}\ (\mathrm{addsnd}\ \mathit{cityOf}\,) \\
&\quad \mathbf{where}\ \mathit{cityOf}\ \ s\ \ n = \mathbf{do}\ \mathit{people} \leftarrow \mathit{ask} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{case}\ \mathit{lookup}\ n\ \mathit{people}\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ c \rightarrow \mathit{return}\ c \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{return}\ \texttt{"NewCity"}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{runReaderPut} :: (s \Leftarrow_{\mathit{Reader}\ s} v) \rightarrow (s \rightarrow v \rightarrow s) \\
&\mathit{runReaderPut}\ \mathit{put}\ s\ v = \mathit{runReader}\ (\mathit{put}\ (\mathit{Just}\ s)\ v)\ s
\end{aligned}
$$

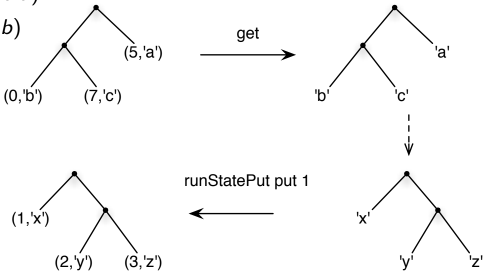


- *get* function

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\begin{array}{l} \mathit{mapTree} :: (a \to b) \to (\mathit{Tree}\ a \to \mathit{Tree}\ b) \\ \mathit{mapTree}\ f\ (\mathit{Tip}\ x) = x \\ \mathit{mapTree}\ f\ (\mathit{Bin}\ l\ r) = \mathit{Bin} \\ \quad (\mathit{mapTree}\ f)\ (\mathit{mapTree}\ g) \end{array}$$

$$\begin{array}{l} \mathit{dropLabels} :: \mathit{Tree}\ (\mathit{Symbol}, a)\ a \\ \mathit{dropLabels} = \mathit{mapTree}\ \mathit{snd} \end{array}$$



- *put*-based lens

$$\begin{array}{l} \mathrm{mapTree} :: (b \Leftarrow_m a) \to (\mathit{Tree}\ b \Leftarrow_m \mathit{Tree}\ a) \\ \mathrm{mapTree}\ f = \mathrm{in} \circ\!\!< (f \oplus \mathrm{mapTree}\ f \otimes \mathrm{mapTree}\ f) \circ\!\!< \mathrm{out} \end{array}$$

$$\begin{array}{l} \mathrm{freshLabels} :: \mathit{Tree}\ (\mathit{Symbol}, a) \Leftarrow_{\mathit{State}\ \mathit{Symbol}}\ a \\ \mathrm{freshLabels} = \mathit{mapTree}\ (\mathrm{addfst}\ \mathit{freshLabel})\ \textbf{where} \\ \quad \mathit{freshLabel}\ s\ v \to \textbf{do}\ \{s \leftarrow \mathit{State.get}; \mathit{State.put}\ (s + 1); \mathit{return}\ s\} \end{array}$$

$$\begin{array}{l} \mathit{runStatePut} :: s \Leftarrow_{\mathit{State}\ st}\ v \to st \to (s \to v \to s) \\ \mathit{runStatePut}\ \mathit{put}\ st\ s\ v = \textbf{let}\ (s', st') = \mathit{runState}\ (\mathit{put}\ (\mathit{Just}\ s)\ v)\ st\ \textbf{in}\ s' \end{array}$$

- exception (Handle failures)

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{MonadException}\ m \textbf{ where} \\
\quad \mathit{catch} :: m\ a \rightarrow m\ a \rightarrow m\ a \\
\textbf{instance } \mathit{MonadException}\ \mathit{Maybe} \textbf{ where} \ldots
\end{array}
$$

$$\mathit{catch}\ \mathit{fail}\ m = m$$

$$\mathit{catch}\ m\ \mathit{fail} = m$$

Inject a tag in the view (using catch)

$$\frac{f \in S_1 \Leftarrow_\mu V_1 \qquad g \in S_2 \Leftarrow_\mu V_2}{\text{injException } f\ g \in S_1 + S_2 \Leftarrow_\mu V_1 \cup V_2}$$

$$
\begin{array}{l}
\text{injException} :: \mathit{MonadException}\ m \Rightarrow (s_1 \Leftarrow_m v) \rightarrow (s_1 \Leftarrow_m v) \\
\qquad\qquad\qquad \rightarrow (\mathit{Either}\ s_1\ s_2 \Leftarrow_m v) \\
\text{injException } f\ g\ \mathit{Nothing}\ v' = \\
\quad \mathit{liftM}\ \mathit{Left}\ (\mathit{put}\ f\ \mathit{Nothing}\ v')\ \text{`}\mathit{catch}\text{`}\ \mathit{liftM}\ \mathit{Right}\ (\mathit{put}\ g\ \mathit{Nothing}\ v') \\
\text{injException } f\ g\ (\mathit{Just}\ (\mathit{Left}\ s_1))\ v' = \\
\quad \mathit{liftM}\ \mathit{Left}\ (\mathit{put}\ f\ (\mathit{Just}\ s_1)\ v')\ \text{`}\mathit{catch}\text{`}\ \mathit{liftM}\ \mathit{Right}\ (\mathit{put}\ g\ \mathit{Nothing}\ v') \\
\text{injException } f\ g\ (\mathit{Just}\ (\mathit{Right}\ s_2))\ v' = \\
\quad \mathit{liftM}\ \mathit{Right}\ (\mathit{put}\ g\ (\mathit{Just}\ s_2)\ v')\ \text{`}\mathit{catch}\text{`}\ \mathit{liftM}\ \mathit{Left}\ (\mathit{put}\ f\ \mathit{Nothing}\ v')
\end{array}
$$

Example (*unwords* with exception)

- *get* function

$$
\begin{aligned}
&unwords :: [String] \rightarrow String\\
&unwords\ [\,] \;= \texttt{""}\\
&unwords\ ws = foldr1\ (\lambda w\ s \rightarrow w +\!\!+\ \texttt{'~'} : s)\ ws
\end{aligned}
$$

$$
\begin{aligned}
&foldr1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a\\
&foldr1\ f\ [x] \qquad = x\\
&foldr1\ f\ (x : xs) = f\ x\ (foldr1\ f\ xs)
\end{aligned}
$$

- *put*-based lens

$$
\begin{aligned}
&\mathrm{words} :: [String] \Leftarrow_{Maybe} String\\
&\mathrm{words} = (\mathrm{nil} \mathbin{\bullet\!\nabla} \mathrm{id}) \circ\!\!< \mathrm{injException}\ (\mathrm{ignore}\ \texttt{""})\ (\mathrm{unfoldr1}\ (\mathrm{appendWithSep}\ \texttt{" "}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{unfoldr1} :: MonadException\ m \Rightarrow ((a, a) \Leftarrow_m a) \rightarrow ([a] \Leftarrow_m a)\\
&\mathrm{unfoldr1}\ f = (\mathrm{cons} \mathbin{\nabla\!\bullet} \mathrm{wrap}) \circ\!\!< \mathrm{injException}\ ((\mathrm{id} \otimes \mathrm{unfoldr1}\ f) \circ\!\!< f)\ \mathrm{id}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{appendWithSep} :: Monad\ m \Rightarrow String \rightarrow ((String, String) \Leftarrow_m String)\\
&\mathrm{ignore} :: Monad\ m \Rightarrow e \Leftarrow_m v
\end{aligned}
$$

```
get words ["a","b","c"] = Just "a b c"
put words Nothing "hu  go " = Just ["hu","","go",""]
```

Conclusions

- a novel point-free put-based BX language (flexible, expressive)
- we propose to shift into a *put* programming style
  - programmers write well-behaved *put*
  - language provides unique *get* for free
- *put programming is more powerful than get programming, not easier, but not necessarily more complex*
- this shift is manageable
  - the combinators offer different default *put* behaviors
  - more complex *put* behaviors using monadic effects
- this shift is necessary
  - programmers can fully control/specify BXs (predictability)
  - more expressive than existing get-based languages (user's intentions)

Demos: Haskell++

- `http://hackage.haskell.org` $\Rightarrow$ `putlenses`

- type checking & type inference
- better static guarantees and programmability
- fully expressive putlens language $\longleftrightarrow$ less expressive higher-level put-based DSL (`BiFlux` in the works...)
- synthesize more efficient *put* and *get* functions
- languages for other domains (e.g., lenses for relational data)

A. Bohannon, B. C. Pierce, and J. A. Vaughan
Relational lenses: a language for updatable views
Principles of Database Systems, 2006.