HOBiT: Programming Lenses without Using Lens Combinators

Kazutaka Matsuda (Tohoku University)  
Meng Wang (University of Bristol)

Apr. 16@ESOP 2018
HOBiT, a Quick Overview

appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] → y with const True by λ_.λ_.[]
  (a:z) → a : appendB z y with not . null by (λ_.λ_.undefined)

- **ML-like programming** for bidirectional transformations (or lenses)
  - with higher-order functions
- **Replacing lens combinators**
  - cf. [Foster+05, 07]
HOBiT, a Quick Overview

- **ML-like programming** for bidirectional transformations (or lenses)
  - with higher-order functions
- **Replacing lens combinators**
  - cf. [Foster+05, 07]

```haskell
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] -> y with const True by λ_.λ_.[]
  (a:z) -> a : appendB z y with not . null by (λ_.λ_.undefined)
```
Background: Bidirectional 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>Kiselyov</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
Kiselyov: oleg
Matsuda: kztk"

[Bancilhon&Spyratos81, Foster+05, 07,...]
**Background: Bidirectional 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>Kiselyov</td>
<td>oleg</td>
<td>203</td>
</tr>
<tr>
<td>Matsuda</td>
<td>kztk</td>
<td>207</td>
</tr>
</tbody>
</table>

\[
\text{get} :: \text{Src} \rightarrow \text{View} \quad \text{"Sumii: sumii\nKiselyov: oleg\nMatsuda: kztk\n"}
\]

\[
\downarrow \text{update!} \quad \text{"Sumii: sumii\nKiselyov: oleg\nMatsuda: kaz\n"}
\]

[Bancilhon&Spyratos81, Foster+05, 07,...]
Background: Bidirectional Trans.

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

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

```text
get :: Src → View
"Sumii: sumii
Kiselyov: oleg
Matsuda: kztk"
```

```text
put :: Src → View → Src
"Sumii: sumii
Kiselyov: oleg
Matsuda: kaz"
```

[Bancilhon&Spyratos81, Foster+05, 07,...]
Well-Behavedness

- Required for “reasonable” bidir. trans.  
  [Bancilhon&Spyratos81, Foster+05,07, ...]

Acceptability (GetPut)  Consistency (PutGet)

No update on the view, no update on the source

“Put” correctly reflects a view update

get

put

get

put

get

update!
Lenses [Foster+05, 07, ...]

- Lens: encapsulated pair of get/put

```haskell
type Lens s v = (s -> v, s -> v -> s)
```

- Well-behavedness preserving combinators

```haskell
fstL :: Lens (a,b) a
fstL = ... {- well behaved -} ...
fstfstL :: Lens ((a,b),c) a
fstfstL = fstL ⊙ fstL
```
Lenses [Foster+05, 07, ...]

- Lens: encapsulated pair of get/put

\[
\text{type } \text{Lens } s \text{ } v = (s \rightarrow v, s \rightarrow v \rightarrow s)
\]

- Well-behavedness preserving combinators

\[
\begin{align*}
\text{fstL} :: & \text{Lens } (a,b) \text{ } a \\
\text{fstL} = & \ldots \{- \text{ well behaved } \} \ldots \\
\text{fstfstL} :: & \text{Lens } ((a,b),c) \text{ } a \\
\text{fstfstL} = & \text{fstL} \bullet \text{fstL}
\end{align*}
\]
Lenses [Foster+05, 07, …]

- **Lens**: encapsulated pair of get/put

  \[
  \text{type } \text{Lens } s \text{ } v = (\text{s } \rightarrow \text{ v, s } \rightarrow \text{ v } \rightarrow \text{ s})
  \]

- **Well-behavedness preserving combinators**

  \[
  \begin{align*}
  \text{fstL} & \colon \text{Lens } (\text{a,b}) \text{ a} \\
  \text{fstL} & = \ldots \text{ \{- well behaved \-} \ldots \\
  \text{fstfstL} & \colon \text{Lens } ((\text{a,b}),\text{c}) \text{ a} \\
  \text{fstfstL} & = \text{fstL} \bullet \text{fstL}
  \end{align*}
  \]
Problem

- Hard to write

appendL :: Lens ([a],[a]) [a]
appendL = cond idL (\_.True) (\_.\_.[])
    (consL • (idL × appendL))
    (not ◦ null) (\_.\_.undefined)

where

rarr :: Lens (Either () (a,b), c)
    (Either c (a,(b,c)))

idL :: Lens a a
consL :: Lens (a,[a]) [a]
outListL :: Lens [a] (Either () (a,[a]))
...

cf.
append x y = case x of
    []     -> y
    (a:z)  -> a : append z y
Problem

- Hard to write

```haskell
appendL :: Lens ([a],[a]) [a]
appendL = cond idL (λ_.True) (λ_.λ_.[])
  (consL • (idL × appendL))
  (not ○ null) (λ_.λ_.undefined)
  • rearr • (outListL × idL)

where
  rearr :: Lens (Either () (a,b), c)
    (Either c (a,(b,c)))

idL :: Lens a a a
consL :: Lens (a,[a]) [a]
outListL :: Lens [a] (Either () (a,[a]))
...
```

Haskell

```haskell
append x y = case x of
  []   -> y
  (a:z) -> a : append z y
```
Problem

- Hard to write

```haskell
appendL :: Lens ([a],[a]) [a]
appendL = cond idL (λ_.True) (λ_.λ_.[])
  (consL • (idL × appendL))
  (not ○ null) (λ_.λ_.undefined)
  • rearr • (outListL × idL)

where
  rearr :: Lens (Either () (a,b), c)
    (Either c (a,(b,c)))
  idL :: Lens a a
  consL :: Lens (a,[a]) [a]
  outListL :: Lens [a] (Either () (a,[a]))
  ...
```
Problem

- Hard to write

Can we fill the gap?

appendL :: Lens ([a],[a]) [a]
appendL = cond idL (\_._.True) (\_._.[[]])
  (consL • (idL × appendL))
  (not ◦ null) (\_._.λ_.undefined)
  • rearr • (outListL × idL)

where
  rearr :: Lens (Either () (a,b), c)
  (Either c (a,(b,c)))
  idL :: Lens a a
  consL :: Lens (a,[a]) [a]
  outListL :: Lens [a] (Either () (a,[a]))
  ...

Haskell

append x y = case x of
  []   -> y
  (a:z) -> a : append z y
cf.
Haskell

Lens in Haskell

fold not fold
Reason of the Complication

append x y = case x of
  []  -> y
  (a:z) -> a : append z y

- "y" is free in the "case"
- Combinators are closed by definition
  ◦ recall: lens is a combinator language
Reason of the Complication

```
append x y = case x of
    []    -> y
    (a:z) -> a : append z y
```

- "y" is free in the "case"
- Combinators are closed by definition
  - recall: lens is a combinator language
Our Solution: HOBiT

- **ML-like** bidirectional language
  - allows *unrestricted use* of *free variables*

```
appendB :: B [a] -> B [a] -> B [a]
appendB x y = case x of
  []    -> y with const True by λ_.λ_.[]
  (a:z) -> a : appendB z y with not . null
             by (λ_.λ_.undefined)
```

- *expressive as the lens framework*
appendB :: B [a] → B [a] → B [a]

appendB x y = \textbf{case} x \textbf{ of}
    [] -> y \textbf{ with} const True \textbf{ by} \lambda_.\lambda_.[]
(a:z) -> a : appendB z y \textbf{ with} not \textbf{ . null} \\by (\lambda_.\lambda_.\text{undefined})

\[ B\sigma \rightarrow B\tau \equiv \text{Lens } \sigma \tau \]
(at the top level)
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] -> y with const True by λ_.λ_.[]
  (a:z) -> a : appendB z y with not . null
             by (λ_.λ_.undefined)

Bσ → Bτ ≡ Lens σ τ
(at the top level)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :get appendBUC ([1], [2,3])
appendB :: B [a] → B [a] → B [a]

appendB x y = case x of
  []   -> y with const True by λ_.λ_[[]]
  (a:z) -> a : appendB z y with not . null by (λ_.λ_.undefined)

Bσ → Bτ ≡ Lens σ τ
(at the top level)

appendBUC :: B ([a], [a]) → B [a]

appendBUC x = let (a,b) = x in appendB a b

HOBiT> :get appendBUC ([1], [2,3])
[1,2,3]
appendB :: B [a] -> B [a] -> B [a]
appendB x y = case x of
  []  -> y with const True by λ_.λ_.[]
(a:z) -> a : appendB z y with not . null
  by (λ_.λ_.undefined)

Bσ → Bτ ≡ Lens σ τ
(at the top level)

appendBUC :: B ([a], [a]) -> B [a]
appendBUC x = let (a,b) = x in appendB a b
appendB :: B [a] → B [a] → B [a]

appendB x y = case x of
  [] -> y with const True by \_.\_.[]
(a:z) -> a :: appendB z y with not . null
  by (\_.\_.undefined)

B σ → B τ ≡ Lens σ τ
(at the top level)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :get appendBUC ([1], [2,3])
[1,2,3]
HOBiT> :put appendBUC ([1], [2,3]) [4,5,6] ([4],[5,6])
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] -> y with const True by λ_.λ_.[]
  (a:z) -> a : appendB z y with not . null
           by (λ_.λ_.undefined)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :put appendBUC ([1], [2,3]) [4,5,6,7]
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] -> y with const True by λ_.λ_.[]
  (a:z) -> a : appendB z y with not . null
    by (λ_.λ_.undefined)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :put appendBUC ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] -> y with const True by \_._.[
  (a:z) -> a : appendB z y with not . null
        by (\_._.undefined)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :put appendBUC ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
HOBiT> :put appendBUC ([1], [2,3]) [4,5]
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] → y with const True by λ_.λ_.[]
  (a:z) → a : appendB z y with not . null by (λ_.λ_.undefined)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :put appendBUC ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
HOBiT> :put appendBUC ([1], [2,3]) [4,5]
([4],[5])
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
    []  -> y with const True by λ_.λ_.[]
(a:z) -> a : appendB z y with not . null
        by (λ_.λ_.undefined)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :put appendBUC ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
HOBiT> :put appendBUC ([1], [2,3]) [4,5]
([4],[5])
HOBiT> :put appendBUC ([1], [2,3]) []
appendB :: B [a] → B [a] → B [a]
appendB x y = case x of
  [] -> y with const True by λ_.λ_.[]
(a:z) -> a : appendB z y with not . null by (λ_.λ_.undefined)

appendBUC :: B ([a], [a]) → B [a]
appendBUC x = let (a,b) = x in appendB a b

HOBiT> :put appendBUC ([1], [2,3]) [4,5,6,7] ([4],[5,6,7])
HOBiT> :put appendBUC ([1], [2,3]) [4,5] ([4],[5])
HOBiT> :put appendBUC ([1], [2,3]) [] ([], [])
Advantages of HOBiT

- **ML-like**
  - familiar programming style

- **Well-behaved**
  - always yielding a well-behaved lens

- **Expressive**
  - at least as the lens framework [Foster+05, 07]
Outline

- Syntax of HOBiT Core
- Semantics
- Expressiveness
- Related Work and Conclusion
Syntax of HOBiT Core

\[ e ::= x \mid \lambda x.e \mid e_1 e_2 \mid \texttt{fix } x.e \]
\[ \mid \text{True} \mid \text{False} \mid [] \mid e_1 : e_2 \]
\[ \mid \texttt{case } e \texttt{ of } \{ p_1 \rightarrow e_1 ; p_2 \rightarrow e_2 \} \]
\[ \mid \texttt{case } e \texttt{ of } \{ p_1 \rightarrow e_1 \texttt{ with } e'_1 \texttt{ by } e''_1 \]
\[ ; p_2 \rightarrow e_2 \texttt{ with } e'_2 \texttt{ by } e''_2 \} \]
Syntax of HOBiT Core

\[ e ::= x \mid \lambda x. e \mid e_1 \cdot e_2 \mid \textbf{fix} \; x. e \mid \text{True} \mid \text{False} \mid [] \mid e_1 : e_2 \mid \text{case } e \text{ of } \{ p_1 \rightarrow e_1; p_2 \rightarrow e_2 \} \]

\[ x \mid \text{True} \mid \text{False} \mid [] \mid e_1 : e_2 \mid \text{case } e \text{ of } \{ p_1 \rightarrow e_1 \; \text{with} \; e'_1 \; \text{by} \; e''_1 \; ; p_2 \rightarrow e_2 \; \text{with} \; e'_2 \; \text{by} \; e''_2 \} \]
Syntax of HOBiT Core

$e ::= x \mid \lambda x.e \mid e_1 \; e_2 \mid \text{fix} \; x.e$

$\mid \text{True} \mid \text{False} \mid [] \mid e_1 : e_2$

$\mid \text{case} \; e \; \text{of} \{ p_1 \rightarrow e_1; p_2 \rightarrow e_2 \}$

$\mid x$}

$\mid \text{True} \mid \text{False} \mid [] \mid e_1 : e_2$

$\mid \text{case} \; e \; \text{of} \{ p_1 \rightarrow e_1 \; \text{with} \; e'_1 \; \text{by} \; e''_1$

$\; ; p_2 \rightarrow e_2 \; \text{with} \; e'_2 \; \text{by} \; e''_2 \}$
Syntax of HOBiT Core

unidirectional part

\[ e ::= x \mid \lambda x.e \mid e_1 \ e_2 \mid \textbf{fix} \ x.e \mid \text{True} \mid \text{False} \mid [] \mid e_1 : e_2 \mid \text{case} \ e \ \textbf{of} \ \{ p_1 \to e_1; \ p_2 \to e_2 \} \]

λs here

bidirectional part

\[ e ::= x \mid \text{True} \mid \text{False} \mid [] \mid e_1 : e_2 \mid \text{case} \ e \ \textbf{of} \ \{ p_1 \to e_1 \ \textbf{with} \ e'_1 \ \textbf{by} \ e''_1 \ \mid ; \ p_2 \to e_2 \ \textbf{with} \ e'_2 \ \textbf{by} \ e''_2 \} \]
(Overly) Simplified HOBiT Core

\[ e ::= \begin{array}{l}
  x \mid \lambda x.e \mid e_1 e_2 \\
  \mid \text{True} \mid \text{False} \\
  \mid \text{let } x = e_1 \text{ in } e_2
\end{array} \]

\[ \lambda s \text{ here} \]

\[ e ::= \begin{array}{l}
  x \mid \text{True} \mid \text{False} \\
  \mid \text{let } x = e_1 \text{ in } e_2
\end{array} \]

\[ \text{bidirectional part} \]
Types

required/ensured by bidir parts

\[ S, T ::= \text{Bool} \mid S \rightarrow T \mid \text{B}\sigma \]
\[ \sigma, \tau ::= \text{Bool} \]

Examples

\[
\begin{align*}
\text{True} : \text{Bool} & \quad \text{OK} \\
\lambda y. \text{let } x = y \text{ in } x : \text{B}\sigma \rightarrow \text{B}\sigma & \quad \text{OK}
\end{align*}
\]

Non Examples

\[
\begin{align*}
\text{let } x = \text{True} \text{ in } x & \quad \text{Bad} \\
\text{let } x = \text{True} \text{ in } \text{True} & \quad \text{Bad}
\end{align*}
\]

(See our paper for typing rules)
Outline

- Syntax of HOBiT Core
- Semantics
- Expressiveness
- Related Work and Conclusion
Staged Semantics

- **Unidirectional** before *get* and *put*

\[
(\lambda x. \text{let } y = x \text{ in } \text{let } z = y \text{ in } z) \; x_0
\]

unidirectional eval. to eliminate \(\lambda\)s

\[
\text{let } z = x_0 \text{ in } z
\]

**first-order** expressions ready for lens (get and put) interpretation

[M&W13, M+10, Hidaka+10]
Unidirectional Evaluation

\[ e \Downarrow E \]

\[ E ::= \lambda x.e \]
\[ | \quad \text{True} \quad | \quad \text{False} \]
\[ | \quad x \quad | \quad \text{True} \quad | \quad \text{False} \]
\[ | \quad \textbf{let} \ x = E_1 \ \textbf{in} \ E_2 \]

defined as usual except:

\[ \]
\[ x \Downarrow x \]
\[ \]
\[ \textbf{let} \ x = e_1 \ \textbf{in} \ e_2 \Downarrow \textbf{let} \ x = E_1 \ \textbf{in} \ E_2 \]
Unidirectional Evaluation

\[ e \downarrow E \]

\[ E ::= \lambda x.e \]
\[ | \quad \text{True} \quad | \quad \text{False} \]
\[ | \quad x \quad | \quad \text{True} \quad | \quad \text{False} \]
\[ | \quad \text{let } x = E_1 \ \text{in} \ E_2 \]

defined as usual except:

\[ e_1 \downarrow E_1 \quad e_2 \downarrow E_2 \]
\[ \text{let } x = e_1 \ \text{in} \ e_2 \downarrow \text{let } x = E_1 \ \text{in} \ E_2 \]
Get/Put Evaluations (1/2)

- Lens between environments and values
  \[ [M&W13, M+10, Hidaka+10] \]

\[
\begin{align*}
\mu \vdash E & \Rightarrow v \\
\mu \vdash x & \Rightarrow \mu(x) \\
\mu \vdash \text{True} & \Rightarrow \text{True}
\end{align*}
\]
\[
\begin{align*}
\mu \vdash v' & \Leftarrow E \vdash \mu' \\
\mu \vdash v' & \Leftarrow x \vdash \{x = v'\} \\
\mu \vdash \text{True} & \Leftarrow \text{True} \vdash {} \}
\end{align*}
\]
Get/Put Evaluations (2/2)

\[ \mu \vdash E \Rightarrow v \]

\[
\mu \vdash E_1 \Rightarrow v \quad \mu[x = v] \vdash E_2 \Rightarrow u \\
\mu \vdash \text{let } x = E_1 \text{ in } E_2 \Rightarrow u
\]

\[ \mu \vdash v \leftarrow E \vdash \mu' \]

\[
\mu \vdash E_1 \Rightarrow v \quad \mu[x = v] \vdash w' \leftarrow E_2 \vdash \mu'[x = v'] \\
\mu \vdash v' \leftarrow E_1 \vdash \mu'' \\
\mu \vdash w' \leftarrow \text{let } x = E_1 \text{ in } E_2 \vdash \mu' \sqcup \mu''
\]

(overly-simplified version)
Correctness

Theorem

Given a closed HOBiT expression of type $\mathbb{B}_\sigma \rightarrow \mathbb{B}_\tau$ we can obtain a well-behaved lens in $\text{Lens } \sigma \rightarrow \tau$

Given $f$, $f \; x_0 \Downarrow E$ and then define:

get $s = v$ if $\{x_0 = s\} \vdash E \Rightarrow v$

put $s \; v = s'$ if $\{x_0 = s\} \vdash v \Leftarrow E \vdash \{x_0 = s'\}$

Well-behavedness is proved by Kripke logical relations
Outline

- Syntax of HOBiT Core
- Semantics
- Expressiveness
  - lifting lenses and lens combinators
- Related Work and Conclusion
Given a lens in $\text{Lens } \sigma \to \tau$, a corresponding function of type $\mathcal{B}\sigma \to \mathcal{B}\tau$ can be added to HOBiT.

\[
\text{incB} :: \mathcal{B}\text{ Int} \to \mathcal{B}\text{ Int} \\
\text{incB} = \text{fromLens} (\lambda x. x + 1) (\lambda \_\_. \lambda y. y - 1)
\]

Similar to [M&W 15]
Lifting Lens Combinators

Property

Given a lens combinator in
\[ \forall s. \text{Lens} (s, \sigma_1) \tau_1 \rightarrow \text{Lens} (s, \sigma_2) \tau_2 \]
a corresponding higher-order function of type
\[ (B\sigma_1 \rightarrow B\tau_1) \rightarrow B\sigma_2 \rightarrow B\tau_2 \]
can be added to HOBiT.

via adding a bidirectional construct

\[ \text{let from} \quad h :: \text{Lens } s \sigma \rightarrow \text{Lens} (s, \sigma) \tau \rightarrow \text{Lens} \ s \tau \]
\[ h \ \text{lens1} \ \text{lens2} = \ \text{lens2} \cdot \ <\text{id}, \ \text{lens1}> \]
\[ \text{case from} \quad \text{a variant of cond} \ [\text{[Foster+05, 07]}] \]
Outline

- Syntax of HOBiT Core
- Semantics
- Expressiveness
- Related Work and Conclusion
Related Work

- 1st-order bidirectional/invertible langs [M&W13, M+10, Hidaka+10]
  - Expressions interpreted as a lens
  - Hard-wired bidirectional constructs
  - Difficult to be higher-order
    - the category of lenses is not closed [Rajkumar+13]
      - no higher-order lenses
Related Work

- Applicative lenses [M&W 15]
  - lifting lenses via Yoneda lemma
  - combinators can be lifted, but with *closedness restriction*

\[
\text{Lens } a \rightarrow b \quad \overset{\text{lift}}{\longrightarrow} \quad \forall s. \text{Lens } s \rightarrow a \rightarrow \text{Lens } s \rightarrow b
\]

\[
\text{Lens } a \rightarrow b \quad \overset{\text{unlift}}{\longleftarrow} \quad \forall s. \text{Lens } s \rightarrow a \rightarrow \text{Lens } s \rightarrow b
\]

\[
\text{Lens } a \rightarrow b \quad \overset{\text{l lift}}{\longrightarrow} \quad \forall s. \text{Lens } s \rightarrow a \rightarrow \text{Lens } s \rightarrow b
\]

\[
\text{Lens } a \rightarrow b \quad \overset{\text{unlift}}{\longleftarrow} \quad \forall s. \text{Lens } s \rightarrow a \rightarrow \text{Lens } s \rightarrow b
\]
Conclusion

- HOBiT: an ML-like bidirectional language
  - more natural-style of programming

```
appendB :: B [a] -> B [a] -> B [a]
appendB x y = case x of
  [] -> y with const True by \_._._.[]
(a:z) -> a : appendB z y with not . null
  by (\_._.undefined)
```

- replacing lens combinators
  - lenses as functions
  - lens combinators as higher-order functions
    ○ via language constructs with binders