“Point-free” Put-based Bidirectional Programming

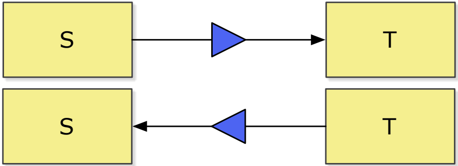
Hugo Pacheco

National Institute of Informatics, Tokyo, Japan

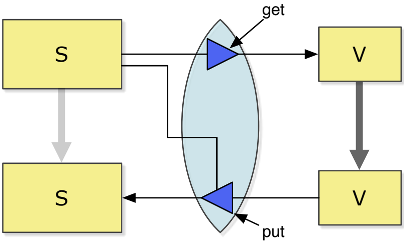
HasLab Seminar

Braga - April 3rd, 2013

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



- lenses are one of the most popular BX frameworks

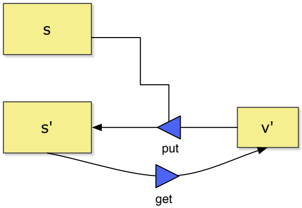


Framework

$$\begin{aligned}\textbf{data}\ S \Rrightarrow V = \mathit{Lens}\ \{ & \mathit{get} : S \rightarrow V \\ , & \mathit{put} : S \rightarrow V \rightarrow S \}\end{aligned}$$

- PUTGET law

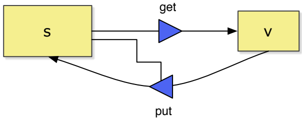
  put must translate view updates exactly. get defined for updated sources.



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

- GETPUT law

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



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

Lens programming

- BX applications vary on the bidirectionalization approach
- common trait: write *get* and derive *put* automatically
- easy and maintainable
- *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.

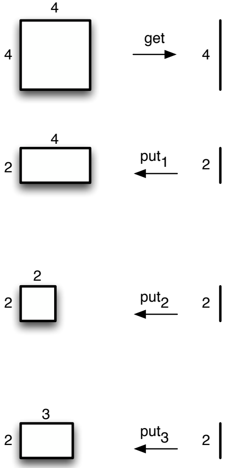
  - *put* partial (– updatability)

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

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

-- keep the width/height ratio

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

-- default width

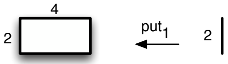
$$\mathit{putheight}_3 : (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \to \mathit{Int}$$
$$\mathit{putheight}_3\ (w, h)\ h' =$$
$$\quad \textbf{let}\ w' = \textbf{if}\ h' \equiv h\ \textbf{then}\ w\ \textbf{else}\ 3\ \textbf{in}\ (w', h')$$

Motivation - An unpractical assumption

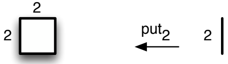
- *get*-based programming has an implicit assumption that

  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 solution! $\Rightarrow$ boom of BX approaches over the last 10 years

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. $(\mathit{flip}\ \mathit{put})\ v$ *is idempotent, i.e.,* $\mathit{put}\ (\mathit{put}\ s\ v)\ v = \mathit{put}\ s\ v$
2. $\mathit{put}\ s$ *is injective*

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

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 expressivness
- write *get* 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 = describe the behavior of a BX completely!

Framework

$$\mathbf{data}\ S \Leftarrow V = \mathit{Putlens}\ \{\mathit{put} : S \to V \to S \\ , \mathit{get} : S \to V\}$$

A point-free put-based bidirectional language

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

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

$$\begin{tikzcd} {[A]} \arrow[d, shift right, "out"'] \\ \mathit{Either}\ ()\ (A,[A]) \arrow[u, shift right, "in"'] \end{tikzcd} \qquad \begin{tikzcd} \mathit{Maybe}\ A \arrow[d, shift right, "out"'] \\ \mathit{Either}\ ()\ A \arrow[u, shift right, "in"'] \end{tikzcd}$$

- 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.

- partial *put* combinators = no updatability problem

Identity and Composition

$$\mathrm{id} : V \Leftarrow V$$
$$put\ s\ v' = v'$$

$$\forall\ f : S \Leftarrow U, g : U \Leftarrow V.\ (f \circ\!\!< g) : S \Leftarrow V$$
$$(f \circ\!\!< g)\ s\ v' = (put_f\ s \circ put_g\ (get_f\ s))\ v'$$

Filtering and bottom

$$\forall\ p : V \to Bool.\ (\Phi\ p : V \Leftarrow V)$$
$$(\Phi\ p)\ s\ v' \mid p\ v' = v'$$

$$\mathrm{bot} :: S \Leftarrow V$$
$$\mathrm{bot}\ s\ v' = \perp$$

- partial *put*: only certain views are permitted

Products - Creating pairs

Add first element to the source

$$\forall\ f : (S_1, V) \to V \to S_1.\ \text{addfst}\ f : (S_1, V) \Leftarrow V$$
$$\mathit{put}\ (s_1, v)\ v' = ({s_1}', v')$$
$$\quad \textbf{where}\ {s_1}' = \textbf{if}\ v' \equiv v\ \textbf{then}\ s_1\ \textbf{else}\ f\ (s_1, v)\ v'$$

Keep first element in the source

$$\text{keepfst} : (S_1, V) \Leftarrow V$$
$$\text{keepfst} = \text{addfst}\ (\lambda(s_1, v)\ v' \to s_1)$$

- similar for *addrPut*, keepsnd

Drop first element in the view

$$\forall\ f : V \rightarrow V_1.\ \mathsf{remfst} : V \Leftarrow (V_1, V)$$
$$put\ v\ ({v_1}', v') \mid f\ v' \equiv {v_1}' = v'$$

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

Products - Parallel *put* application

Apply two putlenses to both sides of a pair

$$\forall\ f : S_1 \Leftarrow V_1, g : S_2 \Leftarrow V_2.\ f \otimes g : (S_1, S_2) \Leftarrow (V_1, V_2)$$

$$\begin{aligned}
&put\ (s_1, s_2)\ ({v_1}', {v_2}') = ({s_1}', {s_2}') \\
&\quad \textbf{where}\ {s_1}' = put_f\ s_1\ {v_1}' \\
&\qquad\qquad\ \ {s_2}' = put_g\ s_2\ {v_2}'
\end{aligned}$$

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

$$\forall\ p : \mathit{Either}\ V\ V \to V \to \mathit{Bool}.\ \mathsf{inj}\ p : \mathit{Either}\ V\ V \Leftarrow V$$

$$\begin{aligned} &\mathit{put}\ s\ v' \mid \mathit{either}\ \mathit{id}\ \mathit{id}\ s \equiv v' = s \\ &\phantom{\mathit{put}\ s\ v'} \mid \mathit{otherwise} = \mathbf{if}\ p\ s\ v'\ \mathbf{then}\ \mathit{Left}\ v'\ \mathbf{else}\ \mathit{Right}\ v' \end{aligned}$$

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

$$\mathsf{injS} : \mathit{Either}\ V\ V \Leftarrow V$$

$$\mathsf{injS} = \mathsf{inj}\ (\lambda s\ v' \to \mathit{either}\ \mathit{True}\ \mathit{False}\ s)$$

Ignore tags in the view

$$\forall\ f : S \Leftarrow V_1, g : S \Leftarrow V_2.\ f \nabla g : S \Leftarrow Either\ V_1\ V_2$$
$$put\ s\ (Left\ v_1) = disjoint\ f\ g\ (put_f\ s\ v_1)$$
$$put\ s\ (Right\ v_2) = disjoint\ g\ f\ (put_g\ s\ v_2)$$
$$disjoint\ x\ y\ s \mid (isJust\ (get\ x\ s)) \wedge isNothing\ (get\ y\ s) = s$$

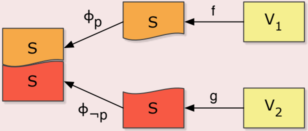
- constraint: the domains of get_f and get_g must be disjoint to guarantee injectivity
- extension ("observable" *get* domains)

$$\textbf{data}\ S \Leftarrow V = PutLens\ \{ put : S \rightarrow V \rightarrow S \\ , get : S \rightarrow Maybe\ V \}$$

Ignore tags in the view (source-based branching)

$$\forall\ p : S \to Bool, f : S \Leftarrow V_1, g : S \Leftarrow V_2.\ f\ \nabla_p\ g : S \Leftarrow Either\ V_1\ V_2$$
$$f\ \nabla_p\ g = (\Phi\ p) \circ f\ \nabla\ (\Phi\ (\neg \circ p)) \circ g$$

$$dom\ f\ s = \textbf{case}\ get_f\ s\ \textbf{of}$$
$$\{Nothing \to False; Just\ \_ \to True\}$$
$$f\ {}_\bullet\!\nabla\ g = f\ \nabla_{dom\ f}\ g$$
$$f\ \nabla_{\!\bullet}\ g = f\ \nabla_{\neg \circ dom\ g}\ g$$



if-then-else view conditionals

$$\forall\ p : S \to V \to Bool, f : S \Leftarrow V, g : S \Leftarrow V.\ \text{ifthenelse}\ p\ f\ g : S \Leftarrow V$$
$$\text{ifthenelse}\ p\ f\ g = (f\ \nabla_{\phi_{dom\ f}}\ g) \circ \text{inj}\ p$$

$$\forall\ p : V \to Bool, f : S \Leftarrow V, g : S \Leftarrow V.\ \text{ifVthenelse}\ p\ f\ g : S \Leftarrow V$$

$$\forall\ p : S \to Bool, f : S \Leftarrow V, g : S \Leftarrow V.\ \text{ifSthenelse}\ p\ f\ g : S \Leftarrow V$$

Sums - Disjoint *put* application

Applies two putlenses to distinct sides of a choice

$$\forall\ f : S_1 \Leftarrow V_1, g : S_2 \Leftarrow V_2.\ f \oplus g : Either\ S_1\ S_2 \Leftarrow Either\ V_1\ V_2$$

$$\begin{array}{ll}
put\ (Just\ (Left\ s_1)) & (Left\ v_1{}') = Left\ (put_f\ (Just\ s_1)\ v_1{}') \\
put\ s & (Left\ v_1{}') = Left\ (put_f\ Nothing\ v_1{}') \\
put\ (Just\ (Right\ s_2)) & (Right\ v_2{}') = Right\ (put_g\ (Just\ s_2)\ v_2{}') \\
put\ s & (Right\ v_2{}') = Right\ (put_g\ Nothing\ v_2{}')
\end{array}$$

- extension (source value creation)

$$\begin{array}{l}
\mathbf{data}\ S \Leftarrow V = PutLens\ \{\ put : Maybe\ S \to V \to S \\
\qquad\qquad\qquad\qquad\quad ,\ get : S \to Maybe\ V\ \}
\end{array}$$

Inject and "uninject" left/right tags

$$\begin{array}{ll}
\mathrm{injl} : Either\ V\ S_2 \Leftarrow V & \mathrm{injr} : Either\ V\ S_2 \Leftarrow V \\
\mathrm{uninjl} : V \Leftarrow Either\ V\ S_2 & \mathrm{uninjr} : V \Leftarrow Either\ V\ S_2
\end{array}$$

Algebraic data types

$$\mathsf{in}_{[A]} : [A] \Leftarrow \mathit{Either}\ ()\ (A, [A]) \qquad \mathsf{out}_{[A]} : \mathit{Either}\ ()\ (A, [A]) \Leftarrow [A]$$

$$\begin{aligned} &\mathsf{nil} : [A] \Leftarrow (), \mathsf{cons} : [A] \Leftarrow (A, [A]) && \mathsf{unnil} : () \Leftarrow [A], \mathsf{uncons} : (A, [A]) \Leftarrow [A] \\ &\mathsf{nil} = \mathsf{in}_{[A]} \circ \mathsf{injl} && \mathsf{unnil} = \mathsf{uninjl} \circ \mathsf{out}_{[A]} \\ &\mathsf{cons} = \mathsf{in}_{[A]} \circ \mathsf{injr} && \mathsf{uncons} = \mathsf{uninjr} \circ \mathsf{out}_{[A]} \end{aligned}$$

Products

$$\begin{aligned} &\mathit{swap} : (B, A) \Leftarrow (A, B) \\ &\mathit{assocl} : ((A, B), C) \Leftarrow (A, (B, C)) \qquad \mathit{assocr} : (A, (B, C)) \Leftarrow ((A, B), C) \end{aligned}$$

Sums

$$\begin{aligned} &\mathit{coswap} : \mathit{Either}\ B\ A \Leftarrow \mathit{Either}\ A\ B \\ &\mathit{coassocl} : \mathit{Either}\ (\mathit{Either}\ A\ B)\ C \Leftarrow \mathit{Either}\ A\ (\mathit{Either}\ B\ C) \\ &\mathit{coassocr} : \mathit{Either}\ A\ (\mathit{Either}\ B\ C) \Leftarrow \mathit{Either}\ (\mathit{Either}\ A\ B)\ C \end{aligned}$$

Distributivity

$$\begin{aligned} &\mathit{distl} : \mathit{Either}\ (A, C)\ (B, C) \Leftarrow (\mathit{Either}\ A\ B, C) \\ &\mathit{distr} : \mathit{Either}\ (A, B)\ (A, C) \Leftarrow (A, \mathit{Either}\ B\ C) \end{aligned}$$

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

Language of point-free putlens combinators

$$
\begin{array}{lcll}
\mathit{Put} & ::= & \mathsf{id} \mid \mathit{Put} \circ\!\!< \mathit{Put} \mid \Phi\ p \mid \mathsf{bot}\ p \mid \mathit{Prod} \mid \mathit{Sum} \mid \mathit{Cond} \mid \mathit{Iso} \mid \mathit{Rec} & \\
\mathit{Prod} & ::= & \mathsf{addfst}\ f \mid \mathsf{addsnd}\ f & \text{-- create pairs} \\
 & \mid & \mathsf{remfst}\ f \mid \mathsf{remsnd}\ f & \text{-- destroy pairs} \\
 & \mid & \mathit{Put} \otimes \mathit{Put} & \text{-- product} \\
\mathit{Sum} & ::= & \mathsf{inj}\ p & \text{-- create choices} \\
 & \mid & \mathit{Put} \nabla \mathit{Put} & \text{-- destroy choices} \\
 & \mid & \mathit{Put} + \mathit{Put} & \text{-- sum} \\
\mathit{Cond} & ::= & \mathsf{ifthenelse} \mid \mathsf{ifVthenelse} \mid \mathsf{ifSthenelse} & \text{-- conditional put app.} \\
\mathit{Iso} & ::= & \mathsf{swap} \mid \mathsf{assocl} \mid \mathsf{assocr} & \text{-- rearrange pairs} \\
 & \mid & \mathsf{coswap} \mid \mathsf{coassocl} \mid \mathsf{coassocr} & \text{-- rearrange choices} \\
 & \mid & \mathsf{distl} \mid \mathsf{distr} & \text{-- distr. choices over pairs} \\
\mathit{Rec} & ::= & \mathsf{in} \mid \mathsf{out} & \text{-- algebraic data types}
\end{array}
$$

Example (list embedding)

- *put* function

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

- *get* function

$$
\begin{aligned}
&\mathit{get} : \mathit{Int} \to [A] \to A \\
&\mathit{get}\ 0\ (x : xs) = x \\
&\mathit{get}\ i\ (x : xs) = \\
&\quad \mathit{get}\ (\mathit{pred}\ i)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{embedAt} : \mathit{Int} \to [A] \Leftarrow A \\
&\mathrm{embedAt}\ i = \mathrm{remsnd}\ (\mathit{const}\ i) \circ\!\!< \mathrm{embedAt}' \\
&\mathrm{embedAt}' : (\mathit{Int}, [A]) \Leftarrow A \\
&\mathrm{embedAt}' = \mathrm{ifSthenelse}\ (\lambda(i, l) \to i \equiv 0)\ \mathit{zero}\ \mathit{nonzero} \\
&\quad \textbf{where}\ \mathit{zero} = \mathrm{addfst}\ (\lambda s\ v \to 0) \circ\!\!< \mathrm{unhead} \\
&\qquad\qquad \mathit{nonzero} = ((+1) \otimes \mathit{untail}) \circ\!\!< \mathrm{embedAt} \\
&\mathrm{unhead} = \mathrm{cons} \circ\!\!< \mathrm{keepsnd} \\
&\mathrm{untail} = \mathrm{cons} \circ\!\!< \mathrm{keepfst}
\end{aligned}
$$

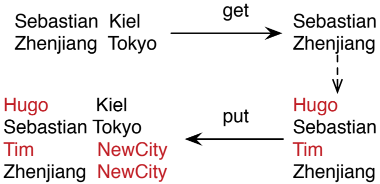
```
get (embedAt 2) "abcd" = Just 'c'
put (embedAt 2) (Just "abcd") 'x' = "abxd"
put (embedAt 2) (Just "a") 'x' = **undefined
```

Example (DB projection)

- *get* function

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

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



- *put*-based lens

$$\mathrm{map} : B \Leftarrow A \to [B] \Leftarrow [A]$$
$$\mathrm{map}\ f = \mathrm{ifVthenelse}\ null\ (\mathrm{nil} \circ\!\!< \mathrm{unnil})\ it$$
$$\quad \textbf{where}\ it = \mathrm{cons} \circ\!\!< (f \otimes \mathrm{map}\ f) \circ\!\!< \mathrm{uncons}$$

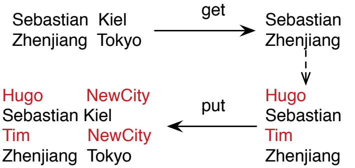
$$\mathrm{peopleNames}_0 : [Person] \Leftarrow [Name]$$
$$\mathrm{peopleNames}_0 = \mathrm{map}\ (\mathrm{addsnd}\ cityOf)$$
$$\quad \textbf{where}\ cityOf\ s\ v = maybe\ \texttt{"NewCity"}\ id\ s$$

Example (DB projection with environment)

- *put*-based lens

$$\text{peopleNames} : [\,Person\,] \Leftarrow_E [\,Name\,]$$
$$\text{peopleNames} = withMbS\ \text{peopleNames}'$$

$$\text{peopleNames}' : [\,Person\,] \Leftarrow_{[\,Person\,]} [\,Name\,]$$
$$\text{peopleNames}' = \text{map}\ (\text{addsnd}\ cityOf\,)$$
$$\quad \textbf{where}\ cityOf\ \ people\ n =$$
$$\qquad \textbf{case}\ lookup\ n\ people\ \textbf{of}$$
$$\qquad\quad Just\ c \rightarrow c$$
$$\qquad\quad Nothing \rightarrow \texttt{"NewCity"}$$



- extension (global environment)

$$\textbf{data}\ S \Leftarrow_E V = PutLens\ \{\, put : Maybe\ S \rightarrow V \rightarrow Reader\ E\ S$$
$$\qquad\qquad\qquad\qquad , get : S \rightarrow Maybe\ V\,\}$$

$$\text{addsnd} : (E \rightarrow A \rightarrow B) \rightarrow (A, B) \Leftarrow_E A$$

$$withMbS\ : (S \Leftarrow_{Maybe\ S} V) \rightarrow (S \Leftarrow_E V)$$
$$withMbV : (S \Leftarrow_{Maybe\ V} V) \rightarrow (S \Leftarrow_E V)$$
$$withV'\ \ : (S \Leftarrow_V V) \rightarrow (S \Leftarrow_E V)$$

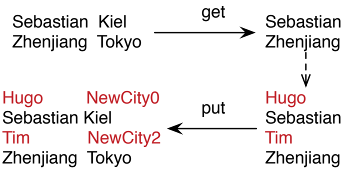
Example (DB projection with state)

- *put*-based lens

$$\text{peopleNames}_1 = \mathit{initSt}\ (\lambda e\ v \rightarrow 0)$$

$$\text{peopleNames}_1{}' : [\mathit{Person}] \Leftarrow^{\mathit{Int}}_{[\mathit{Person}]} [\mathit{Name}]$$

$$\begin{array}{l} \text{peopleNames}_1{}' = \text{map} \\ \quad (\mathit{updateSt}\ \mathit{upd}\ (\text{addsnd}\ \mathit{cityOf})) \\ \qquad \textbf{where}\ \mathit{cityOf}\ i\ \mathit{people}\ n = \\ \qquad\qquad\qquad \textbf{case}\ \mathit{lookup}\ n\ \mathit{people}\ \textbf{of} \\ \qquad\qquad\qquad\quad \mathit{Just}\ c \rightarrow c \\ \qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \texttt{"NewCity"} +\!\!+\ \mathit{show}\ i \\ \qquad\qquad\quad \mathit{upd}\ i\ e\ s = i + 1 \end{array}$$



- extension (state)

$$\begin{array}{l} \textbf{data}\ S \Leftarrow^{\mathit{St}}_{E} V = \mathit{PutLens}\ \{ \mathit{put} : \mathit{Maybe}\ S \rightarrow V \rightarrow \mathit{ReaderT}\ E\ (\mathit{State}\ \mathit{St})\ S \\ \qquad\qquad\qquad\qquad\qquad , \mathit{get} : S \rightarrow \mathit{Maybe}\ V \} \end{array}$$

$$\begin{array}{ll} \mathit{initSt} & : (E \rightarrow V \rightarrow \mathit{St}) \rightarrow (S \Leftarrow^{\mathit{St}}_{E} V) \rightarrow (S \Leftarrow_{E} V) \\ \mathit{updateSt} & : (\mathit{St} \rightarrow E \rightarrow S \rightarrow \mathit{St}) \rightarrow (S \Leftarrow^{\mathit{St}}_{E} V) \rightarrow (S \Leftarrow^{\mathit{St}}_{E} V) \end{array}$$

Conclusions

- a novel point-free put-based BX language
- we propose to shift into a *put* programming style
  - users write well-behaved *put*
  - language provides unique *get* for free
- *put programming is not easier, but rather more powerful*
- this shift is manageable
  - the combinators encapsulate different *put* behaviors
  - complex *put* behaviors by composition (and using extensions)
- this shift is necessary
  - full control of the backward transformation (user's intentions)

+ more expressive than existing total get-based languages
+ better updatability than existing partial get-based languages

Future Work

Demos: Haskell++

- `http://hackage.haskell.org` ⇒ `putlenses`

- synthesize more efficient *put* and *get* functions
- point-free VS point-wise: translate higher-level functional put programming language to lower-level core language
- 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.