Applicative Bidirectional Programming with Lenses

Kazutaka Matsuda (Tohoku University, Japan)
Meng Wang (University of Kent, UK)

August 31st, 2015 @ ICFP 2015
This Talk is about …

- Bidirectional programming f/w that supports **applicative-style** programming with **higher-order** functions

Lens [Foster+07] vs. Our F/W

Lens code:

\[
\text{incTwiceL} = \text{incL} \ast \text{incL}
\]

Our code:

\[
\text{incTwiceF} x = \text{incF} (\text{incF} x)
\]

or

\[
\text{incTwiceF} = \text{twice} \text{ incF}
\]

where

\[
\text{twice} f x = f (f x)
\]
Background: Bidir. Trans.

- a transformation (get) and a translator (put) of updates on the view

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sumii</td>
<td>sumii</td>
<td>205</td>
</tr>
<tr>
<td>Oleg</td>
<td>oleg</td>
<td>203</td>
</tr>
<tr>
<td>Matsuda</td>
<td>kztk</td>
<td>207</td>
</tr>
</tbody>
</table>

get :: Src → View

"Sumii: sumii\nOleg: oleg\nMatsuda: kztk\n"
Background: Bidir. Trans.

- a transformation (get) and a translator (put) of updates on the view

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sumii</td>
<td>sumii</td>
<td>205</td>
</tr>
<tr>
<td>Oleg</td>
<td>oleg</td>
<td>203</td>
</tr>
<tr>
<td>Matsuda</td>
<td>kztk</td>
<td>207</td>
</tr>
</tbody>
</table>

get :: Src → View

"Sumii: sumii
Oleg: oleg
Matsuda: kztk"

↓ update!

"Sumii: sumii
Oleg: oleg
Matsuda: kaz"

[Bancilhon&Spyratos81, Foster+07,...]
Background: Bidir. Trans.

- a transformation (**get**) and a translator (**put**) of updates on the view

<table>
<thead>
<tr>
<th></th>
<th>sumii</th>
<th>205</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sumii</td>
<td>sumii</td>
<td>205</td>
</tr>
<tr>
<td>Oleg</td>
<td>oleg</td>
<td>203</td>
</tr>
<tr>
<td>Matsuda</td>
<td>kztk</td>
<td>207</td>
</tr>
</tbody>
</table>

**get ::** \(\text{Src} \rightarrow \text{View}\)  
"Sumii: sumii\nOleg: oleg\nMatsuda: kztk\n"

**update!**

<table>
<thead>
<tr>
<th></th>
<th>sumii</th>
<th>205</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sumii</td>
<td>sumii</td>
<td>205</td>
</tr>
<tr>
<td>Oleg</td>
<td>oleg</td>
<td>203</td>
</tr>
<tr>
<td>Matsuda</td>
<td>kaz</td>
<td>207</td>
</tr>
</tbody>
</table>

**put ::** \(\text{Src} \rightarrow \text{View} \rightarrow \text{Src}\)  
"Sumii: sumii\nOleg: oleg\nMatsuda: kaz\n"

[Bancilhon&Spyratos81, Foster+07,...]
Applications

- View update problem
  [Bancilhon & Spyratos 81, Hegner 90, Bohannon +06]
- Synchronization of data in different formats
  [Foster +07, Hofmann +11]
- GUI or Web application
  [Hu +04, Hayashi +07, Rajkumar +13]
- Model-driven software development
  [Xiong +07, Yu +12]
Well-Behavedness

- Required for “reasonable” bidir. trans.
  [Bancilhon&Spyratos81, Foster+07, ...]

Acceptability (GetPut)  Consistency (PutGet)
Well-Behavedness

- Required for “reasonable” bidir. trans.  
  [Bancilhon&Spyratos81, Foster+07, ...]

Acceptability (GetPut)  Consistency (PutGet)

No update on the view,  no update on the source

get

put
Well-Behavedness

- Required for “reasonable” bidirectional trans.
  [Bancilhon & Spyrtos 81, Foster + 07, ...]

Acceptability (GetPut)

No update on the view, no update on the source

Consistency (PutGet)

“Put” correctly reflects a view update

Acceptability (GetPut)

Consistency (PutGet)
Bidirectional Language/FW

- Language/framework for bidirectional transformations
  - to guarantee well-behavedness
  - to avoid the maintenance problem of “get” and “put”
Existing F/W: Lens [Foster+07]

- **Lens**: encapsulated pair of “get” and “put”

  ```haskell
  data Lens a b = L { get :: (a -> b),
  put :: (a -> b -> a) }
  ```

- Well-behavedness preserving combinators

  ```haskell
  fstfstL :: Lens ((a,b),c) a
  fstfstL = fstL • fstL
  -- fstL :: Lens (a,b) a

  (•) :: Lens b c -> Lens a b -> Lens a c
  ```
Existing F/W: Lens [Foster+07]

- Lens: encapsulated pair of “get” and “put”
  
  ```haskell
  data Lens a b = L { get :: (a -> b),
                     put :: (a -> b -> a) }
  ```

- Well-behavedness preserving combinators
  
  ```haskell
  -- fstfstL :: Lens ((a,b),c) a
  fstfstL = fstL • fstL
  -- fstL :: Lens (a,b) a
  (•) :: Lens b c -> Lens a b -> Lens a c
  ```

- well-behavedness preserving
Existing F/W: Lens [Foster+07]

- **Lens**: encapsulated pair of “get” and “put”

  ```haskell
data Lens a b = L { get :: (a -> b),
                     put :: (a -> b -> a) }
  ```

- **Well-behavedness preserving combinators**

  ```haskell
fstfstL :: Lens ((a,b),c) a
fstfstL = fstL • fstL               -- fstL :: Lens (a,b) a

(•) :: Lens b c -> Lens a b a -> Lens a c
```

*well-behaved*
Problem

- Programming with unfamiliar combinators in a point-free style

```
fstfstL :: Lens ((a,b),c) a
fstfstL = fstL • fstL
```

- OK for reasoning
- NG for writing and reading

Programming based on compositions w/o variables (cf. Arrows [Hughes00], AoP [Bird&de Moor96])
Our Approach

- To lift lenses to functions

\[ \text{lens} :: \text{Lens} \ (A_1, \ldots, A_n) \ B \]

\[ \text{lift}_n \ \text{lens} :: \ (\text{LT}_s \ A_1, \ldots, \text{LT}_s \ A_n) \rightarrow \text{LT}_s \ B \]

Advantages

- Flexibility of programming style
- More access to the host language in an EDSL impl.
  - Composable by ordinary higher-order functions
LT-type: Bidir Expression

\[
\text{lens} :: \text{Lens} \ (A_1, \ldots, A_n) \ B
\]

\[
\text{lift}_n \ \text{lens} :: (\text{LT} \ s \ A_1, \ldots, \text{LT} \ s \ A_n) \to \text{LT} \ s \ B
\]

\text{lif}_n \ \text{lens} \ (e_1, \ldots, e_n) \text{ constructs an bidirectional expression from bidirectional expressions } e_1, \ldots, e_n.

\text{NB: The construction itself is not bidirectional. (only LT values are bidirectional)}
Our Contributions

- Bidirectional programming f/w
  - *Lenses as functions*
    - Flexibility of programming style
    - More access to the host language in an EDSL impl.
  - Composable by ordinary *higher-order functions*
  - *Well-behavedness by construction*
    - We essentially construct bidir. expressions
  - *Simplified bidirectional programming*
    - similar to programming just “get”
Outlines

‣ Programming in Our F/W
‣ Related Work
‣ Conclusion
Core APIs

data LT s a {- Abstract Type -}

\( \text{lift}_n \) :: Lens \((a_1, \ldots, a_n)\) \(b\) \(\rightarrow\)
\(\rightarrow\) \(\forall s.\) \((LT\ s\ a_1, \ldots, LT\ s\ a_n)\) \(\rightarrow\) \(LT\ s\ b\)

\( \text{unlift}_n \) :: \((\text{Eq} \ a_1, \ldots, \text{Eq} \ a_n)\) \(\Rightarrow\)
\(\Rightarrow\) \((\forall s.\) \((LT\ s\ a_1, \ldots, LT\ s\ a_n)\) \(\rightarrow\) \(LT\ s\ b)\)
\(\rightarrow\) \(\text{Lens} \ (a_1, \ldots, a_n)\) \(b\)

Theorem (embedding)

\( \text{unlift}_n \circ \text{lift}_n = \text{id}_{\text{LT}} \)

Impossible to violate well-behavedness
Example: \texttt{fst}

\begin{verbatim}
fstF :: (LT s a, LT s b) \rightarrow LT s a
fstF (a,b) = a

fstL :: (Eq a, Eq b) \Rightarrow Lens (a,b) a
fstL = \texttt{unlift}_2 \\texttt{fstF}
\end{verbatim}

\begin{verbatim}
*Main> \texttt{get} \texttt{fstL} \ (1,2) \\
1
*Main> \texttt{put} \texttt{fstL} \ (1,2) \ 3 \\
(3,2)
\end{verbatim}

transformation on LT values
Example: unlines (1/2)

Unidirectional unlines

\[
\text{unlines :: [String] } \rightarrow \text{ String} \\
\text{unlines } [] = "" \\
\text{unlines } (x:xs) = \text{catLine } x \ (\text{unlines } xs) \\
\text{where catLine } x \ y = x \ +\ "\\n" +\ y
\]

*Main> unlines ["A","B","C"]
"A\nB\nC\n"
Example: unlines (2/2)

unlinesF :: [LT s String] → LT s String
unlinesF [] = new ""
unlinesF (x:xs) = catLineF x (unlinesF xs)

unlines :: [String] → String
unlines [] = ""
unlines (x:xs) = catLine x (unlines xs)
  where catLine x y = x ++ "\n" ++ y

nullary-lifting

binary-lifting
Example: unlines (2/2)

\[
\begin{align*}
\text{unlinesF} & : [\text{LT s String}] \to \text{LT s String} \\
\text{unlinesF} \; [] & = \text{new } ""
\text{unlinesF} \; (x:xs) & = \text{catLineF} \; x \; (\text{unlinesF} \; xs)
\end{align*}
\]

\[
\begin{align*}
\text{catLineF} & : \text{LT s String} \to \text{LT s String} \to \text{LT s String} \\
\text{catLineF} & = \text{curry} \; (\text{lift}_2 \; \text{catLineL}) \\
& \text{-- catLineL} : \text{Lens} \; \text{(String,String)} \; \text{String}
\end{align*}
\]

\[
\begin{align*}
\text{unlinesL} & : \text{Lens} \; [\text{String}] \to \text{String} \\
\text{unlinesL} & = \text{unliftT} \; \text{unlinesF}
\end{align*}
\]
Example: unlines (2/2)

unlinesF :: [LT s String] \to \ LT s String
unlinesF [] = new ""
unlinesF (x:xs) = catLineF x (unlinesF xs)

\textbf{nullary-lifting}

\textbf{binary-lifting}

catLineF :: LT s String \to LT s String \to LT s String
catLineF = curry (lift_2 catLineL)
  \quad \text{-- catLineL :: Lens (String,String) String}

unlinesL :: Lens [String] \to String
unlinesL = unliftT unlinesF
unlinesF :: [LT s String] → LT s String
unlinesF [] = new ""
unlinesF (x:xs) = catLineF x (unlinesF xs)

catLineF :: LT s String → LT s String → LT s String
catLineF = curry (lift2 catLineL)
  -- catLineL :: Lens (String,String) String
unlinesL :: Lens [String] → String
unlinesL = unliftT unlinesF
unlinesF :: [LT s String] → LT s String
unlinesF []    = new ""
unlinesF (x:xs) = catLineF x (unlinesF xs)
catLineF :: LT s String → LT s String → LT s String
catLineF = curry (lift₂ catLineL)
    -- catLineL :: Lens (String,String) String
unlinesL :: Lens [String] → String
unlinesL = unfoldT unlinesF

*Main> get unlinesL ["A","B","C"]
"A\nB\nC\n"
*Main> put unlinesL ["A","B","C"] "a\nb\ncd\n"
["a","b","cd"]
unlinesF :: [LT s String] → LT s String
unlinesF [] = new ""
unlinesF (x:xs) = catLineF x (unlinesF xs)

catLineF :: LT s String → LT s String → LT s String

---

-- catLineL :: Lens (String,String) String
unlinesL :: Lens [String] → String
unlinesL = unliftT unlinesF

*Main> get unlinesL ["A","B","C"]
"A\nB\nC\n"

*Main> put unlinesL ["A","B","C"] "a\nb\ncd\n"
["a","b","cd"]
*Main> put unlinesL ["A","B","C"] "a\nb\n"

*** Exception: ...

List manipulation itself is not bidirectional
unlinesF :: [LT s String] → LT s String
unlinesF [] = new ""
unlinesF (x:xs) = catLineF x (unlinesF xs)
catLineF :: LT s String → LT s String → LT s String
catLineF = curry (lift2 catLineL)
   -- catLineL :: Lens (String,String) String
unlinesL :: Lens [String] → String
unlinesL = unliftT unlinesF

*Main> get unlinesL ["A","B","C"]
"A\nB\nC\n"
*Main> put unlinesL ["A","B","C"] "a\nb\ncd\n"
["a","b","cd"]
*Main> put unlinesL ["A","B","C"] "a\nb\n"
*** Exception: ...

List manipulation itself is not bidirectional
unlinesF :: [LT s String] → LT s String
unlinesF [] = new ""
unlinesF (x:xs) = catLineF x (unlinesF xs)

catLineF :: LT s String → LT s String → LT s String
catLineF = curry (lift₂ catLineL)

-- catLineL :: Lens (String,String) String
unlinesL :: Lens [String] → String
unlinesL = unliftT unlinesF

*Main> get unlinesL ["A","B","C"]
"A\nB\nC\n"
*Main> put unlinesL ["A","B","C"] "a\nb\ncd\n"
["a","b","cd"]
*Main> put unlinesL ["A","B","C"] "a\nb\n"
*** Exception: ...

List manipulation itself is not bidirectional
unlinesF :: [LT s String] → LT s String
unlinesF = foldr catLineF (new "")

catLineF :: LT s String → LT s String → LT s String
catLineF = curry (\lift_2\ catLineL)
  -- catLineL :: Lens (String,String) String
unlinesL :: Lens [String] → String
unlinesL = \unliftT\ unlinesF

*Main> get unlinesL ["A","B","C"]
"A\nB\nC\n"
*Main> put unlinesL ["A","B","C"] "a\nb\ncd\n"
["a","b","cd"]
*Main> put unlinesL ["A","B","C"] "a\nb\n"
*** Exception: ...

List manipulation itself is not bidirectional
Theorem (guaranteed well-behavedness)

For any

“\( f :: \forall s. (\text{LT } s \ A_1, \ldots, \text{LT } s \ A_n) \rightarrow \text{LT } s \ B \)”

in which lifting functions are applied only to well-behaved lenses, “\( \text{unlift}_n \ f \)” is well-behaved.
Guaranteed Well-Behavedness

Theorem (guaranteed well-behavedness)

For any

\[ f :: \forall s. (\text{LT } s \ A_1, \ldots, \text{LT } s \ A_n) \rightarrow \text{LT } s \ B \]

in which lifting functions are applied only to well-behaved lenses, “\text{unlift}_n \ f” is well-behaved.

by free-theorem [Wadler89, Voigtländer09]

We use the fact that LT is abstract in our f/w.
Topics Not in This Talk

- Realization of LT type
- Observations on LT-values

```hs
data R s a -- Abstract, R s is a monad
liftO :: Eq b => (a -> b) -> LT s a -> R s b
unliftM :: Eq a => (∀s. LT s a -> R s (LT s b)) -> Lens a b
...
```

- To handle practical transformations such as XML queries (cf. [M&W, PPDP13])
- Lifting lens combinator for shape update
- More examples & categorical discussions
Related Work

- Semantic Bidirectionalization
  [Voigtländer 09, M&W13, M&W14]
  - derives well-behaved bidir. trans. only from polymorphic “get”

\[
\text{bff} :: (\forall a. [a] \rightarrow [a]) \\
\rightarrow \forall b. \text{Eq } b \Rightarrow [b] \rightarrow [b] \rightarrow [b]
\]

- Ours subsumes [Voigtländer 09, M&W13]

\[
\text{bff } f = \text{unliftT } (\text{liftT } \text{idL} \circ f)
\]
Related Work

- van Laarhoven lenses [O’Connor11]

\[
\text{Lens } A \to B
\]

\forall f. \text{ Functor } f \Rightarrow (B \to f B) \to (A \to f A)

- No guarantee of well-behavedness with \(\lambda\)-abstractions and applications
- Bidirectional programming is different from programming “get”
Conclusion

- Bidirectional programming f/w
  - *Lenses as functions*
    - Flexibility of programming style
    - More access to the host language in an EDSL impl.
      - Composable by ordinary *higher-order functions*
  - *Well-behavedness by construction*
    - We essentially construct bidir. expressions
  - *Bidirectional programming similar to programming just “get”*

http://hackage.haskell.org/package/app-lens
Example: Evaluator (1/3)

```haskell
data Exp = ENum Int | EInc Exp
  | EVar String
  | EApp Exp Exp
  | EAbs String Exp

data Val a = VNum a | VFun String Exp (Env a)
  deriving (Eq, Show, Functor, Foldable, Traversable)

newtype Env a = Env [(String, Val a)]
  deriving (Eq, Show, Functor, Foldable, Traversable)
```
Example: Evaluator (2/3)

eval :: Exp → Env (LT s Int) → Val (LT s Int)
eval (EEnum i) env = VNum (\new i)
eval (EInc e) env =
  let VNum n = eval e env
  in VNum (\lift incL n)
eval (EFun x e) env = VFun x e env
eval (EApp e1 e2) env =
  let VFun x e env’ = eval e1 env
  in eval e (extend (x,eval e2 env) env’)
eval (EVar x) env = \lookup x env

evalL :: Exp → Lens (Env Int) (Val Int)
evalL e = \unliftT (\lambda env. \liftT idL (eval e env))
Example: Evaluator (3/3)

```haskell
infixl 9 @@ -- @@ is left associative
(@@) = EApp
exp0 = tw @@ tw @@ tw @@ tw @@ inc @@ x

where
  tw = EFun "f" (EFun "x"
    (EVar "f" @@ (EVar "f" (EVar "x"))))
  inc = EFun "x" (EInc (EVar "x"))
  x = EVar "x"

env0 = Env [("x", VNum 3)]

*Main> get (evalLL exp0) env0
VNum 65539
*Main> put (evalLL exp0) env0 (VNum 0)
Env [("x", VNum (-65536))]
```
Example: Evaluator (3/3)

\begin{verbatim}
infixl 9 @@ -- @@ is left associative
(@@) = EApp
exp0 = tw @@ tw @@ tw @@ tw @@ inc @@ x

where
  tw = EFun "f" (EFun "x"
    (EVar "f" @@ (EVar "f" (EVar "x"))))
  inc = EFun "x" (EInc (EVar "x"))
  x = EVar "x"

env0 = Env [("x", VNum 3)]
\end{verbatim}

*Main> get (evalL exp0) env0
VNum 65539
*Main> put (evalL exp0) env0 (VNum 0)
Env [("x", VNum (-65356))]
Example: Evaluator (3/3)

\textbf{infixl 9 @@ -- @@ is left associative}

@@ = EApp

\textbf{exp0 = tw @@ tw @@ tw @@ tw @@ inc @@ x}

\textbf{where}

\begin{align*}
\text{tw} &= \text{EFun "f" (EFun "x" (EVar "f" @@ (EVar "f" (EVar "x")))))} \\
\text{inc} &= \text{EFun "x" (EInc (EVar "x"))} \\
\text{x} &= \text{EVar "x"}
\end{align*}

\textbf{env0 = Env [("x", VNum 3)]}

\textbf{*Main> get (evalL exp0) env0}

\textbf{VNum 65539}

\textbf{*Main> put (evalL exp0) env0 (VNum 0)}

\textbf{Env [("x", VNum (-65536))]}
Key Difference in HO Eval

Naive Approach

\[ [\Gamma \vdash e : A] : \text{Lens} \quad [\Gamma] \quad [A] \]
\[ [B] = B \text{ for base type } B \]

Our Approach

\[ [\Gamma \vdash e : A]_S : [\Gamma]_S \rightarrow [A]_S \]
\[ [B]_S = \text{Lens } S B \text{ for base type } B \]
More Detail (1/3)

- [Voigländer09]
  - maps updatable slots to slots
  - slots must be *untouched* in “get”
[M&W13]
- maps updatable slots to slots
- contents in slots can be observed but not transformed in “get”

get holds “A”?
Ours

- maps updatable slots to slots
- contents in slots can be *observed*, *transformed*, and *merged* in “get”