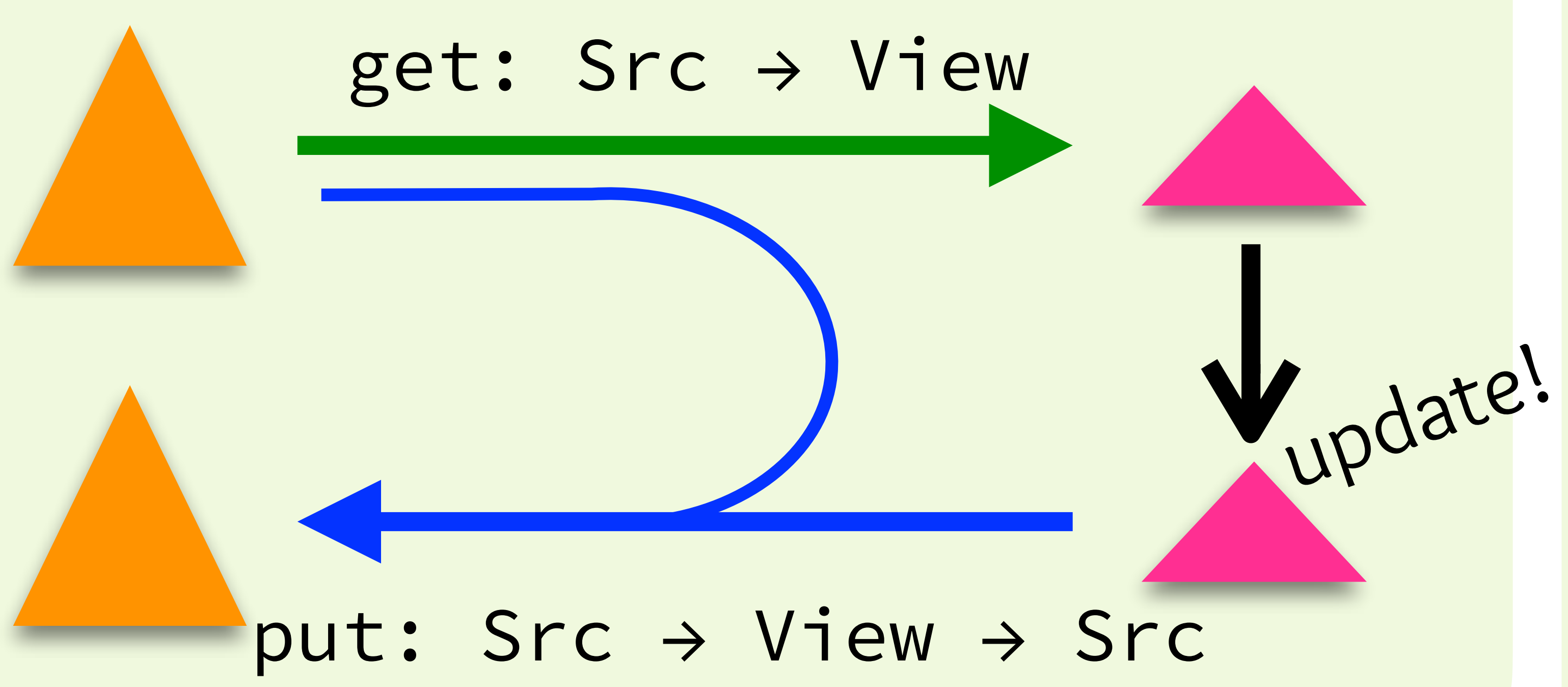
HOBiT

A Higher-Order Language that Bridges Uni- and Bi-directional Programming

Kazutaka Matsuda (Tohoku University) and Meng Wang (University of Kent)

Background: Bidir. Trans. (BX)



get: Src → View

update!

put: Src → View → Src

Motivation

- Language to guarantee ***well-behavedness*** [Bancilhon&Spyratos 81, Foster+07, …]
- ***Applicative*** bidir. programming with ***higher-order*** func. (cf. lens [Foster+07])
- ***Control*** over how updates will be translated (cf. [M&W 15])

Programming Language HOBiT

Core Syntax

```
P ::= x₁ = e₁ … xₙ = eₙ
e ::= x | λx.e | e₁ e₂ | True | False | [] | e₁ : e₂
    | case e of { p₁ -> e₂; x₂ -> e₂ }
    | Trueʙ | Falseʙ | []ʙ | e₁ :ʙ e₂
    | caseʙ e of { p₁ -> e₁ with e₁';
                   x₂ -> e₂ with e₂'}
```

- A simple type system with the unary type constructor **B**
  - τ of **B**τ must not contain "→" and "**B**"
  - The result type of `case`ʙ must be of **B**-type

`B [String]`	Updatable lists of strings
`[B String]`	Lists of updatable strings
`[String]`	Lists of strings

  - "**B**σ→**B**τ"-typed programs represent well-behaved BX between σ and τ (correctness)

Denotational Semantics (Sketch)

Δ is for variables introduced by **case**ʙ

$$[\![\Gamma;\Delta\vdash e:\tau]\!] \in \prod_{S\in\mathsf{Tuple}}(\mathsf{BX}\ S\ [\![\Delta]\!] \to [\![\Gamma]\!]_S \to [\![\tau]\!]_S)$$

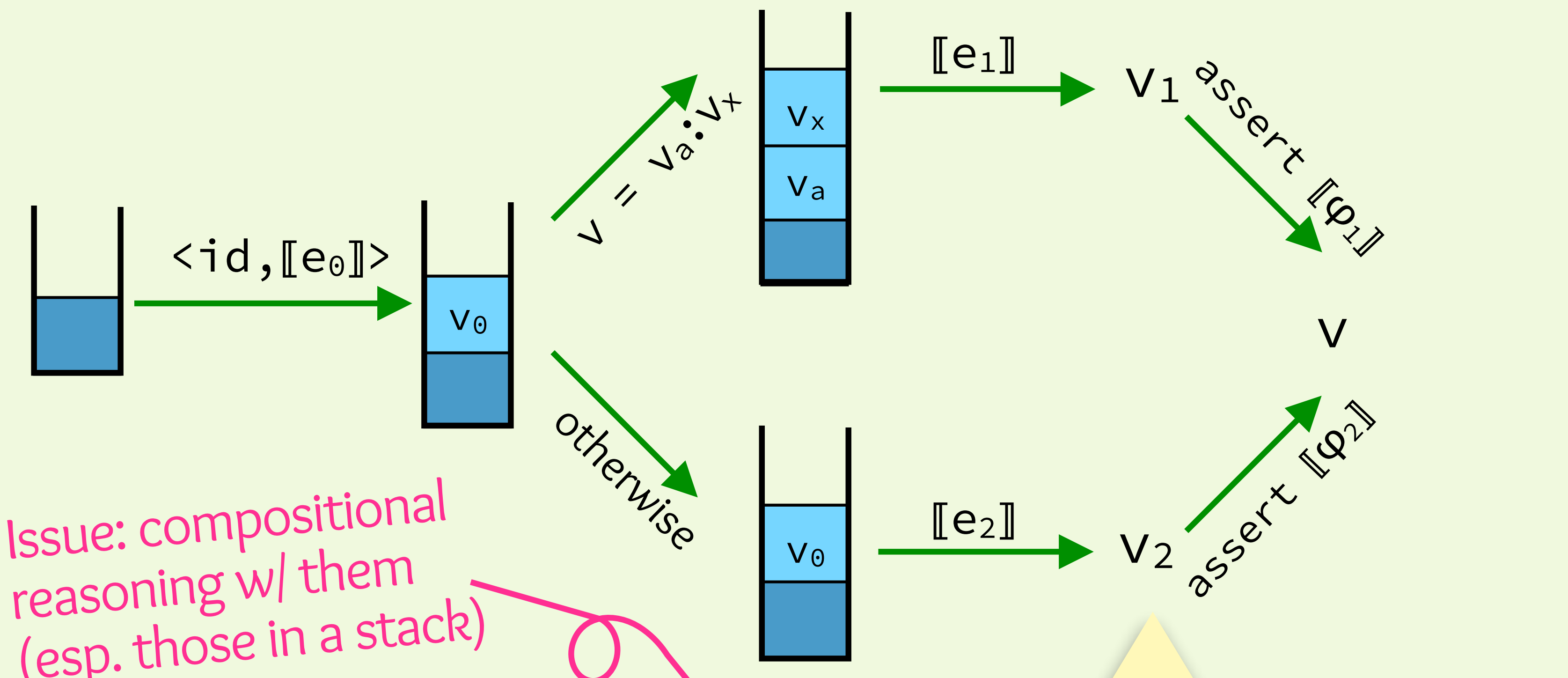
$$[\![\mathbf{B}\tau]\!]_H = \mathsf{BX}\ H\ [\![\tau]\!]$$

$$[\![\sigma \to \tau]\!]_H = \prod_{S\in\mathsf{Tuple}}(\mathsf{BX}\ S\ H \to [\![\sigma]\!]_S \to [\![\tau]\!]_S)$$

Intuitively, S, H, ⟦Δ⟧ are sets of "stacks" to be updated

Idea underlying Denotation of

caseʙ e_0 **of** { a:x -> e_1 **with** φ_1; y -> e_2 **with** φ_2 }



Issue: compositional reasoning w/ them (esp. those in a stack)

Branch switching w/ ***missing values*** during `put` [Foster+07]

Examples (More in Demo)

```
reverse :: B [a] -> B [a]
reverse z = h z []B null
h z r p = caseB z of
  []  -> r with p
  a:x -> h x (a :B r) (p . tail) with not . p
```

```
-- from [M&W 15]
data Val = VFun (Val -> Val) | B Int
data Exp = Var String | App Exp Exp
         | Abs String Exp | Inc Exp

incL :: B Int -> B Int
incL = liftInj (λx.x+1) (λx.x-1)

eval :: Exp -> [(String,Val)] -> Val
eval e env = case e of
  Var x     -> fromJust (lookup x env)
  App e1 e2 -> let VFun f = eval e1 env
               in f (eval e2 env)
  Abs x e1  -> VFun (λv. eval e1 ((x,v):env))
  Inc e     -> VNum (incL n)
```

[M&W15]: Kazutaka Matsuda and Meng Wang: Applicative Bidirectional Programming with Lenses. ICFP 2015. pp. 62–74