# Generic Point-free Lenses

Hugo Pacheco    Alcino Cunha

DI-CCTC, Universidade do Minho

Mathematics of Program Construction (MPC'10)

Quebec - June 22nd 2010

# Motivation

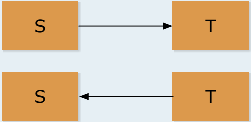
## Unidirectional transformations

- Data transformations abound in software engineering


- Ideally, unidirectional transformations would suffice

## Bidirectional transformations (classical approach)

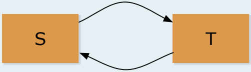
- In real MDSE scenarios, we need to run a transformation backwards




- Manual semantics
- Expensive, error-prone and a maintenance problem

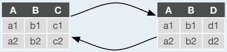
## Bidirectional transformations (better approach)

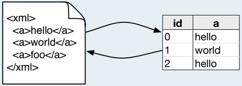
- Derive both from the same specification

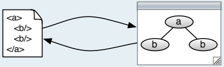


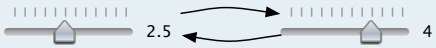

- Clean semantics
- Compositional


## Bidirectional languages exist for...2LT (Two-level Transformation)



# A point-free design

- An application domain

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

- A syntax for combinators

$$id : A \rightarrow A$$
$$\circ : (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$$
$$\pi_1 : A \times B \rightarrow A$$
$$\times : (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow (A \times B \rightarrow C \times D)$$

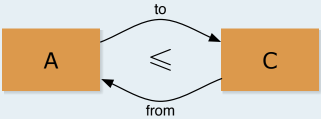
- A set of calculation/simplification laws

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

# What we have just seen

## Refinements

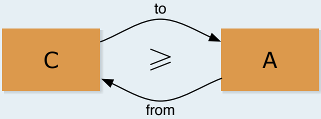
$$\begin{aligned} to &: A \rightarrow C \\ from &: C \rightarrow A \end{aligned}$$




$$from \circ to = id \quad \textsc{Ref}$$

## Abstractions

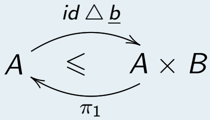
$$\begin{aligned} to &: C \rightarrow A \\ from &: A \rightarrow C \end{aligned}$$



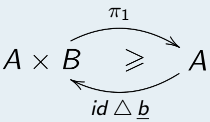

$$to \circ from = id \quad \textsc{Abs}$$

## Add/Drop element

$addR^b : A \leqslant A \times B$

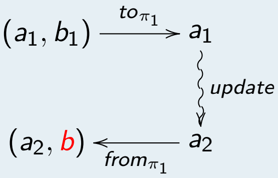



${\pi_1}^b : A \times B \geqslant A$




$$from_{addR} \circ to_{addR} = to_{\pi_1} \circ from_{\pi_1} = \pi_1 \circ (id \bigtriangleup \underline{b}) = id$$

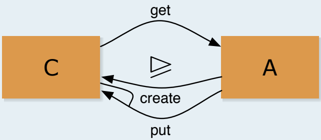
- Updating the abstract value



# A “small” step into lenses

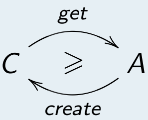
## Stateful abstractions

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



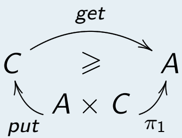

## Properties for well-behaved lenses

- CreateGet



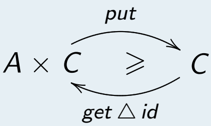

$get \circ create = id$

- PutGet




$get \circ put = \pi_1$

- GetPut




$put \circ (get \mathbin{\triangle} id) = id$

# Projection as a lens

## Drop element

$$\pi_1{}^{b} : A \times B \trianglerighteq A$$

$$\begin{aligned} get &: A \times B \to A \\ get &= \pi_1 \\ create &: A \to A \times B \\ create &= id \bigtriangleup \underline{b} \end{aligned}$$

$$A \times (A \times B) \xrightarrow{put = id \times \pi_2} A \times B$$

## Properties

$$get \circ put = \pi_1 \circ (id \times \pi_2) = \pi_1$$

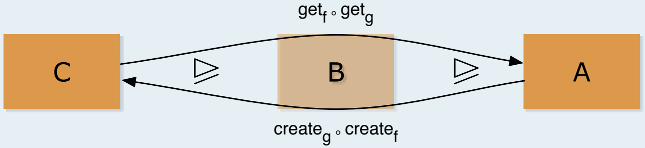
$$put \circ (get \bigtriangleup id) = (id \times \pi_2) \circ (\pi_1 \bigtriangleup id) = \pi_1 \bigtriangleup \pi_2 = id$$

# Composition as a lens

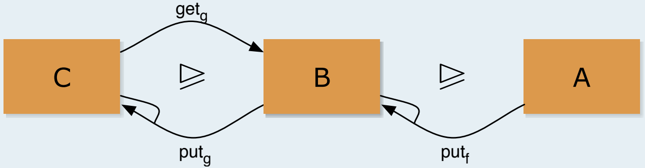
## Lens composition

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

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




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



# More non-recursive lens combinators

## Grammar for combinators

$$
\begin{aligned}
\mathit{Lens} &::= \mathit{id} \mid \mathit{Lens} \circ \mathit{Lens} \mid !^{c} \mid \mathit{Prod} \mid \mathit{Sum} \mid \mathit{Iso} \mid \mathit{Dist} \\
\mathit{Prod} &::= {\pi_1}^{b} \mid {\pi_2}^{a} \mid \mathit{Lens} \times \mathit{Lens} \\
\mathit{Sum} &::= \mathit{Lens} \;{}_\bullet\!\nabla\; \mathit{Lens} \mid \mathit{Lens} \;\nabla_{\!\bullet}\; \mathit{Lens} \mid \mathit{Lens} + \mathit{Lens} \\
&\mid \;\; i_1 \nabla \mathit{Lens} \mid \mathit{Lens} \nabla i_2 \\
\mathit{Iso} &::= \mathit{assocl} \mid \mathit{assocr} \mid \mathit{coassocl} \mid \mathit{coassocr} \\
&\mid \;\; \mathit{swap} \mid \mathit{coswap} \mid \mathit{distl} \mid \mathit{distr}
\end{aligned}
$$

$$
\cdot \;{}_\bullet\!\nabla\; \cdot, \cdot \;\nabla_{\!\bullet}\; \cdot : (A \unrhd C) \to (B \unrhd C) \to (A + B) \unrhd C
$$

## Notable exceptions

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

# How about recursion?

## Some recursive lenses

$length : [A] \trianglerighteq \mathbb{N}$

$$
\begin{aligned}
get\ [\,] &= 0 \\
get\ (x : xs) &= (get\ xs) + 1
\end{aligned}
$$

$plus : \mathbb{N} \times \mathbb{N} \trianglerighteq \mathbb{N}$

$$
\begin{aligned}
get\ (0, m) &= m \\
get\ (n + 1, m) &= get\ (n, m + 1)
\end{aligned}
$$

- *create* is rather easy to define
- A well-behaved definition of *put* is more difficult to obtain

## Question

- Can we provide these definitions for free? Yes

## Hint

- Both *length* and *plus* are easy to define using point-free folds and unfolds
- Good: lensify recursion patterns + reuse combinators

# Cata or fold as a lens

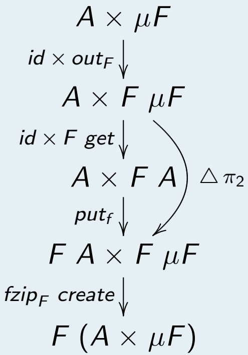
## Catamorphism lens

$$\forall f : F\ A \trianglerighteq A.\ \ \llparenthesis f \rrparenthesis_F : \mu F \trianglerighteq A$$

$$get\ :\ \mu F \rightarrow A$$
$$get = \llparenthesis get_f \rrparenthesis_F$$

$$create\ :\ A \rightarrow \mu F$$
$$create = [\!( create_f )\!]_F$$

$$put\ :\ A \times \mu F \rightarrow \mu F$$
$$put = \llbracket h \rrbracket_F$$
$$h : A \times \mu F \rightarrow F\ (A \times \mu F)$$




## Functor zipping preserves abstract values

$$fzip_F : (A \times C) \rightarrow F\ A \times F\ C \rightarrow F\ (A \times C)$$

$$F\ \pi_1 \circ fzip_F\ f = \pi_1 \qquad \text{FZIP-CANCEL}$$

# Cata or fold as a lens (termination)

## Properties

$get_{(\!|f|\!)} \circ create_{(\!|f|\!)} = id \Leftrightarrow (\!|get_f|\!) \circ [\!(create_f)\!] = id$
$get_{(\!|f|\!)} \circ put_{(\!|f|\!)} = \pi_1 \Leftrightarrow (\!|get_f|\!) \circ [\!(h)\!] = \pi_1$
$put_{(\!|f|\!)} \circ (get_{(\!|f|\!)} \triangle id) = id \Leftrightarrow \ldots$

## Recursive anamorphisms

- Anamorphisms can generate infinite values
- The composition of a cata after an ana (hylo) is not always well-defined and is difficult to reason about

$$(\!|g|\!) \circ [\!(h)\!] \sqsubseteq id \Leftarrow g \circ h = id$$

$$\begin{array}{ccc} \mu F & \xleftarrow{in_F} & F\ \mu F \\ {\scriptstyle [\![h]\!]_F} \uparrow & & \uparrow {\scriptstyle F\ [\![h]\!]_F} \\ A & \xrightarrow[h]{} & F\ A \end{array}$$

- Need anamorphisms that always terminate
  - $h$ well-founded/F-reductive/recursive $\Rightarrow [\![h]\!]$ recursive ana
- Safe composition in SET (recursive hylo uniqueness)

$$(\!|g|\!) \circ [\![h]\!] = f \Leftrightarrow g \circ F\ f \circ h = f$$

# An (extremely) well-behaved case

## Length

- *length* is definable as a catamorphism:

$$length^a = (\!|in_N \circ (id + \pi_2{}^a)|\!)_{L_A} : [A] \trianglerighteq \mathbb{N}$$

- We need to prove that $create_{length}$ and $put_{length}$ are recursive
- However, *length* is also definable as an anamorphism:

$$length^a = [\![(id + \pi_2{}^a) \circ out_{L_A}]\!]_N : [A] \trianglerighteq \mathbb{N}$$

## Natural lens

- A recursive function $f : \mu F \to \mu G$ is a well-behaved lens if there exists a natural transformation $\eta : F \stackrel{.}{\to} G$ such that:

$$f = (\!|in_G \circ \eta|\!)_F = [\![\eta \circ out_F]\!]_G$$

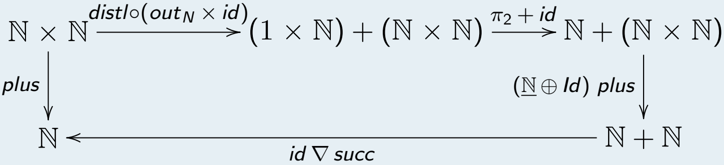
- Good: $\eta$ is a natural lens $\Rightarrow$ termination is guaranteed
- Mapping is another example of a natural lens:

$$map\ f = (\!|in_{L_B} \circ (id + f \times id)|\!) = [\![(id + f \times id) \circ out_{L_A}]\!]$$

## Plus

- *plus* is definable as a recursive hylomorphism:

$$
\begin{aligned}
\mathit{plus} &: \mathbb{N} \times \mathbb{N} \trianglerighteq \mathbb{N} \\
\mathit{plus} &= (\![\mathit{in} \circ (\mathit{out} \nabla i_2)]\!)_{\underline{\mathbb{N}} \oplus \mathit{Id}} \\
&\qquad \circ [\![(\pi_2 + \mathit{id}) \circ \mathit{distl} \circ (\mathit{out} \times \mathit{id})]\!]_{\underline{\mathbb{N}} \oplus \mathit{Id}}
\end{aligned}
$$




- Given that the co-algebras are recursive, a well-behaved lens for *plus* is automatically derived

# Conclusions

## Pros & Cons

+ Construct a bidirectional functional language from standard point-free combinators
+ Support for recursive lenses by using recursion patterns
+ Identify precise termination conditions for bidirectional folds and unfolds

– We cannot discard termination proofs for many recursive lenses
– Not all point-free combinators are well-behaved lenses

## Demo: Haskell++

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

# Future work

- A point-free lens calculus ⇒ bidirectional program calculation
  - lift the point-free laws to lenses:

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

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

  - optimization of complex bidirectional transformations

- Introduce support for data invariants
  - some transformations involve structures of a particular shape
  - can $sort : [A] \to [A]$ be made into a well-behaved lens?

- Provide a better treatment of termination
  - terminating anamorphisms ⇐ well-founded coalgebras
  - link with existing static termination checkers