High-Level Languages for Bidirectional Transformations

Experiences and Future Directions

Kazutaka Matsuda
Tohoku University, Japan

This Talk is About …

- A brief introduction of our recent projects on bidirectional transformation languages
  - visit https://bx-lang.github.io/EXHIBIT/

PIs

Kazutaka Matsuda
Tohoku University, Japan

Meng Wang
University of Bristol, UK

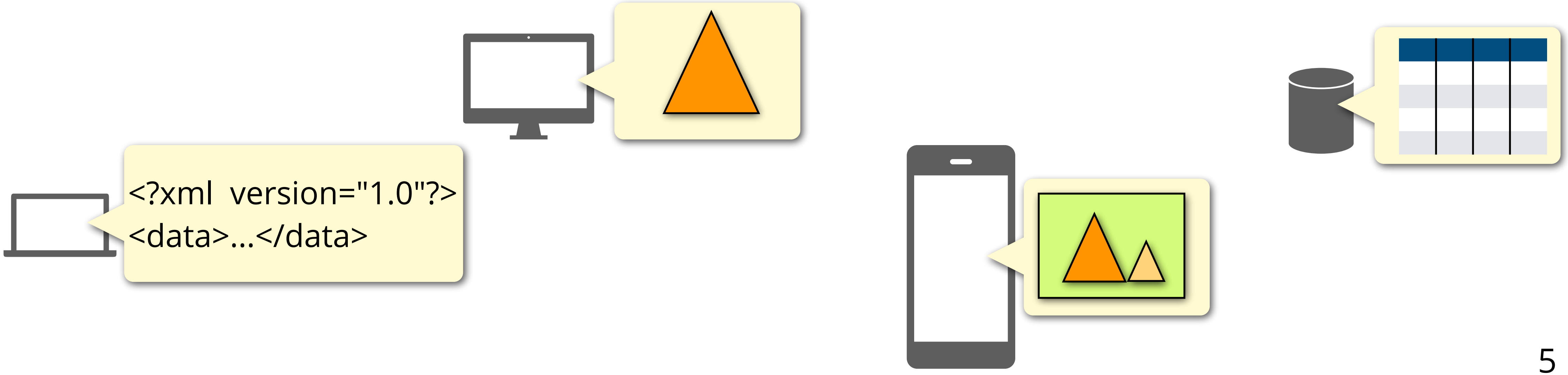
Structure of This Talk

- Background
- A key result
  - HOBiT [M&W ESOP 2018]
- Recent progress
  - SPARCL [M&W ICFP 2020]
- (A few of) future directions

Background

Motivation

- Data that share some information
  - maybe in different formats
  - maybe with non-trivial correspondences
- To keep the shared information in sync



Example of Scenarios

- A classical view updating [Bancilhon&Spyratos 81]

EMP	DEP
Sato	Sales
Suzuki	Dev
Takahashi	Dev

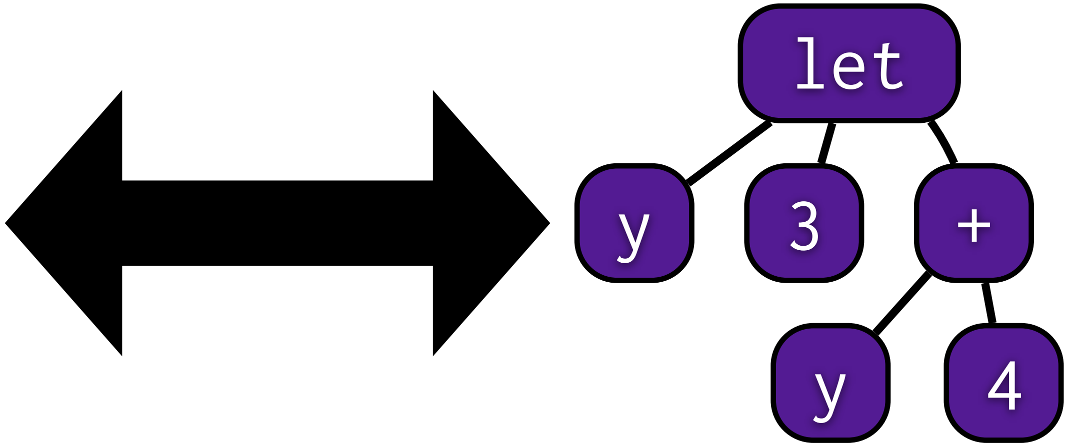
DEP	MGR
Sales	Ito
Dev	Watanabe

EMP	MGR
Sato	Ito
Suzuki	Watanabe
Takahashi	Watanabe

Example of Scenarios

- Program texts and ASTs

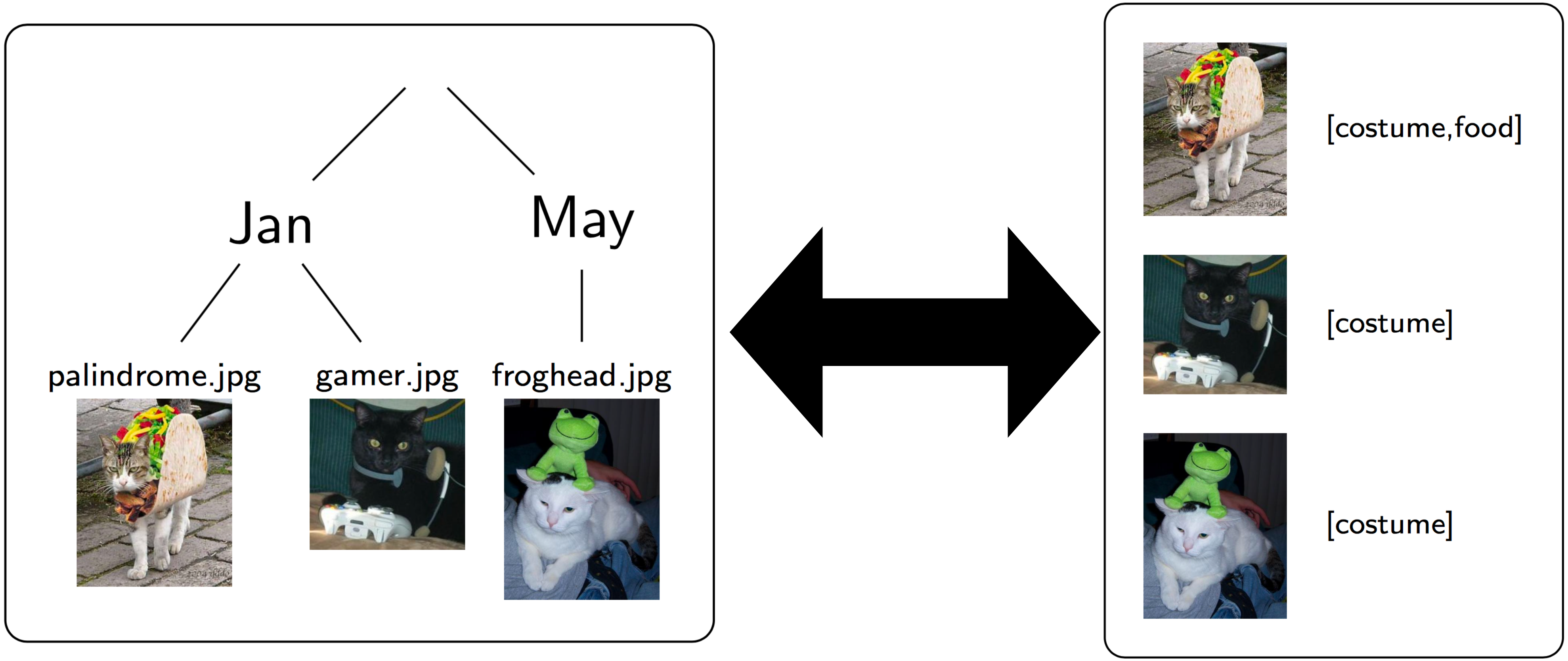
```
-- comment
let y = 3
in y + 4
```



[M&W 13, 18, Danielson 13, Zhu+ 15, 16, ...]

Example of Scenarios

- Folders and lists with tags



http://bx-community.wikidot.com/examples:catpictures

Bidirectional Transformation (BX)
(in a broader sense)

- A mechanism to achieve synchronization of data
  - a couple of transformations
  - change propagators
  - ...
- ... with some laws to hold
  - e.g.: no propagation is triggered for the "sync state"

Ground Goals (in PL research)

- Foundations of BX programming
  - building blocks for BX
  - languages for BX
    - syntax & semantics?
    - type systems?
  - programming techniques in BX languages
- BX scenarios in PL

Ground Goals (in PL research)

- Foundations of BX programming
  - building blocks for BX
  - languages for BX
    - syntax & semantics?
    - type systems?
  - programming techniques in BX languages
- BX scenarios in PL

Our Main Focus

Our Goal

- Design easy-to-use programming languages
  - Slogan: make BX programming more accessible to mainstream programmers

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)
```

Our Goal

- Design easy-to-use programming languages
  - Slogan: make BX programming more accessible to mainstream programmers

***higher-order & functional** (e.g., Haskell, OCaml, ...)*

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)
```

Background

A Key Result

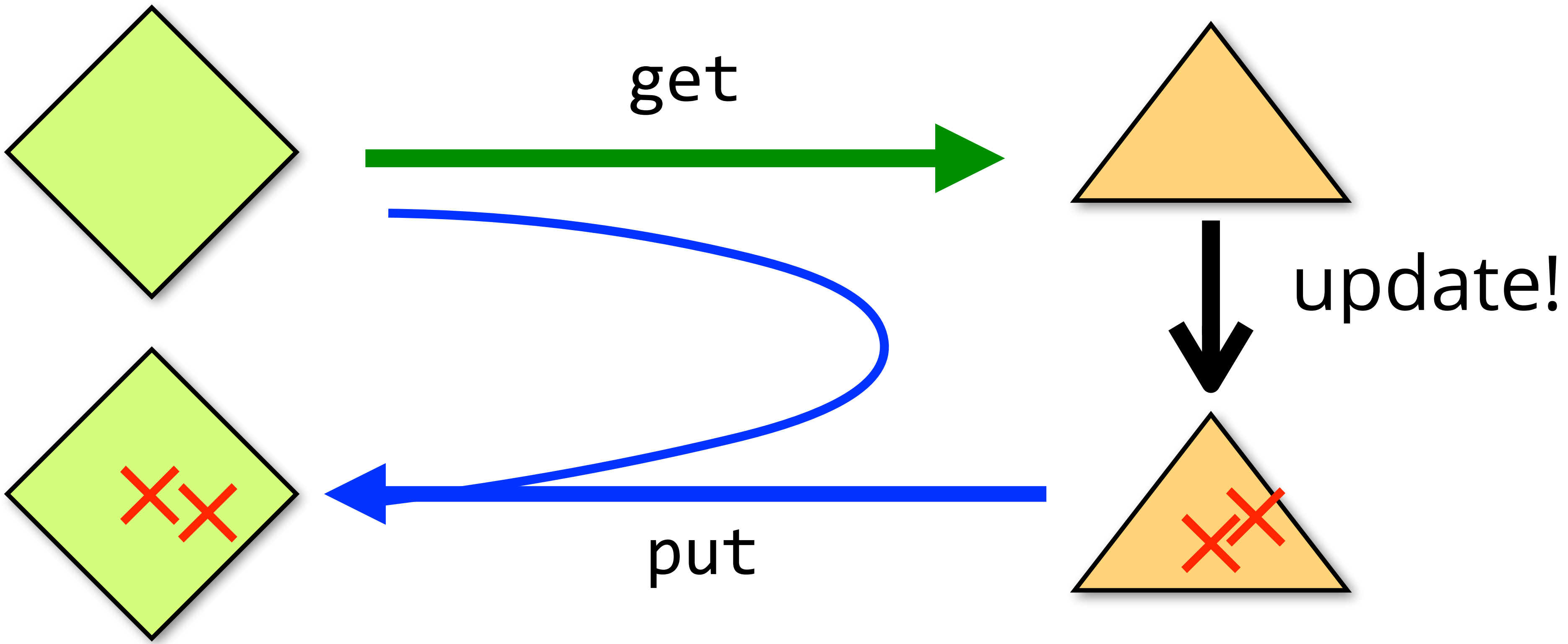
Recent Progress

Future Directions

HOBiT [M&W ESOP 2018]

Background: (Asymmetric) Lenses

- A pair of $\mathsf{get} : S \to V$ and $\mathsf{put} : S \times V \to S$



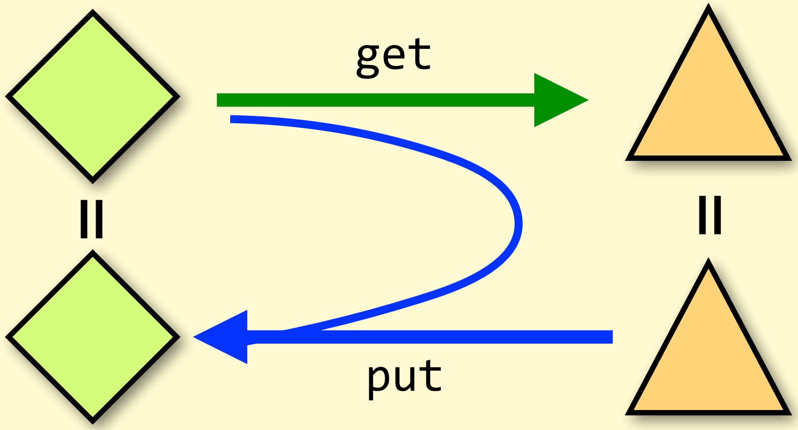
[Bancilhon&Spyratos81, Foster+05, 07, ...]

Well-Behavedness

- Required for “reasonable” BX

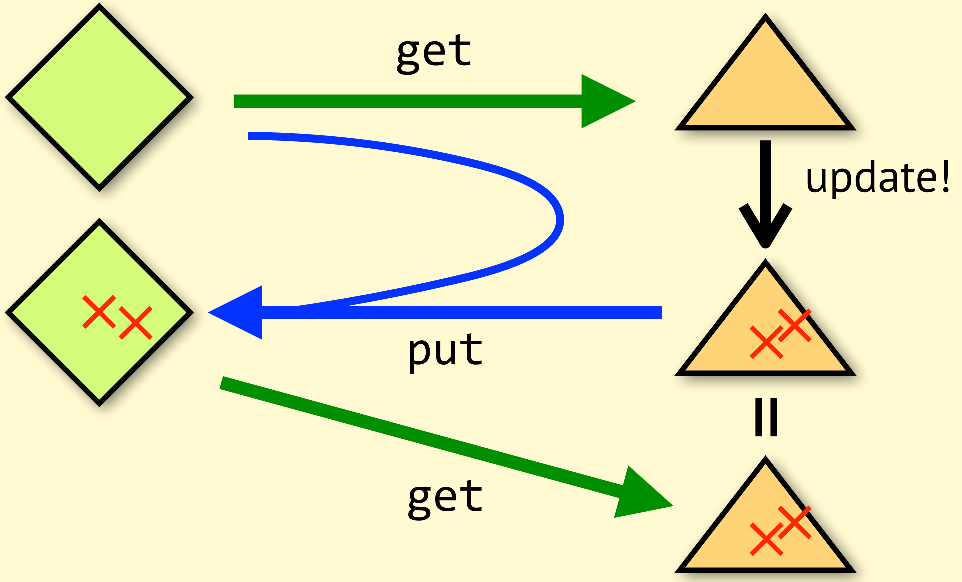
Acceptability (GetPut)

No update on the view,
no update on the source



Consistency (PutGet)

“Put” correctly reflects a view update



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

Background: Lens Programming

- Compose lenses by lens combinators

```
fstL :: Lens (A × B) A
(•)  :: Lens B C -> Lens A B -> Lens A C


fstfstL :: Lens ((A × B) × C) A
fstfstL = fstL • fstL
```

Background: Lens Programming

- Compose lenses by lens combinators

well-behaved

```
fstL :: Lens (A × B) A
(•)  :: Lens B C -> Lens A B -> Lens A C


fstfstL :: Lens ((A × B) × C) A
fstfstL = fstL • fstL
```

Background: Lens Programming

- Compose lenses by lens combinators

well-behaved

```
fstL :: Lens (A × B) A
(•)  :: Lens B C -> Lens A B -> Lens A C
```

well-behavedness preserving

```
fstfstL :: Lens ((A × B) × C) A
fstfstL = fstL • fstL
```

Background: Lens Programming

- Compose lenses by lens combinators

well-behaved

```
fstL :: Lens (A × B) A
(•)  :: Lens B C -> Lens A B -> Lens A C
```

well-behavedness preserving

```
fstfstL :: Lens ((A × B) × C) A
fstfstL = fstL • fstL
```

well-behaved by construction

Problem: Lens Programming is Hard

- Programs get complicated quickly

```
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]))
    ...
```

Existing Approaches

- Bidirectionalization
  [M+ 07, Voigtländer 09, Voigtländer+ 10, 13, M&W 13, 15]
  - derives lenses from a program of "get"
- Inductive programming [Maina+ 18, Miltner+ 18, 19]
  - synthesizes lenses from I/O examples
- Applicative lenses [M&W 15]
  - represents `Lens S V` as `∀s. Lens s S -> Lens s V`
  - not expressive enough

Challenge

- Programming lenses without using lens combinators
  - keeping the good properties of lenses
    - ***compositional reasoning*** and ***expressiveness***
  - based on ***applications*** and variables
    (i.e., applicative programming)
    - ***higher-order***, in particular
    - but, lenses are ***not*** functions

HOBiT [M&W 2018]

- A ***higher-order*** bidirectional programming language
  - lenses as functions
  - lens combinators as higher-order functions
  - supporting ***applicative programming style***

```
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 [M&W 2018]

- A ***higher-order*** bidirectional programming language
  - lenses as functions
  - lens combinators as higher-order functions
  - supporting ***applicative programming style***

```
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)
```

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

```
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 → **B**τ ≡ Lens σ τ (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)

```
appB :: B ([a] × [a]) → B [a]
appB 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 λ_.λ_.[]
 (a:z) -> a : appendB z y with not . null
                          by (λ_.λ_.undefined)
```

Bσ → Bτ ≡ Lens σ τ (at the top level)

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

```
HOBiT> :get appB ([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 → **B**τ ≡ Lens σ τ (at the top level)

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

```
HOBiT> :get appB ([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 → **B**τ ≡ Lens σ τ (at the top level)

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

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

Bσ → Bτ ≡ Lens σ τ (at the top level)

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

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

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

```
HOBiT> :put appB ([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)
```

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

```
HOBiT> :put appB ([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 λ_.λ_.[]
 (a:z) -> a : appendB z y with not . null
                              by (λ_.λ_.undefined)
```

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

```
HOBiT> :put appB ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
HOBiT> :put appB ([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)
```

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

```
HOBiT> :put appB ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
HOBiT> :put appB ([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)
```

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

```
HOBiT> :put appB ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
HOBiT> :put appB ([1], [2,3]) [4,5]
([4],[5])
HOBiT> :put appB ([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)
```

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

```
HOBiT> :put appB ([1], [2,3]) [4,5,6,7]
([4],[5,6,7])
HOBiT> :put appB ([1], [2,3]) [4,5]
([4],[5])
HOBiT> :put appB ([1], [2,3]) []
([], [])
```

Advantages of HOBiT

- ***Applicative style***
  - familiar programming style
- ***Correctness by construction***
  - always yielding well-behaved lenses
- ***Expressiveness***
  - at least as the lens framework [Foster+ 05, 07]
    - lenses as functions
    - lens combinators as higher-order functions

Outlines (of HOBiT introduction)

- Syntax of HOBiT Core
- Semantics
- Properties

Syntax of HOBiT Core

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

Syntax of HOBiT Core

unidirectional part

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

BX 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

λs here

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

BX part

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

Syntax of HOBiT Core

unidirectional part

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

λs here

BX part

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

Types

required/ensured by BX parts

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

$$\sigma, \tau ::= Bool \mid [\sigma]$$

Examples

$\mathrm{True} : Bool$ OK

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

$\lambda y.\underline{\mathbf{case}}\ y\ \underline{\mathbf{of}}\ \{x \to x\} : \mathbf{B}\sigma \to \mathbf{B}\sigma$ OK

Non Examples

$\underline{\mathbf{case}}\ \mathrm{True}\ \underline{\mathbf{of}}\ \{x \to x\}$ Bad

$\underline{\mathbf{case}}\ \underline{\mathrm{True}}\ \underline{\mathbf{of}}\ \{x \to \mathrm{True}\}$ Bad

Outlines

- Syntax of HOBiT Core
- Semantics
- Properties

Staged Evaluation (inspired by [Moggi 98])

- *Unidirectional* before ***get***/***put***

$$(\lambda f.\lambda y.f\ y)\ (\lambda x.x \mathbin{\underline{:}} \underline{[\,]})\ x_0 \quad : \mathbf{B}[Bool]$$

unidirectional eval. to eliminate λs

- BX parts are treated as constructors

$$x_0 \mathbin{\underline{:}} \underline{[\,]}$$

only BX parts remain

1st-order: ready for lens (***get***/***put***) interp.

$$\{x_0 \mapsto \mathsf{True}\} \vdash x_0 \mathbin{\underline{:}} \underline{[\,]} \Rightarrow [\mathsf{True}]$$

$$\{x_0 \mapsto \mathsf{True}\} \vdash [\mathsf{False}] \Leftarrow x_0 \mathbin{\underline{:}} \underline{[\,]} \dashv \{x_0 \mapsto \mathsf{False}\}$$

Outlines

- Syntax of HOBiT Core
- Semantics
- Properties

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

Lifting Lenses

Property

Given a well-behaved 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 the applicative lens [M&W 15]

Lifting Lens Combinators

Property

Given a well-behavedness preserving lens combinator

$\forall s.\ \texttt{Lens}\ (s \times \sigma_1)\ \tau_1 \rightarrow \texttt{Lens}\ (s \times \sigma_2)\ \tau_2$

a corresponding higher-order function of type

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

can be added to HOBiT.

via adding a ***bidirectional construct***

case from a variant of cond [Foster+ 05, 07]

Summary

- HOBiT: a higher-order bidirectional language
  - ***in familiar (i.e., applicative) programming style***

```
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***

SPARCL [M&W ICFP 2020]

(A Very Brief Introduction)

Background

↓

A Key Result

↓

Recent Progress

↓

Future Directions

Motivation (in the context of BX programming)

- We want to convert **B**-typed values to non-**B**-typed ones
  - Everything bidirectional has type **B** in HOBiT

```
appendB :: B [a] → B [a] → B [a]
```

  - but, some functions requires non-**B** values

```
incByB :: Nat -> B Nat → B Nat
```

Example: Huffman Encoding

```
huff :: B [Symbol] -> B (HuffTable × [Bit])
huff s = ???

makeHuff :: [Symbol] -> HuffTable
encode   :: HuffTable -> B [Symbol] -> B [Bit]
```

- Construction of the Huffman encoding table is easier to implement with non-**B** types
- Bidirectional encoding is easier to implement with non-**B** typed Huffman encoding tables

Sparcl [M&W 20]

- A programming language ...
  - for writing ***invertible*** functions
  - through composing ***partially-invertible*** functions

    being invertible after fixing some arguments
    (e.g., addition, multiplication, Huffman encoding)

  - supported by ***linear types***
    - a key to ***correctness by construction***

SPARCL, compared with HOBiT

- Focuses on bijective lenses
- Uses linear types
  - discarding of variables should be prohibited
    - cf. f x = 42
  - based on $\lambda^{q}_{\rightarrow}$ [Bernardy+ 18] with inference [M 20]
    - no syntactic overhead
- Has ***the pin operator***

```
pin :: B S ⊸ (S -> B T) ⊸ B (S ⊗ T)
```

huff in SPARCL (follows HOBiT's syntax)

```
huff :: B [Symbol] ⊸ B (HuffTable ⊗ [Bit])
huff s =
  let (s,h) = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)

new      :: (a -> a -> Bool) -> a -> B a
makeHuff :: [Symbol] -> HuffTable
encode   :: HuffTable -> B [Symbol] ⊸ B [Bit]
pin      :: B s ⊸ (s -> B t) ⊸ B (s ⊗ t)
```

huff in SPARCL (follows HOBiT's syntax)

```
huff :: B [Symbol] ⊸ B (HuffTable ⊗ [Bit])
huff s =                              :: [Symbol]
  let (s,h) = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)

new      :: (a -> a -> Bool) -> a -> B a
makeHuff :: [Symbol] -> HuffTable
encode   :: HuffTable -> B [Symbol] ⊸ B [Bit]
pin      :: B s ⊸ (s -> B t) ⊸ B (s ⊗ t)
```

huff in SPARCL (follows HOBiT's syntax)

```
huff :: B [Symbol] ⊸ B (HuffTable ⊗ [Bit])
huff s = :: B HuffTable      :: [Symbol]
  let (s,h) = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)

new      :: (a -> a -> Bool) -> a -> B a
makeHuff :: [Symbol] -> HuffTable
encode   :: HuffTable -> B [Symbol] ⊸ B [Bit]
pin      :: B s ⊸ (s -> B t) ⊸ B (s ⊗ t)
```

huff in SPARCL (follows HOBiT's syntax)

```
huff :: B [Symbol] ⊸ B (HuffTable ⊗ [Bit])
huff s = 
  let (s,h) = pin s (λs'.new eqHuff (makeHuff s'))
  in pin h (λh'. encode h' s)
```

:: B HuffTable

:: [Symbol]

:: HuffTable

```
new      :: (a -> a -> Bool) -> a -> B a
makeHuff :: [Symbol] -> HuffTable
encode   :: HuffTable -> B [Symbol] ⊸ B [Bit]
pin      :: B s ⊸ (s -> B t) ⊸ B (s ⊗ t)
```

A few of Future Directions

Background

↓

A Key Result

↓

Recent Progress

↓

Future Directions

Unified Framework

- BX languages share ideas
  - many variant of lenses (asymmetric, bijective, ...)
  - functional representations for different variants

$\mathbf{B}_{\mathtt{asym}}$ `S -> ` $\mathbf{B}_{\mathtt{asym}}$ `T` $\mathbf{B}_{\mathtt{bij}}$ `S ⊸ ` $\mathbf{B}_{\mathtt{bij}}$ `T`

- Unify them so that we can reuse programs/ecosystems
  - qualified typing [Jones 95, Vytiniotis+11] helps?

```
Asym k => k S ⊸ k T
```

```
Bij k => k S ⊸ k T
```

```
Bij k => Asym k
```

Integration to Main-Stream Systems

- Approach 1: embedded implementation

  language constructs as (higher-order) functions

  - unembedding [Atkey 09, Atkey+09] would help
    - cf. embedded FliPpr [M&W 18]
- Approach 2: compilation
  - compiling into main-stream languages
  - Issue: how to preserve types?
    - difficulty: staged semantics

Synthesis/Inductive Programming

- HOBiT programming is easier, but still requires effort
  ⇒ synthesize HOBiT programs from examples and "get"

  - Our current team members:
    - Me
    - Meng Wang (University of Bristol)
    - Cristina David (University of Bristol)
    - Masaomi Yamaguchi (Student in Tohoku University)

Other Future Directions

- Better inference of linear types
- Using refinement types
  - hint for acceptable updates
  - for "with" conditions
- Programming techniques
- Multiple views
  - **B** S represents updatable artifacts of type S

Conclusion

- Experiences and future directions on high-level bidirectional programming languages
  - Slogan: make BX programming more accessible to mainstream (functional) programmers

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)
```