A Point-free Approach to Bidirectional Transformation

Hugo Pacheco

DI-CCTC, Universidade do Minho, Braga, Portugal

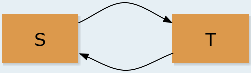
BiG Meeting

Chiba - October 27th 2010

Bidirectional Transformations

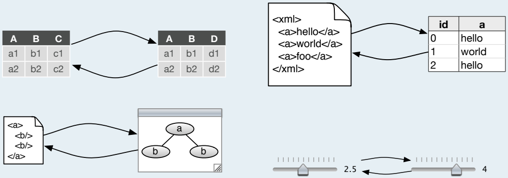
Bidirectional languages

- derive two unidirectional transformations from a specification



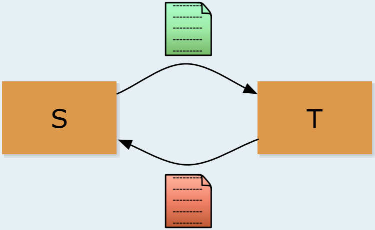
- clean semantics: consistency properties
- compositional

Bidirectional languages exist for...



Motivation - Specification

- bidirectional behaviour is defined by sophisticated procedures



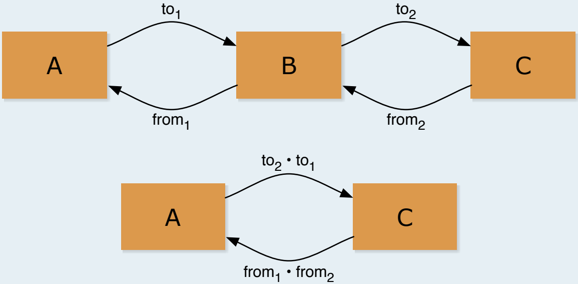
- hard to prove properties about the bidirectional transformations

Goal

- a good framework to prove properties?

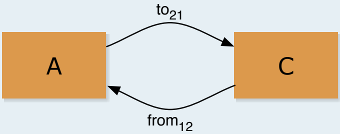
Motivation - Calculation

- compositionality = cluttering



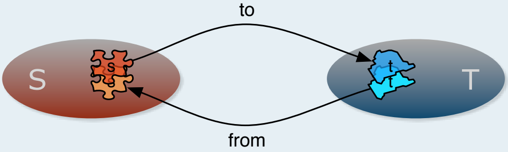
Goal

- 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)

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

- A syntax for combinators

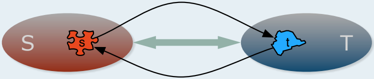
$$id : A \to A$$
$$\circ : (B \to C) \to (A \to B) \to (A \to C)$$
$$\pi_1 : A \times B \to A$$
$$\times : (A \to C) \to (B \to D) \to (A \times B \to C \times D)$$

- A set of calculation/simplification laws

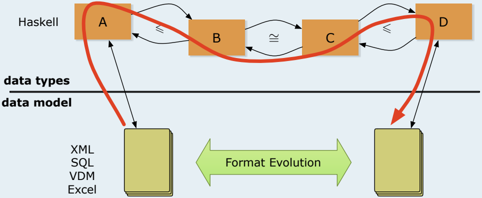
$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\text{-Assoc}$$
$$\pi_1 \circ (f \triangle g) = f \wedge \pi_2 \circ (f \triangle g) = g \qquad \times\text{-Cancel}$$
$$(f \times g) \circ (h \triangle i) = f \circ h \triangle g \circ i \qquad \times\text{-Absor}$$

2LT Framework

- a bidirectional two-level transformation:

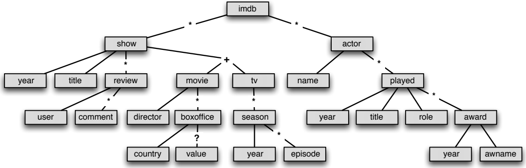


- Two-level Type-level transformation and corresponding value-level transformations.
- an experimental framework for model transformations:

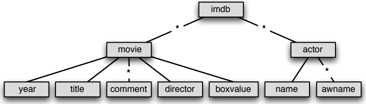


- Type-safe Type-checking guarantees that the data migration functions are well-formed in relation to the transformation

A bidirectional transformation: XML querying



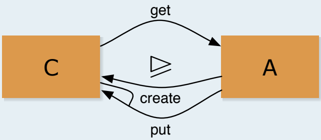
$$
\begin{aligned}
imdb &= shows \times actors \\
shows &= map\ ((id \times reviews) \times id) \circ filter\_l \\
&\quad \circ\ map\ distr \circ map\ (id \times (movie + tv)) \\
reviews &= length \circ concat \circ map\ \pi_{comments} \\
movie &= id \times boxoffices \\
boxoffices &= sum \circ filter\_r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



Data abstraction

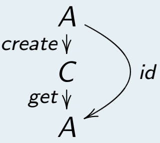
Lenses

$$get \quad : C \rightarrow A$$
$$create : A \rightarrow C$$
$$put \quad : A \times C \rightarrow C$$



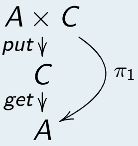
Properties for well-behaved lenses

- CreateGet



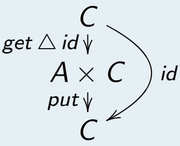
$$get \circ create = id$$

- PutGet



$$get \circ put = \pi_1$$

- GetPut



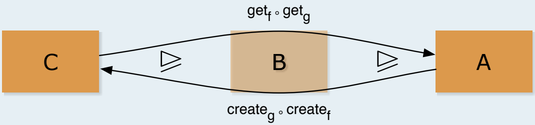
$$put \circ (get \bigtriangleup id) = id$$

Composition as a lens

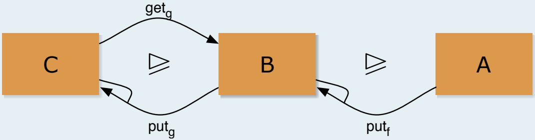
Lens composition

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

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



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



$$id \circ f = f = f \circ id \qquad \text{ID-NAT}$$

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

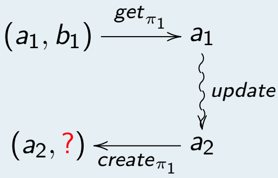
Products - Projection lenses

Drop element

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

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

- Choice of create



Properties

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

Products - More lenses

Split

- $f \triangle g : A \to B \times C$ is not a abstraction $\Leftarrow$ duplication

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

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

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

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

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

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

Injections

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

"Conditional" choice

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

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

- Choice of create

$$C \xrightarrow{p?} C + C \xrightarrow{create_f + create_g} A + B$$

Sum combinator

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

- Choice of put

$$(C + D) \times (A + B)$$

$$\downarrow dists$$

$$C \times A \; + \; C \times B \; + \; D \times A \; + \; D \times B$$

$$\downarrow put_f \quad \downarrow \ldots \quad \downarrow \ldots \quad \downarrow put_g$$

$$\ldots \quad \ldots \quad \ldots \quad \ldots$$

Some laws

$f \circ (g \nabla h)^p = (f \circ g \nabla f \circ h)^{p \circ create_f}$	+-Fusion
$(f \nabla g)^p \circ (h + i)^{j,k} = (f \circ h \nabla g \circ i)^p$	+-Absor
$(id + id)^{f,g} = id$	+-Functor-Id
$(f + g) \circ (h + i) = f \circ h + g \circ i$	+-Functor-Comp
$(f + g)^{j,k} \circ (h + i)^{l,m} = (f \circ h + g \circ i)^{o,p} \Leftrightarrow \ldots$	+-Comp

Recursive lenses

Some recursive lenses

$$\begin{array}{ll} \mathit{length} : [A] \trianglerighteq \mathbb{N} & \mathit{map} : (A \trianglerighteq B) \to [A] \trianglerighteq [B] \\ \mathit{get}\ [\,] = 0 & \mathit{get}\ f\ [\,] = [\,] \\ \mathit{get}\ (x : xs) = (\mathit{get}\ xs) + 1 & \mathit{get}\ f\ (x : xs) = \mathit{get}_f\ x : \mathit{get}\ xs \end{array}$$

- How can we bidirectionalise recursive lenses?

Recursive point-free combinators

$$\begin{array}{ll} \mathit{in}_F : F\ \mu F \trianglerighteq \mu F & \forall f : F\ A \trianglerighteq A.\ (\![f]\!)_F : \mu F \trianglerighteq A \\ \mathit{out}_F : \mu F \trianglerighteq F\ \mu F & \forall f : A \trianglerighteq F\ A.\ [\![f]\!]_F : A \trianglerighteq \mu F \end{array}$$

- Recursive lenses as bidirectional folds and unfolds

"Lensified" examples

$$\begin{array}{ll} \mathit{length}^A : [A] \trianglerighteq \mathbb{N} & \mathit{map} : (A \trianglerighteq B) \to [A] \trianglerighteq [B] \\ \mathit{length}^a = (\![\mathit{in}_N \circ (\mathit{id} + \pi_2{}^{\underline{a}})]\!)_L & \mathit{map}\ f = (\![\mathit{in}_L \circ (\mathit{id} + f \times \mathit{id})]\!)_L \end{array}$$

- Good: reuse existing point-free combinators!

Recursive lenses - Laws

Some recursive laws

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

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

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

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

Some derived laws for lists

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

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

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

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

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

Summary (Lens language)

Grammar for lens combinators

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

Notable exceptions

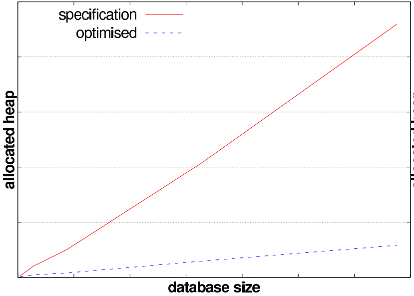
$$
\begin{aligned}
NonLens ::= {} & i_1 : A \to A + B \mid i_2 : B \to A + B \\
\mid \ & \underline{\cdot} : 1 \to B \\
\mid \ & \cdot \triangle \cdot : (A \to B) \to (A \to C) \to (A \to B \times C)
\end{aligned}
$$

Optimisation

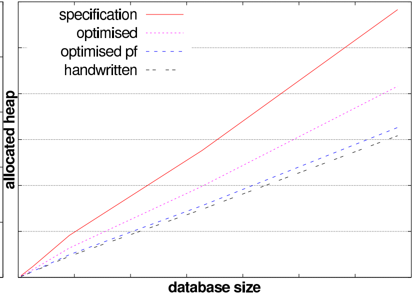
- calculational laws $\Rightarrow$ automated optimisation tool
- how about an handwritten definition?
- simpler example

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

running example



women example



References

Pablo Berdaguer, Alcino Cunha, Hugo Pacheco and Joost Visser
Coupled Schema Transformation and Data Conversion for XML and SQL.
Practical Aspects of Declarative Languages, 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
Submitted, October 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)
− Opaque isomorphism combinators

Further discussion

Currently

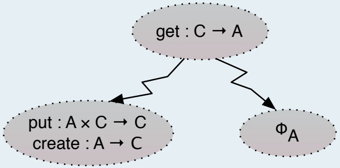
- no full products (π_1,π_2, $\triangle$) or sums (i_1,i_2, ∇)
- counterexample (duplication):

$$id \triangle id : A \trianglerighteq A \times A id \triangle id : A \trianglerighteq A \times A_\phi$$

- why is it not a valid lens (in our setting)?

$$\phi_{A\times A} = \{ (a_1, a_2) \in A\times A \,|\, a_1 = a_2 \}$$

Going relational... (current work)



- total transformations for the restricted domains