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**

```
incTwiceL =
  incL • incL
```

applicative style

```
incTwiceF x =
  incF (incF x)
```

or

with higher-order func.

```
incTwiceF = twice incF
  where
    twice f x = f (f x)
```

Background: Bidir. Trans.

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

Sumii	sumii	205
Oleg	oleg	203
Matsuda	kztk	207

```
get :: Src → View
```

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

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

Background: Bidir. Trans.

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

Sumii	sumii	205
Oleg	oleg	203
Matsuda	kztk	207

`get :: Src → View`

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

↓ update!

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

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

Background: Bidir. Trans.

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

Sumii	sumii	205
Oleg	oleg	203
Matsuda	kztk	207

get :: Src → View

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

↓ update!

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

put :: Src → View → Src

Sumii	sumii	205
Oleg	oleg	203
Matsuda	**kaz**	207

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

Applications

- View update problem
  [Bancilhon&Spyratos81, Hegner90, 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, ...]

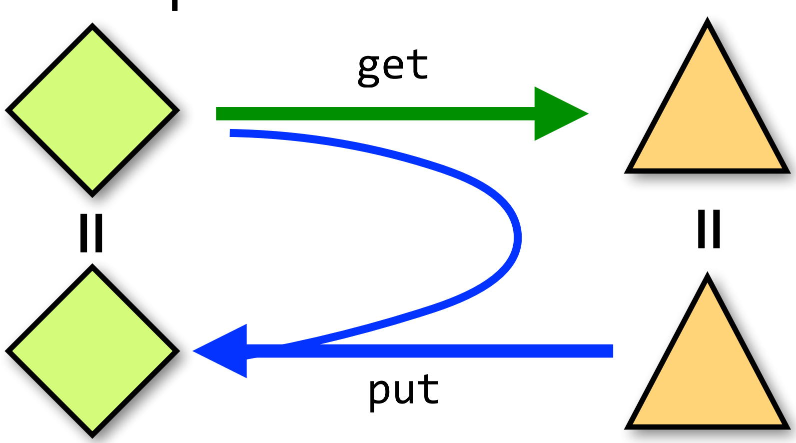
Acceptability (GetPut) **Consistency (PutGet)**

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

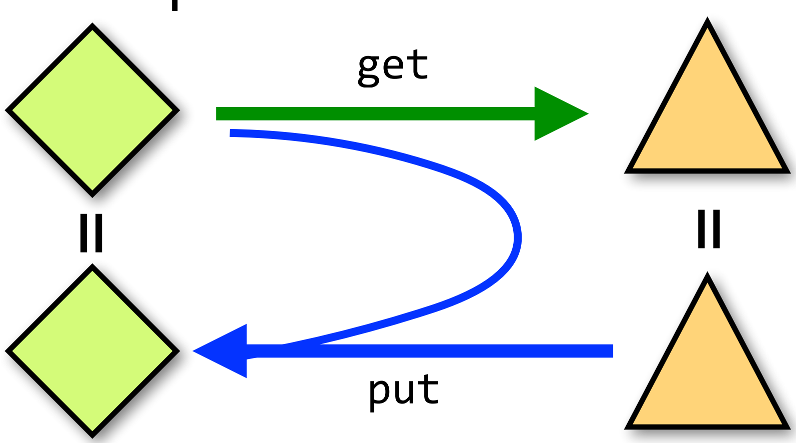


Well-Behavedness

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

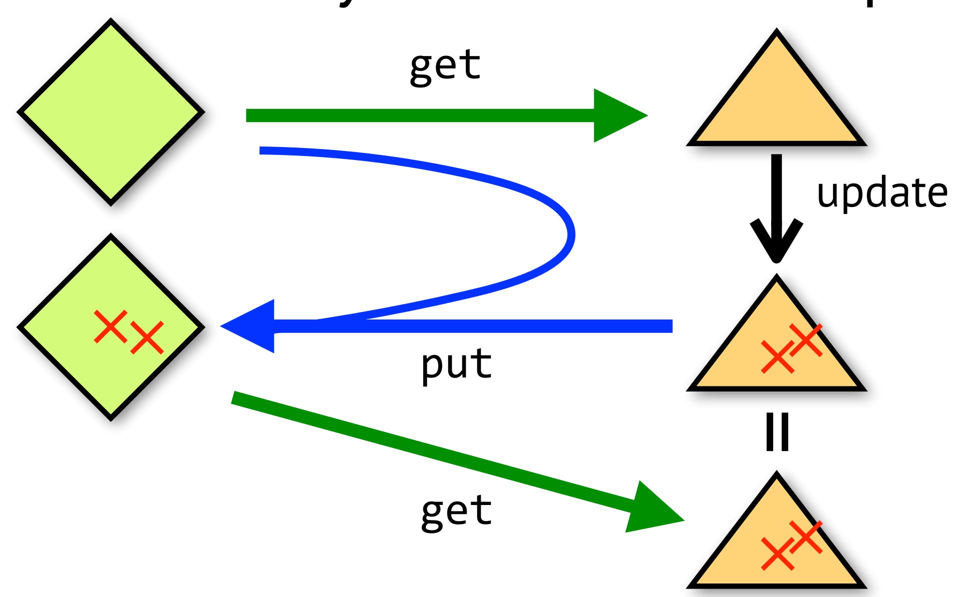
Acceptability (GetPut)

No update on the view,
no update on the source



Consistency (PutGet)

“Put” correctly reflects a view update



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"

```
data Lens a b = L { get :: (a → b),
                    put :: (a → b → a) }
```

- Well-behavedness preserving combinators

```
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"

```
data Lens a b = L { get :: (a → b),
                    put :: (a → b → a) }
```

- Well-behavedness preserving combinators

```
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"

```
data Lens a b = L { get :: (a → b),
                    put :: (a → b → a) }
```

- Well-behavedness preserving combinators

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

well-behaved

```
(•) :: Lens b c → Lens a b → Lens a c
```

well-behavedness preserving

Problem

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

- Programming with ***unfamiliar combinators*** in a ***point-free style***

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

- OK for reasoning
- NG for writing and reading

Our Approach

- ***To lift lenses to functions***

$$\mathtt{lens} :: \mathtt{Lens}\ (A_1,\ldots,A_n)\ B$$

$$\mathtt{lift}_n\ \mathtt{lens} :: (\mathtt{LT}\ s\ A_1,\ldots,\mathtt{LT}\ s\ A_n) \to \mathtt{LT}\ s\ B$$

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

```
lens :: Lens (A₁,…,Aₙ) B
```

```
liftₙ lens :: (LT s A₁,…,LT s Aₙ) → LT s B
```

`liftₙ lens (e₁,...,eₙ)` constructs an ***bidirectional expression*** from ***bidirectional expressions*** `e₁,...,eₙ`.

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

Impossible to violate well-behavedness

```
data LT s a {- Abstract Type -}

liftₙ :: Lens (a₁,…,aₙ) b →
         → ∀s. (LT s a₁,…,LT s aₙ) → LT s b

unliftₙ :: (Eq a₁,…,Eq aₙ) =>
           (∀s. (LT s a₁,…,LT s aₙ) → LT s b)
           → Lens (a₁,…,aₙ) b
```

Theorem (embedding)

$$\mathrm{unlift}_n \circ \mathrm{lift}_n = \mathrm{idL}$$

Example: fst

```
fstF :: (LT s a, LT s b) → LT s a
fstF (a,b) = a
```

transformation on LT values

```
fstL :: (Eq a, Eq b) => Lens (a,b) a
fstL = unlift₂ fstF
```

```
*Main> get fstL (1,2)
1
*Main> put fstL (1,2) 3
(3,2)
```

Example: unlines (1/2)

```
*Main> unlines ["A","B","C"]
"A\nB\nC\n"
```

- Unidirectional unlines

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

Example: unlines (2/2)

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

nullary-lifting

```
catLineF :: LT s String → LT s String → LT s String
catLineF = curry (lift₂ catLineL)
 -- catLineL :: Lens (String,String) String
```

binary-lifting

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

Example: unlines (2/2)

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

nullary-lifting

```
catLineF :: LT s String → LT s String → LT s String
catLineF = curry (lift2 catLineL)
 -- catLineL :: Lens (String,String) String
```

binary-lifting

```
unlinesL :: Lens [String] → String
unlinesL = unliftT unlinesF
```

Example: unlines (2/2)

```
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
```

nullary-lifting

binary-lifting

```
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 = 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"]
```

```
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 []       = 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 (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

Guaranteed Well-Behavedness

Theorem (guaranteed well-behavedness)

For any
“`f :: ∀s. (LT s A₁,…,LT s Aₙ) → LT s B`”
in which `lifting` functions are applied only to well-behaved lenses, “`unliftₙ f`” is well-behaved.

Guaranteed Well-Behavedness

Theorem (guaranteed well-behavedness)

For any
"`f :: ∀s. (LT s A`$_1$`,…,LT s A`$_n$`) → LT s B`"
in which `lifting` functions are applied only to well-behaved lenses, "`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

```
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"

```
bff :: (∀a. [a] → [a])
        → ∀b. Eq b => [b] → [b] → [b]
```

  - Ours subsumes [Voigtländer 09, M&W13]

```
bff f = unliftT (liftT idL ∘ f)
```

Related Work

- van Laarhoven lenses [O'Connor11]

```
Lens A B
```

⬇

```
∀f. Functor f => (B → f B) → (A → f A)
```

  - No guarantee of well-behavedness with λ-abstractions and applications
  - Bidirectional programming is different from programming "get"

Conclusion

http://hackage.haskell.org/package/app-lens

- 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"***

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 (ENum 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 = lkup x env

evalL :: Exp → Lens (Env Int) (Val Int)
evalL e = unliftT (λenv. liftT idL (eval e env))
```

Example: Evaluator (3/3)

```
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 (evalL exp0) env0
VNum 65539
*Main> put (evalL exp0) env0 (VNum 0)
Env [("x", VNum (-65536))]
```

Example: Evaluator (3/3)

```
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)]
```

EInc^{65536} (EVar "x")

```
*Main> get (evalL exp0) env0
VNum 65539
*Main> put (evalL exp0) env0 (VNum 0)
Env [("x", VNum (-65536))]
```

Example: Evaluator (3/3)

```
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)]
```

EInc^{65536} (EVar "x")

The ***first*** bidirectional evaluator for a ***HO***-lang

```
*Main> get (evalL exp0) env0
VNum 65539
*Main> put (evalL exp0) env0 (VNum 0)
Env [("x", VNum (-65536))]
```

Key Difference in HO Eval

Naive Approach

$$[\![\Gamma \vdash e : A]\!] : Lens\ [\![\Gamma]\!]\ [\![A]\!]$$

$$[\![B]\!] = B \text{ for base type } B$$

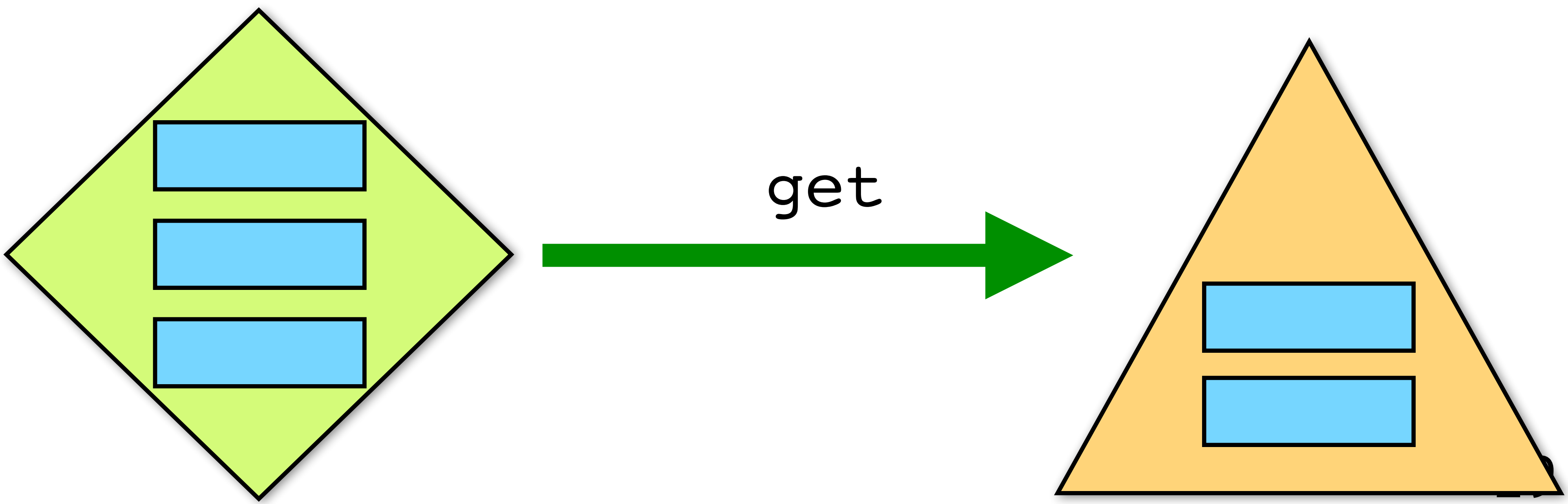
Our Approach

$$[\![\Gamma \vdash e : A]\!]_S : [\![\Gamma]\!]_S \to [\![A]\!]_S$$

$$[\![B]\!]_S = Lens\ S\ B \text{ for base type } B$$

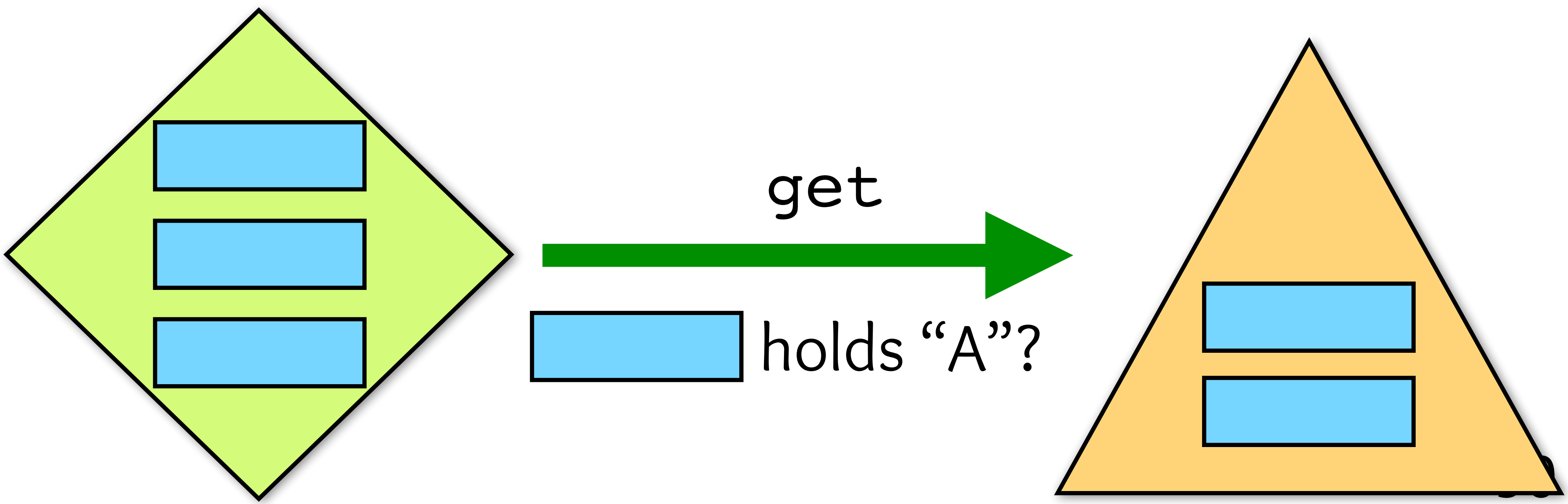
More Detail (1/3)

- [Voigländer09]
  - maps updatable slots to slots
  - slots must be ***untouched*** in “get”



More Detail (2/3)

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



More Detail (3/3)

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