“Point-free” Put-based Bidirectional Programming

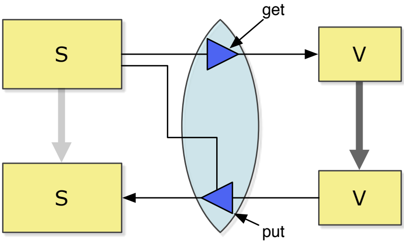
Hugo Pacheco

HASLab, INESC TEC & University of Minho, Braga, Portugal
Former
National Institute of Informatics, Tokyo, Japan
Future

Big Camp

Karuizawa - February 18th, 2013

- lenses are one of the most popular BX frameworks



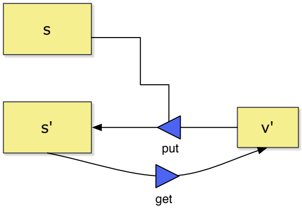
Framework

$$\textbf{data}\ S \Rightarrow V = \mathit{Lens}\ \{\mathit{get} : S \rightarrow V$$
$$,\ \mathit{put} : S \rightarrow V \rightarrow S\}$$

Lens laws w/ partiality

- PutGet law

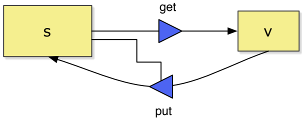
  put must translate view updates exactly. get defined for updated sources.



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

- 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: derive a lens from a *get* specification
- *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.

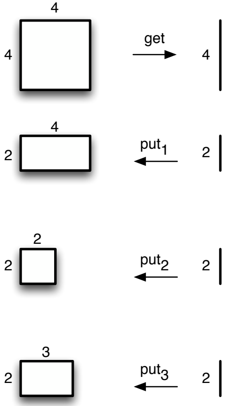
  - *put* partial (– updatability)

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*

- it is well-known that there are many possible well-behaved *puts* for a *get*



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

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

$$\mathit{putheight}_2 : (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{Int} \rightarrow \mathit{Int}$$
$$\mathit{putheight}_2\ (w, h)\ h' = \textbf{let}\ w' = h'\ \textbf{in}\ (w', h')$$

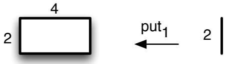
$$\mathit{putheight}_3 : (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{Int} \rightarrow \mathit{Int}$$
$$\mathit{putheight}_3\ (w, h)\ h' = \textbf{let}\ w' = \textbf{if}\ h' \equiv h\ \textbf{then}\ w\ \textbf{else}\ 3\ \textbf{in}\ (w', h')$$

Motivation - An unpractical assumption

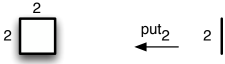
- *get*-based programming has an implicit assumption that

  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 $get = height$...
  - shall put_{height} preserve the width? (rectangle)



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



- each BX approach will provide its own solution!

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. $(\mathit{flip}\ \mathit{put})\ v$ *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

- however, writing $put : S \to V \to S$ is much more difficult than writing $get : S \to V$
- *get*-based = combinators hide synchronization

$$S \overset{f}{\Longrightarrow} U \overset{g}{\Longrightarrow} V$$

$$get_{f;g} = get_f ; get_g$$
$$put_{f;g}\ s = put_f\ s \circ put_g\ (get_f\ s)$$

- idea: language of injective *put s* combinators from V to S
- *put*-based = combinators hide synchronization

$$S \overset{f}{\Longleftarrow} U \overset{g}{\Longleftarrow} V$$

Framework

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

A point-free put-based bidirectional language

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

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

$$[A] \underset{\mathit{in}}{\overset{\mathit{out}}{\rightleftarrows}} \mathit{Either}\ ()\ (A,[A]) \qquad \mathit{Maybe}\ A \underset{\mathit{in}}{\overset{\mathit{out}}{\rightleftarrows}} \mathit{Either}\ ()\ A$$

- 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.

Add left element to the source

$$\forall\ f : (A, B) \rightarrow B \rightarrow A.\ \mathit{addl} : (A, B) \Leftarrow B$$
$$\mathit{put}\ (x, y)\ y' = (x', y')$$
$$\quad \textbf{where}\ x' = \textbf{if}\ y' \equiv y\ \textbf{then}\ x\ \textbf{else}\ f\ (x, y)\ y'$$
$$\mathit{get}\ (x, y) = y$$

Keep left element in the source

$$\mathit{keepl} : (A, B) \Leftarrow B$$
$$\mathit{keepl} = \mathit{addl}\ (\lambda(x, y)\ y' \rightarrow x)$$

- similar for *addr*, *keepr*

Drop right element in the view

$$\forall\ f : A \to B.\ eql : A \Leftarrow (A, B)$$
$$put\ x\ (x', y') \mid f\ x' \equiv y' = x$$
$$get\ x = (x, f\ x)$$

- partial *put* and *get*: equality test to guarantee injectivity
- for every pair (x, y), y can be reconstructed from $f\ x$
- similar for *eqr*

Products - Parallel *put* application

Apply two putlenses to both sides of a pair

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

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

Sums - Creating choices

Retrieve a choice from the source

$$\begin{array}{l} \mathit{choice} : \mathit{Either}\ A\ A \Leftarrow A \\ \mathit{put}\ (\mathit{Left}\ x)\ x' = \mathit{Left}\ x' \\ \mathit{put}\ (\mathit{Right}\ x)\ x' = \mathit{Right}\ x' \\ \mathit{get}\ s = \mathit{either}\ \mathit{id}\ \mathit{id}\ s \end{array}$$

Create a choice in the source (conditional)

$$\begin{array}{l} \forall\ p : \mathit{Either}\ A\ A \to A \to \mathit{Bool}.\ p? : \mathit{Either}\ A\ A \Leftarrow A \\ \mathit{put}\ s\ x' \mid \mathit{either}\ \mathit{id}\ \mathit{id}\ s \equiv x' = s \\ \qquad\quad\ \mid \mathit{otherwise} = \mathbf{if}\ p\ s\ x'\ \mathbf{then}\ \mathit{Left}\ x'\ \mathbf{else}\ \mathit{Right}\ x' \\ \mathit{get}\ s = \mathit{either}\ \mathit{id}\ \mathit{id}\ s \end{array}$$

Insert a left/right choice in the source

$$\begin{array}{l} \mathit{inl} : \mathit{Either}\ A\ B \Leftarrow A \\ \mathit{put}\ s\ x' = \mathit{Left}\ x' \\ \mathit{get}\ (\mathit{Left}\ x) = x \end{array}$$

$$\begin{array}{l} \mathit{inr} : \mathit{Either}\ A\ B \Leftarrow B \\ \mathit{put}\ s\ y' = \mathit{Right}\ y' \\ \mathit{get}\ (\mathit{Right}\ y) = y \end{array}$$

Ignore a choice in the view

$$\forall\ f : S \Leftarrow V_1, g : S \Leftarrow V_2.\ f \triangledown g : S \Leftarrow Either\ V_1\ V_2$$
$$put\ s\ (Left\ v_1) = put_f\ s\ v_1$$
$$put\ s\ (Right\ v_2) = put_g\ s\ v_2$$
$$get\ s \mid isJust\ (get_f\ s) \wedge isNothing\ (get_g\ s) = fromJust\ (get_f\ s)$$
$$\mid isNothing\ (get_f\ s) \wedge isJust\ (get_g\ s) = fromJust\ (get_g\ s)$$

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

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

Delete a left/right choice from the view

$$inl^\circ : A \Leftarrow Either\ A\ B$$
$$put\ s\ (Left\ x) = x$$
$$get\ x = Just\ (Left\ x)$$

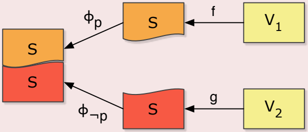
$$inr^\circ : B \Leftarrow Either\ A\ B$$
$$put\ s\ (Right\ y) = y$$
$$get\ y = Just\ (Left\ y)$$

Ignore choice in the view w/ source conditional

$\forall\ p : S \rightarrow Bool, f : S \Leftarrow V_1, g : S \Leftarrow V_2.\ f \triangledown_p g : S \Leftarrow Either\ V_1\ V_2$

$f \triangledown_p g = \phi_p \circ f \triangledown \phi_{\neg\, p} \circ g$

$dom\ f\ s = \mathbf{case}\ get_f\ s\ \mathbf{of}$
$\{ Nothing \rightarrow False$
$; Just\ \_ \rightarrow True \}$



Coreflexive filter

$\forall\ p : A \rightarrow Bool.\ \phi_p : A \Leftarrow A$

$put\ s\ v \mid p\ v = v$

$get\ s = \mathbf{if}\ p\ s\ \mathbf{then}\ Just\ s\ \mathbf{else}\ Nothing$

if-then-else view conditional

$\forall\ p : S \rightarrow V \rightarrow Bool, f : S \Leftarrow V, g : S \Leftarrow V.\ \mathbf{if}\ p\ \mathbf{then}\ f\ \mathbf{else}\ g : S \Leftarrow V$

$\mathbf{if}\ p\ \mathbf{then}\ f\ \mathbf{else}\ g = (f \triangledown_{\phi_{dom\ f}} g) \circ p?$

Sums - Disjoint *put* application

Applies two putlenses to distinct sides of a choice

$$\forall\ f : S_1 \Leftarrow V_1, g : S_2 \Leftarrow V_2.\ f + g : \mathit{Either}\ S_1\ S_2 \Leftarrow \mathit{Either}\ V_1\ V_2$$

$$\begin{aligned}
&put\ (Just\ (Left\ s_1))\ \ (Left\ {v_1}') = Left\ (put_f\ (Just\ s_1)\ {v_1}')\\
&put\ _\ \ (Left\ {v_1}') = Left\ (put_f\ Nothing\ {v_1}')\\
&put\ (Just\ (Right\ s_2))\ (Right\ {v_2}') = Right\ (put_g\ (Just\ s_2)\ {v_2}')\\
&put\ _\ \ (Right\ {v_2}') = Right\ (put_g\ Nothing\ {v_2}')\\
&get\ (Left\ s_1) = liftM\ Left\ (get_f\ s_1)\\
&get\ (Right\ s_2) = liftM\ Right\ (get_g\ s_2)
\end{aligned}$$

- extension (source value creation)

$$\mathbf{data}\ S \Leftarrow V = PutLens\ \{put : Maybe\ S \to V \to S\ ,\ get : S \to Maybe\ V\}$$

Products

$$swap : (B, A) \Leftarrow (A, B)$$

$$assocl : ((A, B), C) \Leftarrow (A, (B, C)) \qquad assocr : (A, (B, C)) \Leftarrow ((A, B), C)$$

Sums

$$coswap : Either\ B\ A \Leftarrow Either\ A\ B$$

$$coassocl : Either\ (Either\ A\ B)\ C \Leftarrow Either\ A\ (Either\ B\ C)$$

$$coassocr : Either\ A\ (Either\ B\ C) \Leftarrow Either\ (Either\ A\ B)\ C$$

Distributivity

$$distl : Either\ (A, C)\ (B, C) \Leftarrow (Either\ A\ B, C)$$

$$distr : Either\ (A, B)\ (A, C) \Leftarrow (A, Either\ B\ C)$$

Algebraic data types

$$in_{[A]} : [A] \Leftarrow Either\ ()\ (A, [A]) \qquad out_{[A]} : Either\ ()\ (A, [A]) \Leftarrow [A]$$

$$nil : [A] \Leftarrow 1, cons : [A] \Leftarrow A, [A] \qquad nil^{\circ} : 1 \Leftarrow [A], cons^{\circ} : A, [A] \Leftarrow [A]$$

$$nil = in_{[A]} \circ inl \qquad nil^{\circ} = inl^{\circ} \circ out_{[A]}$$

$$cons = in_{[A]} \circ inr \qquad cons^{\circ} = inr^{\circ} \circ out_{[A]}$$

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

Language of point-free putlens combinators

$$
\begin{array}{lll}
Put & ::= id \mid Put \circ Put \mid Prod \mid Sum \mid Cond \mid Iso \mid Rec & \\
Prod & ::= addl\ f \mid addr\ f \mid keepl \mid keepr & \text{-- create pairs} \\
 & \mid\ \ eql\ f \mid eqr\ f & \text{-- destroy pairs} \\
 & \mid\ \ Put \times Put & \text{-- product} \\
Sum & ::= choice \mid p? \mid inl \mid inr & \text{-- create choices} \\
 & \mid\ \ Put \mathbin{\triangledown} Put \mid Put \mathbin{\triangledown_p} Put \mid inl^{\circ} \mid inr^{\circ} & \text{-- destroy choices} \\
 & \mid\ \ Put + Put & \text{-- sum} \\
Cond & ::= \phi_p \mid \textbf{if}\ p\ \textbf{then}\ Put\ \textbf{else}\ Put & \text{-- conditional put app.} \\
Iso & ::= swap \mid assocl \mid assocr & \text{-- rearrange pairs} \\
 & \mid\ \ coswap \mid coassocl \mid coassocr & \text{-- rearrange choices} \\
 & \mid\ \ distl \mid distr & \text{-- distr. choices over pairs} \\
Rec & ::= in \mid out \mid \mu(X : Put_X) & \text{-- recursive put}
\end{array}
$$

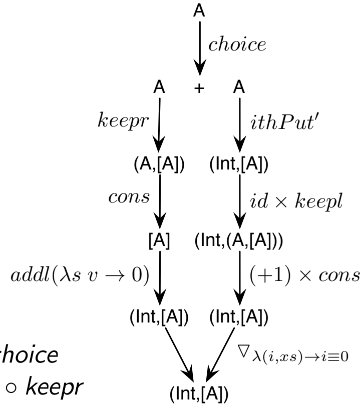
- *get* function

$ith : Int \rightarrow [A] \rightarrow A$
$ith\ 0\ (x : xs) = x$
$ith\ i\ (x : xs) = ith\ (i - 1)\ xs$

- *put*-based lens

$ithPut : Int \rightarrow [A] \Leftarrow A$
$ithPut\ i = eqr\ (const\ i) \circ ithPut'$

$ithPut' : (Int, [A]) \Leftarrow A$
$ithPut' = (zero \nabla_{\lambda(i,xs) \rightarrow i \equiv 0}\ nonzero) \circ choice$
where $zero = addl\ (\lambda s\ v \rightarrow 0) \circ cons \circ keepr$
$\quad nonzero = ((+1) \times cons \circ keepl) \circ ithPut'$

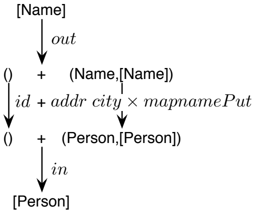



```
get (ithPut 2) "abcde" = Just 'c'
put (ithPut 2) (Just "abcde") 'x' = "abxde"
```

Example (DB projection)

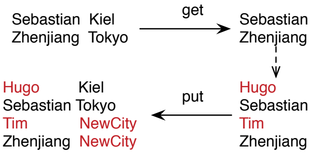
- *get* function

$$\textbf{type}\ Person = (Name, City)$$
$$mapname : [Person] \to [Name]$$
$$mapname\ [\,] = [\,]$$
$$mapname\ ((n, c) : xs) = n : mapname\ xs$$



- *put*-based lens

$$mapnamePut : [Person] \Leftarrow [Name]$$
$$mapnamePut = mapPut\ (addr\ city)$$
$$\quad \textbf{where}\ city\ s\ v = maybe\ \texttt{"NewCity"}\ id\ s$$
$$mapPut : B \Leftarrow A \to [B] \Leftarrow [A]$$
$$mapPut\ f = in \circ (id + f \times mapPut\ f) \circ out$$

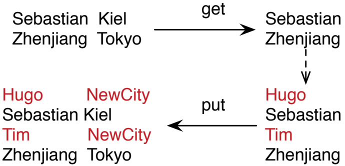


Example (DB projection w/ environment)

- *put*-based lens

$$\mathit{mapnamePut} : [\mathit{Person}] \underset{[\mathit{Person}]}{\Longleftarrow} [\mathit{Name}]$$

$$\begin{aligned}
&\mathit{mapnamePut} = \mathit{mapPut}\ (\mathit{addr}\ \mathit{city}) \\
&\quad \textbf{where}\ \mathit{city}\ \mathit{people}\ n = \\
&\qquad \textbf{case}\ \mathit{lookup}\ n\ \mathit{people}\ \textbf{of} \\
&\qquad\quad \mathit{Just}\ c \to c \\
&\qquad\quad \mathit{Nothing} \to \texttt{"NewCity"}
\end{aligned}$$



- extension (global environment)

$$\begin{aligned}
\textbf{data}\ S \underset{E}{\Longleftarrow} V = \mathit{PutLens}\ \{ & \mathit{put} : (E \to \mathit{Maybe}\ S) \to E \to V \to S \\
& , \mathit{get} : S \to \mathit{Maybe}\ V \}
\end{aligned}$$

$$\mathit{addr} : (E \to A \to B) \to (A, B) \underset{E}{\Longleftarrow} A$$

$$\mathit{local} : S \underset{\mathit{Maybe}\ S}{\Longleftarrow} V \to S \underset{E}{\Longleftarrow} V$$

$$\mathit{local}\ f = f\ \{ \mathit{put}\ \mathit{e2s}\ e\ v = \mathit{put}\ f\ \mathit{id}\ (\mathit{e2s}\ e)\ v \}$$

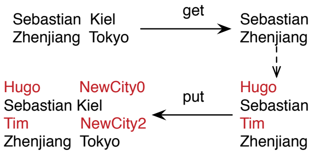
Example (DB projection w/ state)

- *put*-based lens

$$mapnamePut = runST\ (\lambda e\ v \to 0)$$

$$mapnamePutST : [Person] \underset{[Person],Int}{\Longleftarrow} [Name]$$

$$\begin{array}{l} mapnamePutST = mapPut\ \$ \\ \quad updateST\ upd\ (addr\ city) \\ \quad\quad \textbf{where}\ city\ i\ people\ n = \\ \quad\quad\quad\quad \textbf{case}\ lookup\ n\ people\ \textbf{of} \\ \quad\quad\quad\quad\quad Just\ c \to c \\ \quad\quad\quad\quad\quad Nothing \to \texttt{"NewCity"} +\!\!+\ show\ i \\ \quad\quad\quad upd\ i\ e\ s = i + 1 \end{array}$$



- extension (state)

$$\textbf{data}\ S \underset{E,St}{\Longleftarrow} V = PutLens\ \{ put : (E \to Maybe\ S) \to E \to V \to State\ St\ S$$
$$, get : S \to Maybe\ V \}$$

$$runST : (E \to V \to St) \to S \underset{E,St}{\Longleftarrow} V \to S \underset{E}{\Longleftarrow} V$$

$$updateST : (St \to E \to S \to St) \to S \underset{E,St}{\Longleftarrow} V \to S \underset{E,St}{\Longleftarrow} V$$

"Supercompositional" Example (maximum segment sum)

- *get* function

  $$mss : [Int] \to Int$$
  $$mss = maximum \circ map\ sum \circ segments$$

  $$[Int] \xrightarrow{segments} [[Int]] \xrightarrow{map\ sum} [Int] \xrightarrow{maximum} Int$$

- but for *put*...
  1. $put_{map\ sum}$ has to return a consistent list of segments
  2. $put_{maximum}$ has to return a list of sums that correspond to the sums of the updated segments

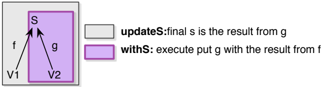
1. decompose segments into a data index/ segments of positions (**type** $Idx\ A = Map\ Pos\ A$)

$$[Int] \xrightarrow{indexes} [(Pos, Int)] \xrightarrow{Map.fromList \times segments \circ map\ \pi_1} (Idx\ Int, [[Pos]])$$

$$(Idx\ Int, [[Pos]]) \xrightarrow{mapsumsegsmax} Int$$

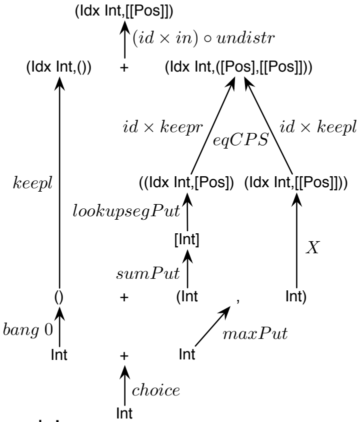
“Supercompositional” Example (maximum segment sum)

❷ $put_{mapsumsegsmax}$ in CPS to keep the data index updated



-- continuation-passing style

$$eqCPS :: S \underset{E,St}{\Longleftarrow} V_1$$
$$\rightarrow S \underset{E,St}{\Longleftarrow} V_2$$
$$\rightarrow S \underset{E,ST}{\Longleftarrow} (V_1, V_2)$$



- extension (*put* “side-effects”)

-- modify the original source before applying *put*

$$withS :: (St \rightarrow E \rightarrow S \rightarrow V \rightarrow S) \rightarrow S \underset{E,St}{\Longleftarrow} V \rightarrow S \underset{E,St}{\Longleftarrow} V$$

-- modify the updated view before applying *put*

$$withV :: (St \rightarrow E \rightarrow V \rightarrow V) \rightarrow S \underset{E,St}{\Longleftarrow} V \rightarrow S \underset{E,St}{\Longleftarrow} V$$

-- modify the updated source after applying *put*

$$updateS :: (St \rightarrow E \rightarrow S \rightarrow S) \rightarrow S \underset{E,St}{\Longleftarrow} V \rightarrow S \underset{E,St}{\Longleftarrow} V$$

Framework

$$\mathbf{data}\ S \underset{E,St}{\Longleftarrow} V = PutLens\ \{put : (E \to Maybe\ S) \to E \to V \to State\ St\ S$$
$$,\ get : S \to Maybe\ S\}$$

Tupled Framework

$$\mathbf{data}\ S \underset{E,St}{\Longleftarrow} V = PutLens\ \{getput : S \to (Maybe\ V, E \to V \to State\ St\ S)$$
$$,\ create : E \to V \to State\ St\ S\}$$

Conclusions

- a point-free put-based BX language
- a *put* specification style dual to specifying *get*
  - users write *put*
  - the combinators provide *get* for free
- "similar" maintainability
  - the combinators encapsulate different *put* behaviors
  - complex *put* behaviors by composition (and using extensions)

+ 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

- prove completeness

Conjecture

Our language can express every well-behaved put function for any get function in the following point-free language.

$$Get ::= \pi_1 \mid \pi_2 \mid \triangle \mid \times \mid inl \mid inr \mid p? \mid \nabla \mid + \mid in \mid out \mid \mu(X : Get_X)$$

- *put*-based recursion patterns
- 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.