# 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)
```

HOBiT

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

# HOBiT, a Quick Overview

HOBiT

```
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]

# Background: Bidirectional Trans.

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

| | | |
|---|---|---|
| Sumii | sumii | 205 |
| Kiselyov | oleg | 203 |
| Matsuda | kztk | 207 |

```
get :: Src → View
```

```
"Sumii: sumii\n
Kiselyov: oleg\n
Matsuda: kztk\n"
```

[Bancilhon&Spyratos81, Foster+05, 07, ...]

# Background: Bidirectional Trans.

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

| | | |
|---|---|---|
| Sumii | sumii | 205 |
| Kiselyov | oleg | 203 |
| Matsuda | kztk | 207 |

```
get :: Src → View
```

```
"Sumii: sumii\n
Kiselyov: oleg\n
Matsuda: kztk\n"
```

update!

```
"Sumii: sumii\n
Kiselyov: oleg\n
Matsuda: kaz\n"
```

[Bancilhon&Spyratos81, Foster+05, 07, …]

# Background: Bidirectional Trans.

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

| | | |
|---|---|---|
| Sumii | sumii | 205 |
| Kiselyov | oleg | 203 |
| Matsuda | kztk | 207 |

```
get :: Src → View
```

```
"Sumii: sumii\n
Kiselyov: oleg\n
Matsuda: kztk\n"
```

↓ update!

```
"Sumii: sumii\n
Kiselyov: oleg\n
Matsuda: kaz\n"
```

```
put ::
 Src → View → Src
```

| | | |
|---|---|---|
| Sumii | sumii | 205 |
| Kiselyov | oleg | 203 |
| Matsuda | **kaz** | 207 |

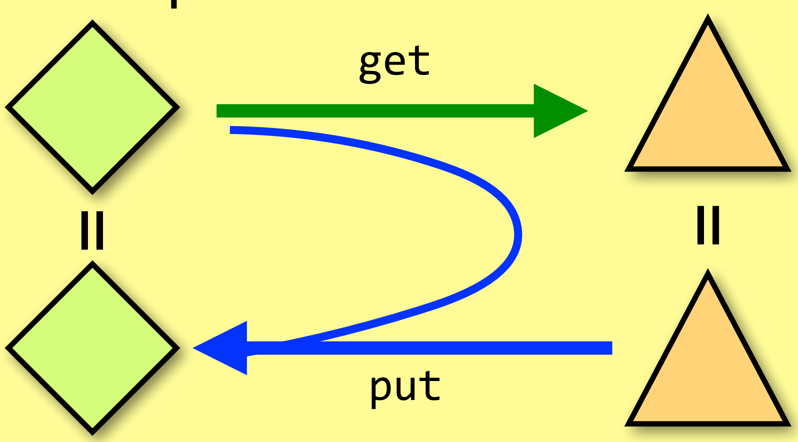
[Bancilhon&Spyratos81, Foster+05, 07, ...]

# Well-Behavedness

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

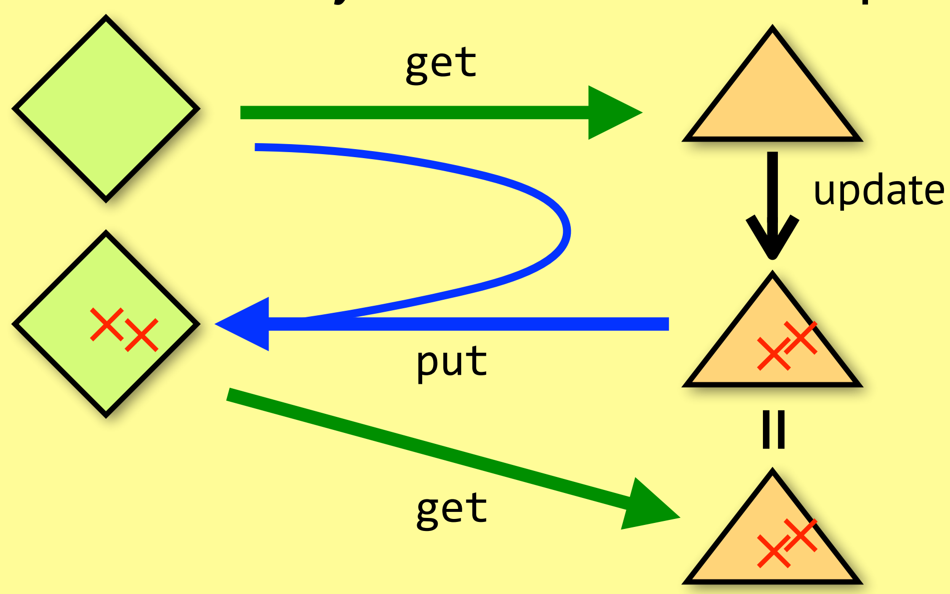
## Acceptability (GetPut)

No update on the view, no update on the source




## Consistency (PutGet)

“Put” correctly reflects a view update



# Lenses [Foster+05, 07, ...]

- Lens: encapsulated pair of get/put

Haskell

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

- Well-behavedness preserving combinators

Lens in Haskell

```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

Haskell

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

- Well-behavedness preserving combinators

Lens in Haskell

```
fstL :: Lens (a,b) a
fstL = ... {- well behaved -} ...
fstfstL :: Lens ((a,b),c) a
fstfstL = fstL ● fstL
```

*well-behavedness preserving*

# Lenses [Foster+05, 07, …]

- Lens: encapsulated pair of get/put

Haskell

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

- Well-behavedness preserving combinators

Lens in Haskell

```
fstL :: Lens (a,b) a
fstL = ... {- well behaved -} ...
fstfstL :: Lens ((a,b),c) a
fstfstL = fstL • fstL
```

*well-behaved*

*well-behavedness preserving*

# Problem

❖ Hard to write

cf.

Haskell

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

Lens in 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

fold

cf.

Haskell

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

❖ Hard to write

Lens in 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

fold

cf.

Haskell

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

- Hard to write

Lens in 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]))
    ...
```

not fold

# Problem

fold

cf.

Haskell

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

❖ Hard to write

*Can we fill the gap?*

Lens in 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]))
    ...
```

not fold

# Reason of the Complication

Haskell

```
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

Haskell

```
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***

HOBiT

```
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 = 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)

HOBiT

```
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)

HOBiT

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

```
HOBiT> :get appendBUC ([1], [2,3])
```

HOBiT

```
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)

HOBiT

```
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

```
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)
```

$$\mathbf{B}\sigma \to \mathbf{B}\tau \equiv \mathtt{Lens}\ \sigma\ \tau$$

(at the top level)

HOBiT

```
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]
```

HOBiT

```
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)
```

$$\mathbf{B}\sigma \to \mathbf{B}\tau \equiv \mathtt{Lens}\ \sigma\ \tau$$

(at the top level)

HOBiT

```
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])
```

HOBiT

```
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)
```

HOBiT

```
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]
```

HOBiT

```
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)
```

HOBiT

```
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

```
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)
```

HOBiT

```
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]
```

HOBiT

```
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)
```

HOBiT

```
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

```
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)
```

HOBiT

```
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]) []
```

HOBiT

```
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)
```

HOBiT

```
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

$$
\begin{array}{rl}
e ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{fix}\ x.e \\
\mid & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid e_1 : e_2 \\
\mid & \mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to e_1; p_2 \to e_2\} \\
\\
\mid & x \\
\mid & \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid e_1 \,\underline{:}\, e_2 \\
\mid & \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_1 \to e_1\ \underline{\mathbf{with}}\ e_1'\ \underline{\mathbf{by}}\ e_1'' \\
& \qquad\qquad ; p_2 \to e_2\ \underline{\mathbf{with}}\ e_2'\ \underline{\mathbf{by}}\ e_2''\}
\end{array}
$$

# Syntax of HOBiT Core

unidirectional part

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{fix}\ x.e$$
$$\mid \mathsf{True} \mid \mathsf{False} \mid [\,] \mid e_1 : e_2$$
$$\mid \mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to e_1; p_2 \to e_2\}$$

bidirectional part

$$\mid \underline{x}$$
$$\mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid e_1 \mathbin{\underline{:}} e_2$$
$$\mid \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_1 \to e_1\ \underline{\mathbf{with}}\ e'_1\ \underline{\mathbf{by}}\ e''_1$$
$$; p_2 \to e_2\ \underline{\mathbf{with}}\ e'_2\ \underline{\mathbf{by}}\ e''_2\}$$

# Syntax of HOBiT Core

*unidirectional part*

$$
\begin{aligned}
e ::= {} & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{fix}\ x.e \\
\mid {} & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid e_1 : e_2 \\
\mid {} & \mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to e_1; p_2 \to e_2\}
\end{aligned}
$$

λs here

*bidirectional part*

$$
\begin{aligned}
\mid {} & \underline{x} \\
\mid {} & \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid e_1 \,\underline{:}\, e_2 \\
\mid {} & \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_1 \to e_1\ \underline{\mathbf{with}}\ e'_1\ \underline{\mathbf{by}}\ e''_1 \\
& \qquad ; p_2 \to e_2\ \underline{\mathbf{with}}\ e'_2\ \underline{\mathbf{by}}\ e''_2\}
\end{aligned}
$$

# Syntax of HOBiT Core

*unidirectional part*

$$\begin{array}{rl} e ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{fix}\ x.e \\ \mid & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid e_1 : e_2 \\ \mid & \mathbf{case}\ e\ \mathbf{of}\ \{p_1 \to e_1; p_2 \to e_2\} \end{array}$$

λs here

*bidirectional part*

$$\begin{array}{rl} \mid & x \\ \mid & \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid e_1 \underline{:} e_2 \\ \mid & \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_1 \to e_1\ \underline{\mathbf{with}}\ e_1'\ \underline{\mathbf{by}}\ e_1'' \\ & \qquad ; p_2 \to e_2\ \underline{\mathbf{with}}\ e_2'\ \underline{\mathbf{by}}\ e_2''\} \end{array}$$

# (Overly) Simplified HOBiT Core

*unidirectional part*

$$e ::= x \mid \lambda x.e \mid e_1\ e_2$$
$$\mid \mathsf{True} \mid \mathsf{False}$$
$$\mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$$

λs here

*bidirectional part*

$$\mid \underline{x}$$
$$\mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}}$$
$$\mid \underline{\textbf{let}}\ x = e_1\ \underline{\textbf{in}}\ e_2$$

# Types

required/ensured by bidir parts

$$S, T ::= Bool \mid S \to T \mid \mathbf{B}\sigma$$

$$\sigma, \tau ::= Bool$$

**Examples**

$\text{True} : Bool$ OK

$\underline{\text{True}} : \mathbf{B}Bool$ OK

$\lambda y.\underline{\mathbf{let}}\ x = y\ \underline{\mathbf{in}}\ x : \mathbf{B}\sigma \to \mathbf{B}\sigma$ OK

**Non Examples**

$\underline{\mathbf{let}}\ x = \text{True}\ \underline{\mathbf{in}}\ x$ Bad

$\underline{\mathbf{let}}\ x = \underline{\text{True}}\ \underline{\mathbf{in}}\ \text{True}$ Bad

(See our paper for typing rules)

# Outline

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

# Staged Semantics

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

$$(\lambda x.\mathbf{let}\ y = x\ \mathbf{in}\ \underline{\mathbf{let}}\ z = y\ \underline{\mathbf{in}}\ z)\ x_0$$

*unidirectional* eval.
to eliminate λs

$$\underline{\mathbf{let}}\ z = x_0\ \underline{\mathbf{in}}\ z$$

***first-order*** expressions ready for
lens (***get*** and ***put***) interpretation
[M&W13, M+10, Hidaka+10]

# Unidirectional Evaluation

$$\boxed{e \Downarrow E}$$

**Residual Exp.**

$$\begin{array}{rcl} E & ::= & \lambda x.e \\ & | & \mathsf{True} \mid \mathsf{False} \\ & | & x \\ & | & \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \\ & | & \underline{\mathbf{let}}\ x = E_1\ \underline{\mathbf{in}}\ E_2 \end{array}$$

defined as usual except:

$$\frac{}{x \Downarrow x} \qquad \frac{e_1 \Downarrow E_1 \quad e_2 \Downarrow E_2}{\underline{\mathbf{let}}\ x = e_1\ \underline{\mathbf{in}}\ e_2 \Downarrow \underline{\mathbf{let}}\ x = E_1\ \underline{\mathbf{in}}\ E_2}$$

# Unidirectional Evaluation

$$\boxed{e \Downarrow E}$$

**Residual Exp.**

$$
\begin{aligned}
E ::=\ & \lambda x.e \\
\mid\ & \mathsf{True} \mid \mathsf{False} \\
\mid\ & x \\
\mid\ & \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \\
\mid\ & \underline{\mathbf{let}}\ x = E_1\ \underline{\mathbf{in}}\ E_2
\end{aligned}
$$

*from B-typed exps.*

defined as usual except:

$$\frac{}{x \Downarrow x} \qquad \frac{e_1 \Downarrow E_1 \quad e_2 \Downarrow E_2}{\underline{\mathbf{let}}\ x = e_1\ \underline{\mathbf{in}}\ e_2 \Downarrow \underline{\mathbf{let}}\ x = E_1\ \underline{\mathbf{in}}\ E_2}$$

# Get/Put Evaluations (1/2)

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

$$\boxed{\mu \vdash E \Rightarrow v} \qquad \boxed{\mu \vdash v' \Leftarrow E \dashv \mu'}$$

$$\frac{}{\mu \vdash x \Rightarrow \mu(x)} \qquad \frac{}{\mu \vdash v' \Leftarrow x \dashv \{x = v'\}}$$

$$\frac{}{\mu \vdash \underline{\mathsf{True}} \Rightarrow \mathsf{True}} \qquad \frac{}{\mu \vdash \mathsf{True} \Leftarrow \underline{\mathsf{True}} \dashv \{\}}$$

# Get/Put Evaluations (2/2)

$$\boxed{\mu \vdash E \Rightarrow v}$$

$$\frac{\mu \vdash E_1 \Rightarrow v \quad \mu[x = v] \vdash E_2 \Rightarrow u}{\mu \vdash \underline{\mathbf{let}}\ x = E_1\ \underline{\mathbf{in}}\ E_2 \Rightarrow u}$$

$$\boxed{\mu \vdash v \Leftarrow E \dashv \mu'}$$

$$\frac{\begin{array}{c}\mu \vdash E_1 \Rightarrow v \quad \mu[x = v] \vdash w' \Leftarrow E_2 \dashv \mu'[x = v'] \\ \mu \vdash v' \Leftarrow E_1 \dashv \mu''\end{array}}{\mu \vdash w' \Leftarrow \underline{\mathbf{let}}\ x = E_1\ \underline{\mathbf{in}}\ E_2 \dashv \mu' \curlyvee \mu''}$$

(overly-simplified version)

# Correctness

**Theorem**

Given a closed HOBiT expression of type `Bσ -> Bτ` we can obtain a well-behaved lens in `Lens σ τ`

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 $\{x_0 = s\} \vdash v \Leftarrow E \dashv \{x_0 = s'\}$

Well-behavedness is proved by Kripke logical relations

# Outline

- Syntax of HOBiT Core
- Semantics
- Expressiveness
  - lifting lenses and lens combinators
- Related Work and Conclusion

# Lifting Lenses

**Property**

Given a lens in `Lens σ τ`,
a corresponding function of type **B**σ → **B**τ
can be added to HOBiT.

```
incB :: B Int → B Int
incB = fromLens (λx.x + 1) (λ_.λy.y - 1)
```

Similar to [M&W 15]

# Lifting Lens Combinators

**Property**

Given a lens combinator in

$\forall s.\ \mathtt{Lens}\ (s,\sigma_1)\ \tau_1 \to \mathtt{Lens}\ (s,\sigma_2)\ \tau_2$

a corresponding higher-order function of type

$(\mathbf{B}\sigma_1 \to \mathbf{B}\tau_1) \to \mathbf{B}\sigma_2 \to \mathbf{B}\tau_2$

can be added to HOBiT.

*via adding a __bidirectional construct__*

e.g.: __**let**__ from

```
h :: Lens s σ → Lens (s,σ) τ → Lens s τ
h lens1 lens2 = lens2 • <id, lens1>
```

__**case**__ from a variant of `cond` [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

$$\texttt{Lens a b} \underset{\texttt{unlift}}{\overset{\texttt{lift}}{\rightleftarrows}} \forall\texttt{s.Lens s a} \to \texttt{Lens s b}$$

  - combinators can be lifted, but with ***closedness restriction***

$$\begin{array}{l}\texttt{Lens a b} \\ \to \texttt{Lens c d}\end{array} \rightleftarrows \begin{array}{l}(\forall\texttt{s.Lens s a} \to \texttt{Lens s b}) \\ \to (\forall\texttt{t.Lens t c} \to \texttt{Lens t d})\end{array}$$

# Conclusion

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

```
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)
```

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