Bidirectional Programming in BiGUL

Syouki Tsuyama

Tokyo Institute of Technology

My interest

- Types and programming languages that satisfies specifications and properties.
- Implementation and Logics of above

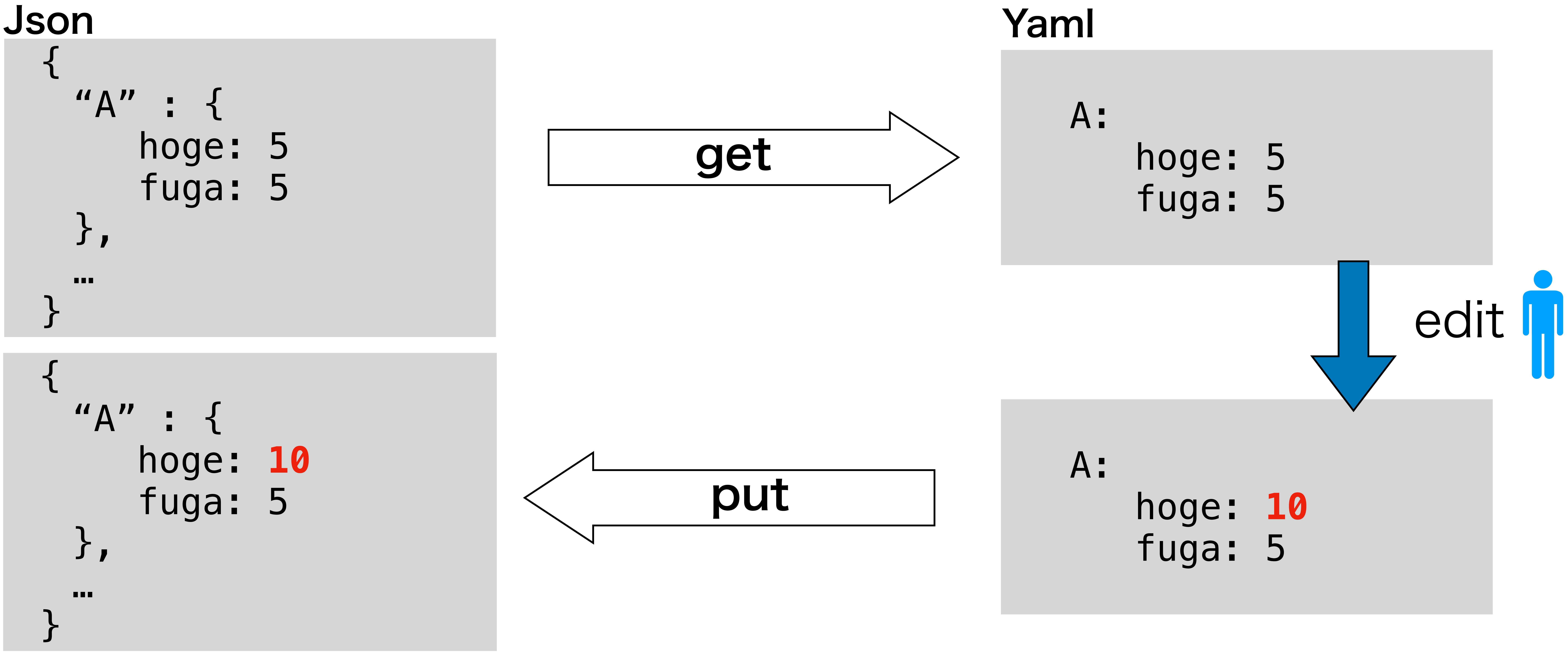
Contents

- Introduction to Bidirectional Programming
- Introduction to BiGUL
- Underlying Logics of BiGUL
- Future Work

Contents

- Introduction to Bidirectional Programming
- Introduction to BiGUL
- Underlying Logics of BiGUL
- Future Works

Synchronization Problem
—Consistency Maintenance—



Json

```
{
  “A” : {
      hoge: 5
      fuga: 5
  },
  …
}
```

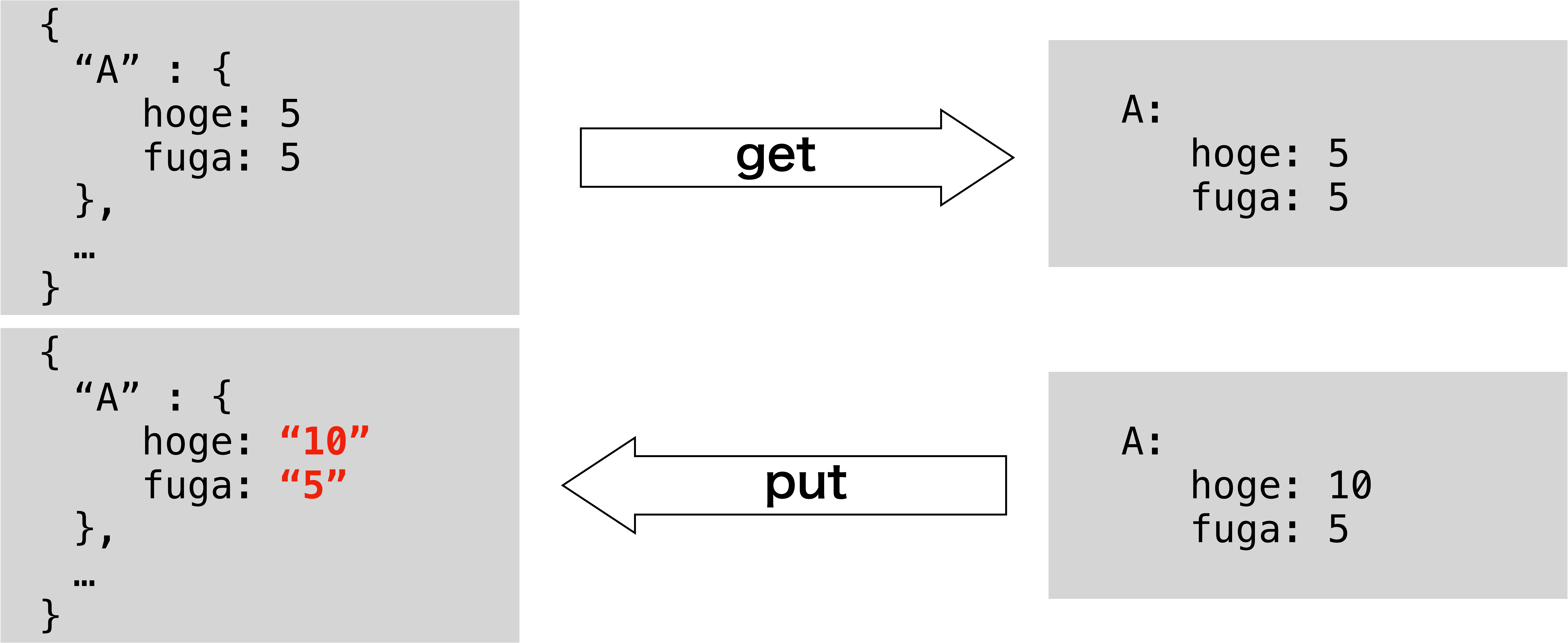
```
{
  “A” : {
      hoge: 10
      fuga: 5
  },
  …
}
```

Yaml

```
A:
    hoge: 5
    fuga: 5
```

```
A:
    hoge: 10
    fuga: 5
```

Synchronization Problem
—Broken Consistency—



```
{
  “A” : {
      hoge: 5
      fuga: 5
  },
  …
}
```

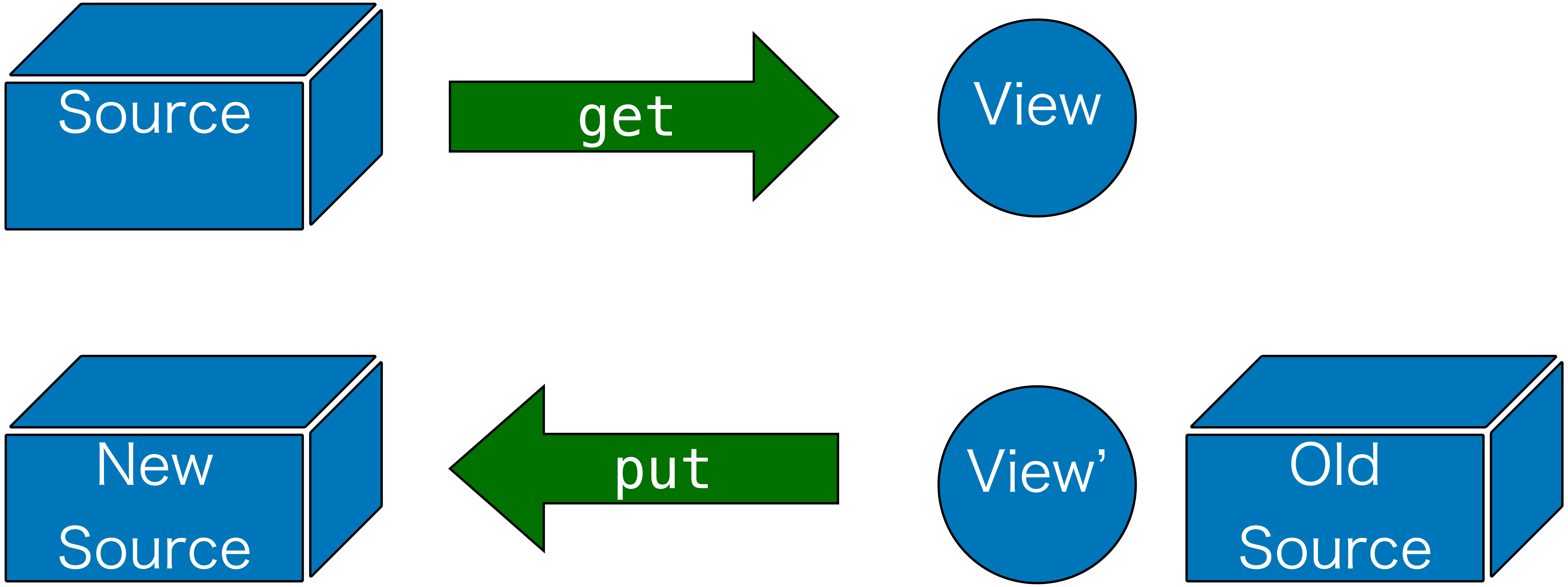
```
A:
    hoge: 5
    fuga: 5
```

```
{
  “A” : {
      hoge: “10”
      fuga: “5”
  },
  …
}
```

```
A:
    hoge: 10
    fuga: 5
```

Bidirectional programming[Foster12]

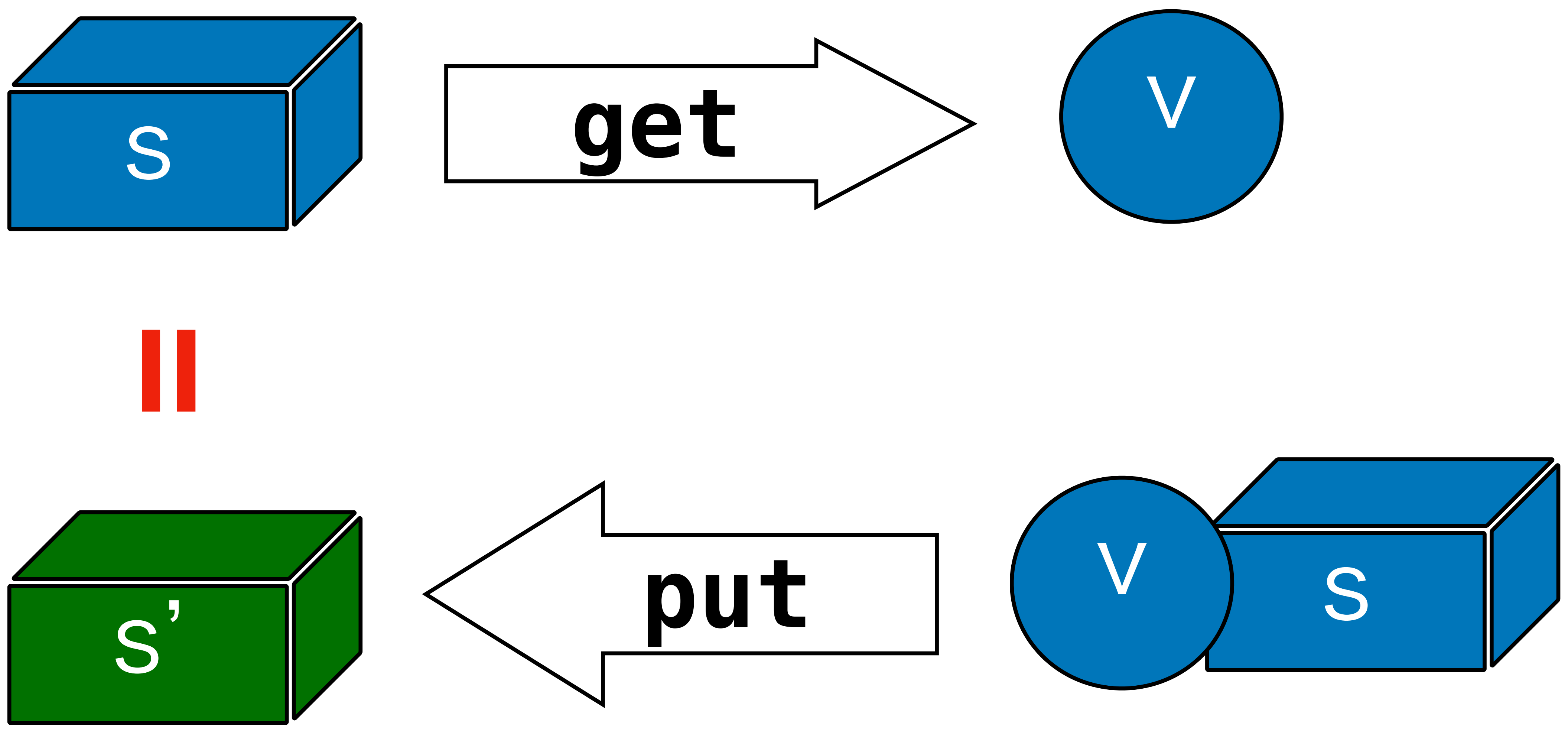
Develop bidirectional transformations: **get** and **put**



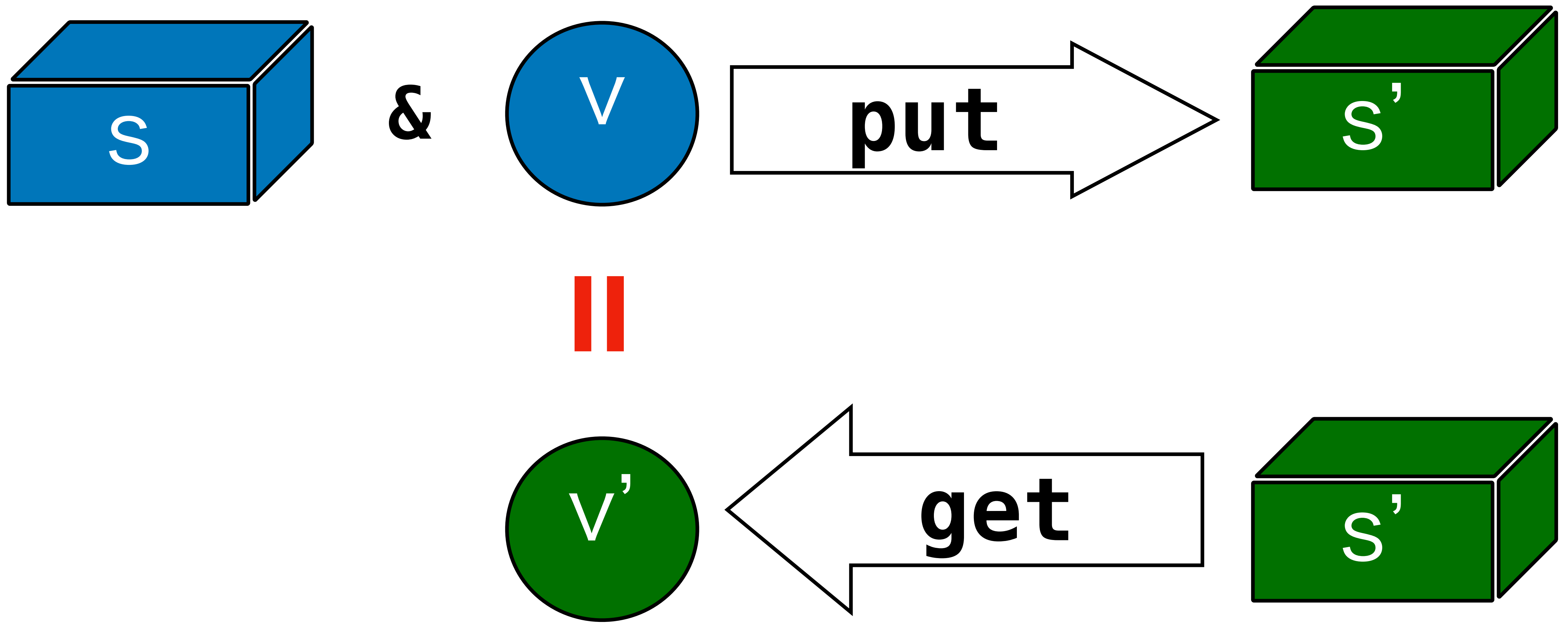
Well-behaved get & put

Well-behavedness: **get** and **put** satisfy two relation

`put s (get s) = s  [GETPUT]`



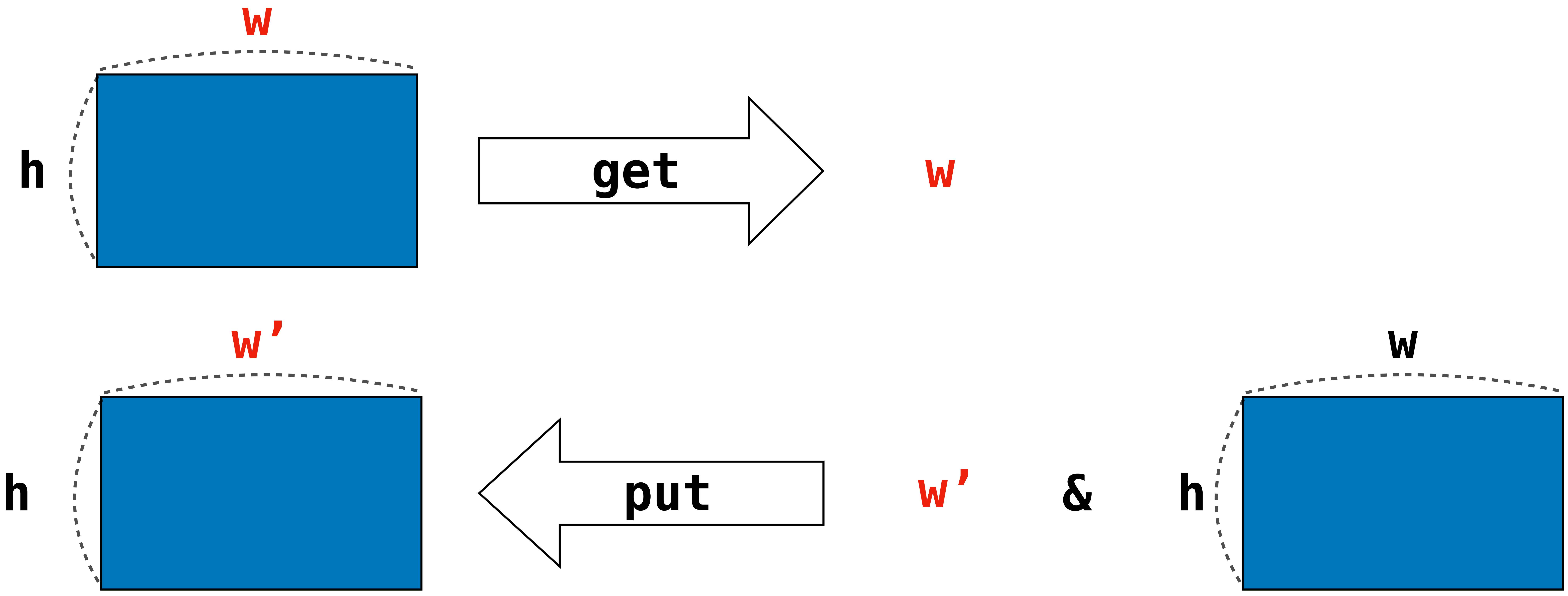
`get (put s v) = v  [PUTGET]`



Ex.) Bidirectional XFMR on Rectangle

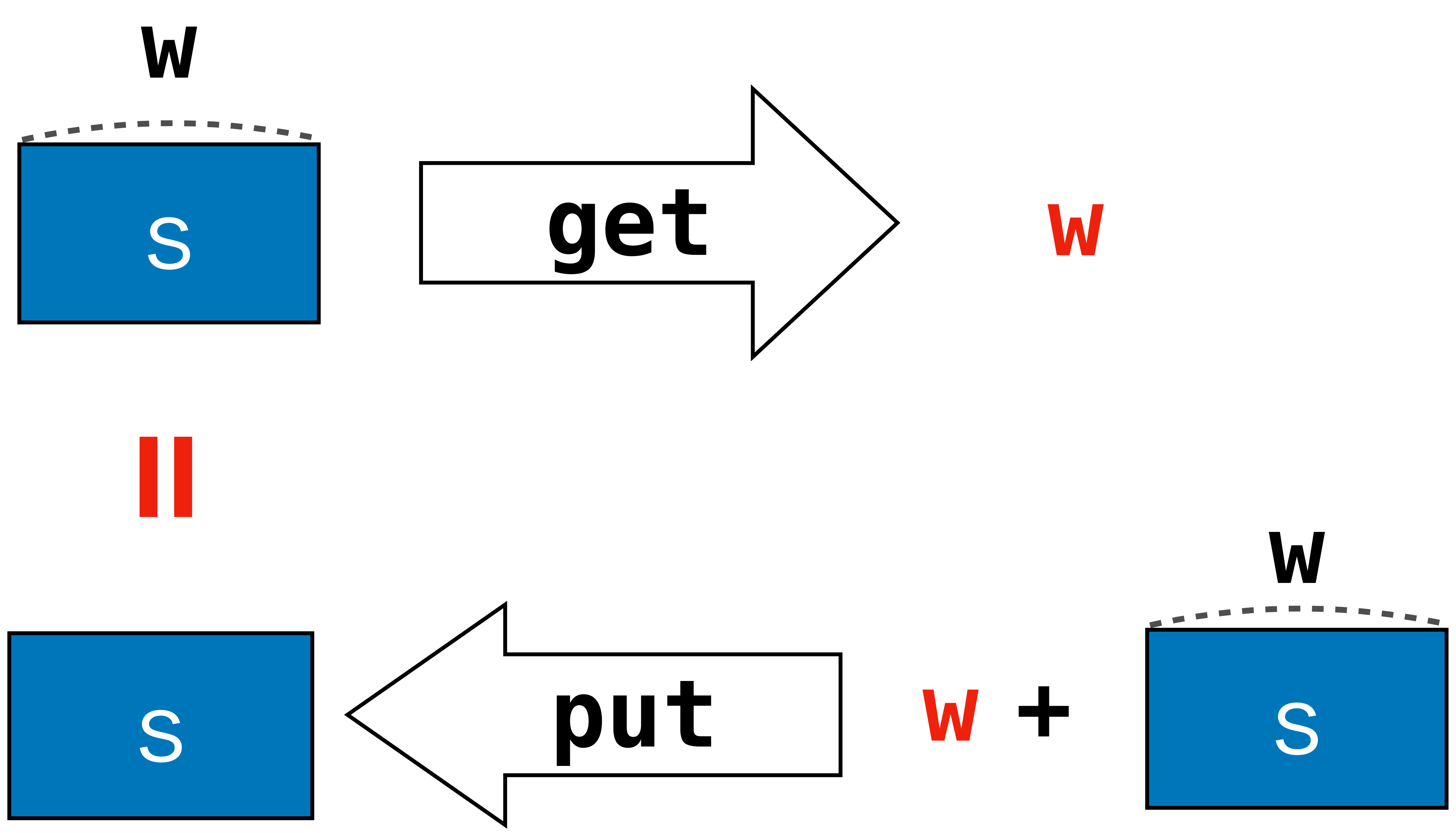
source: rectangle

view: width

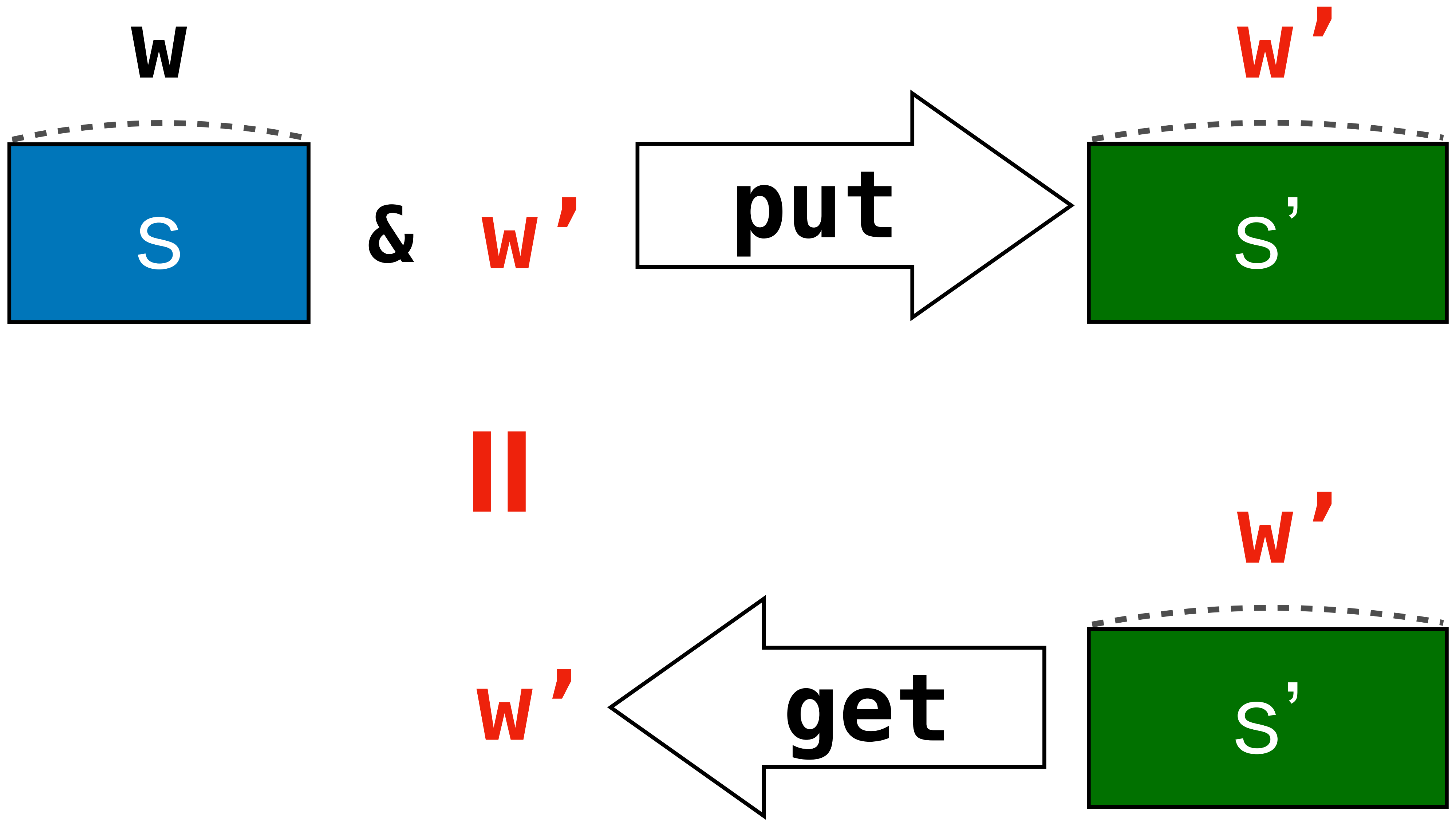


Ex.) Well-behavedness

[GETPUT] `put s (get s) = s`

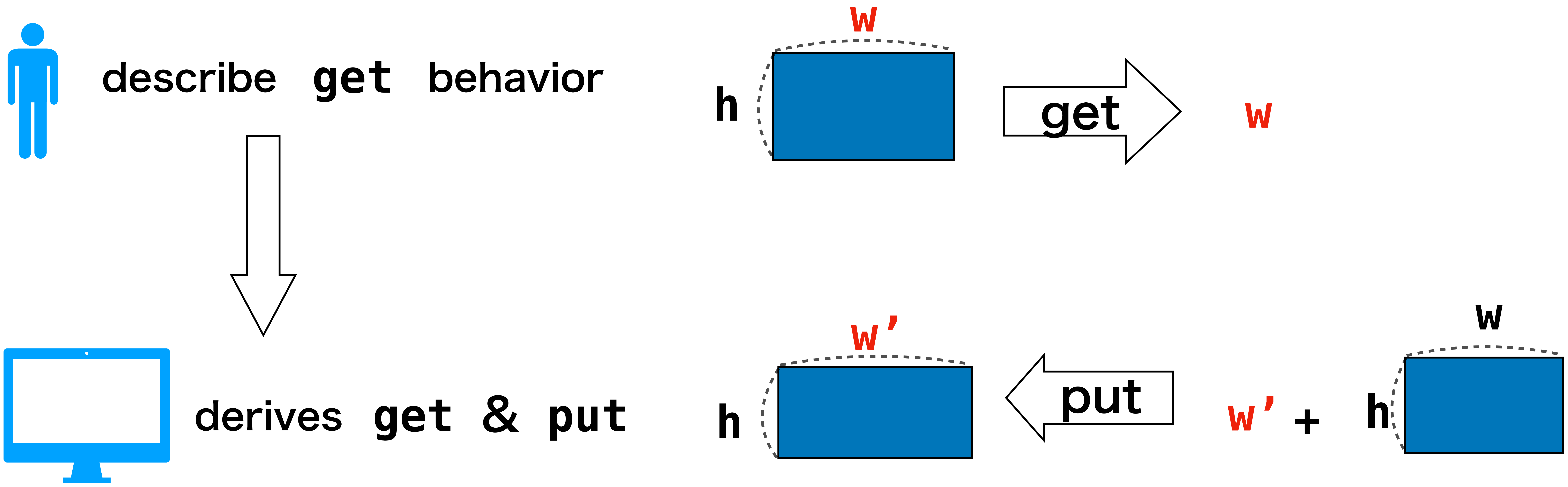


[PUTGET] `get (put s v) = v`



Get-based approach

Existing bidirectionl frameworks can derive put based on get-based approach [Bohannon08][Foster07]

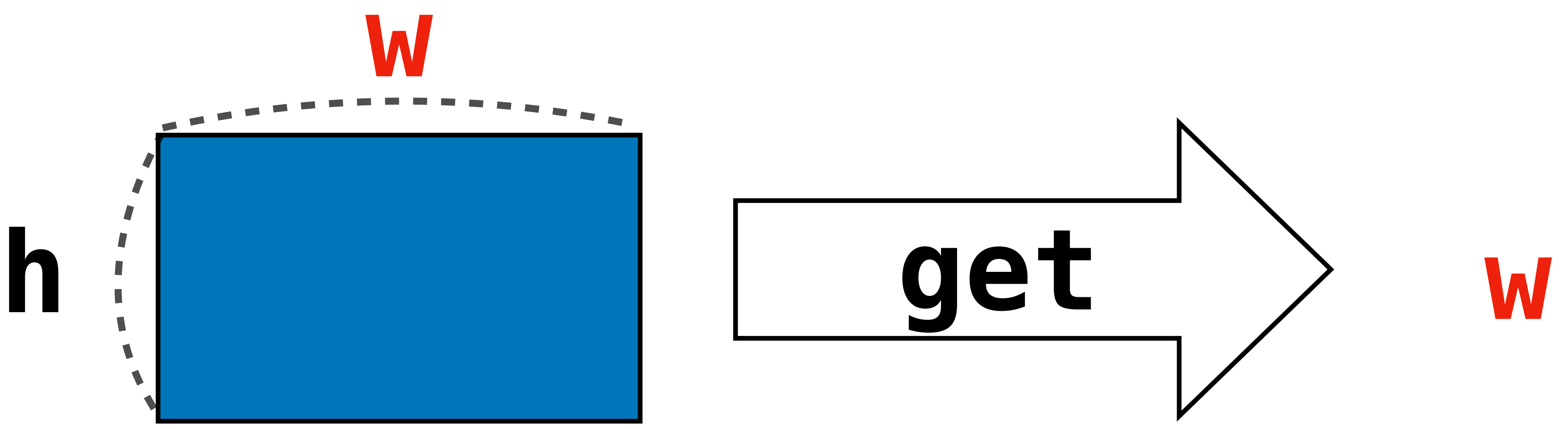


describe **get** behavior

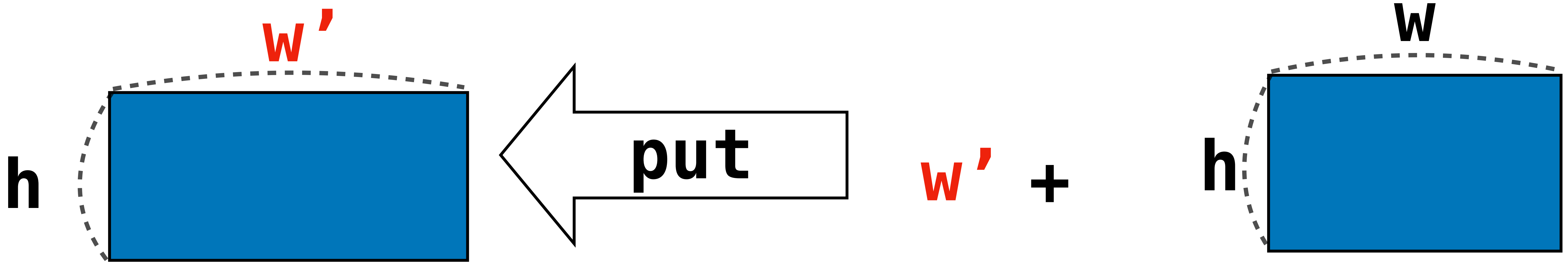
derives **get & put**

A problem of get-based approach

E.g) `get` has multiple strategies for well-behaved `put`

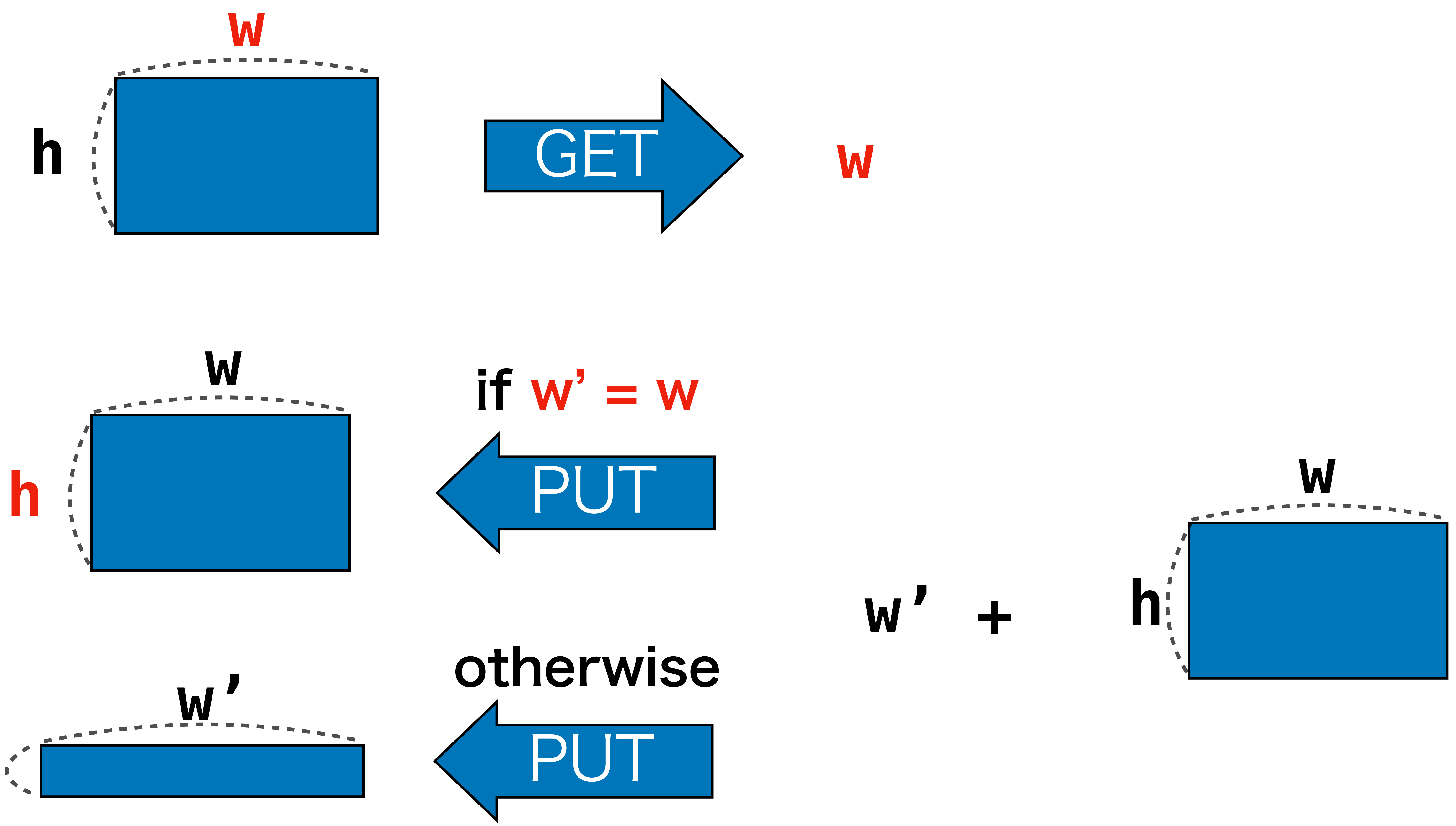


① keepHeight



A problem of get-based approach

E.g) `get` has multiple strategies for well-behaved `put`



Issues from multiple `put`

Programmers need to verify that

the derived **`put`** meets the application specifications

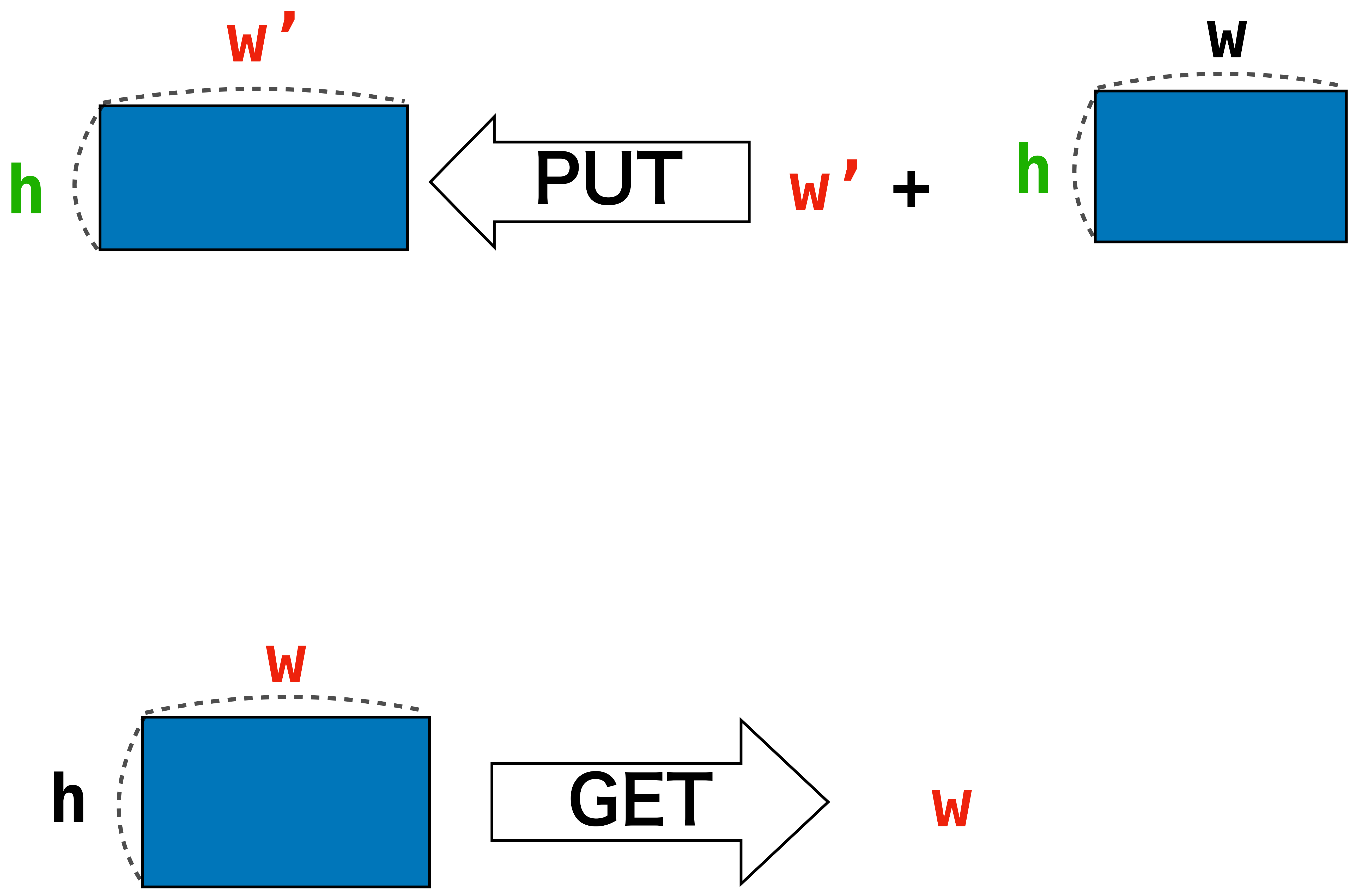
and need a mechanism to control **`put`**

Putback-based approach[Pacheco14][Ko16]

describe **put** behavior

(instead of **get**)

derives **get** & **put**



Pros & Cons

	Pros	Cons
get-based	No need to prove well-behavedness	Need a control of **put**
putback-based	• No need to prove well-behavedness • **Full-control** consistency retention behavior	

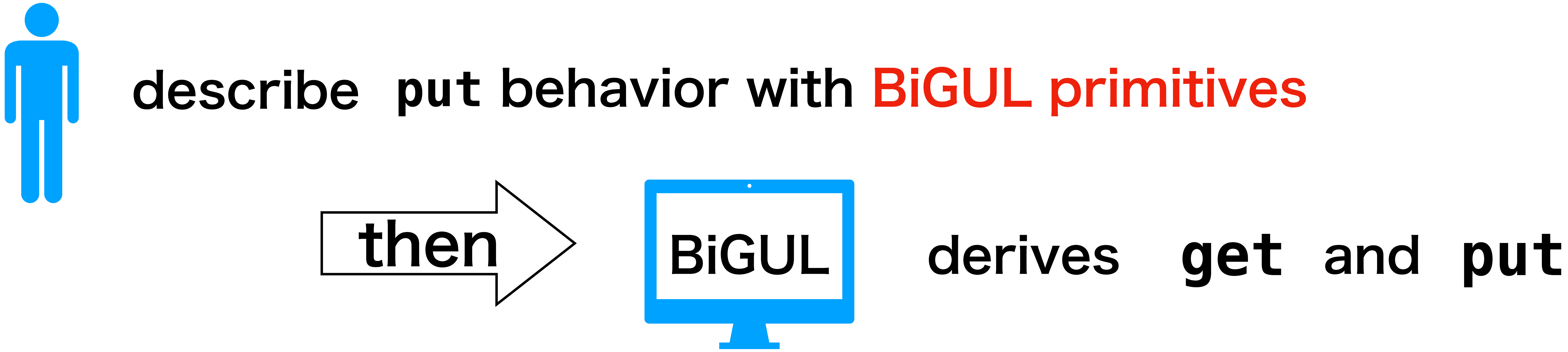
today’s topic

Contents

- Introduction to Bidirectional Programming
- Introduction to BiGUL
- Underlying Logics of BiGUL
- Future Work

BiGUL[Ko16]

- **Putback-based** bidirectional programming language
- Implemented as embedded DSL in Haskell



describe **`put`** behavior with **BiGUL primitives**

derives **`get`** and **`put`**

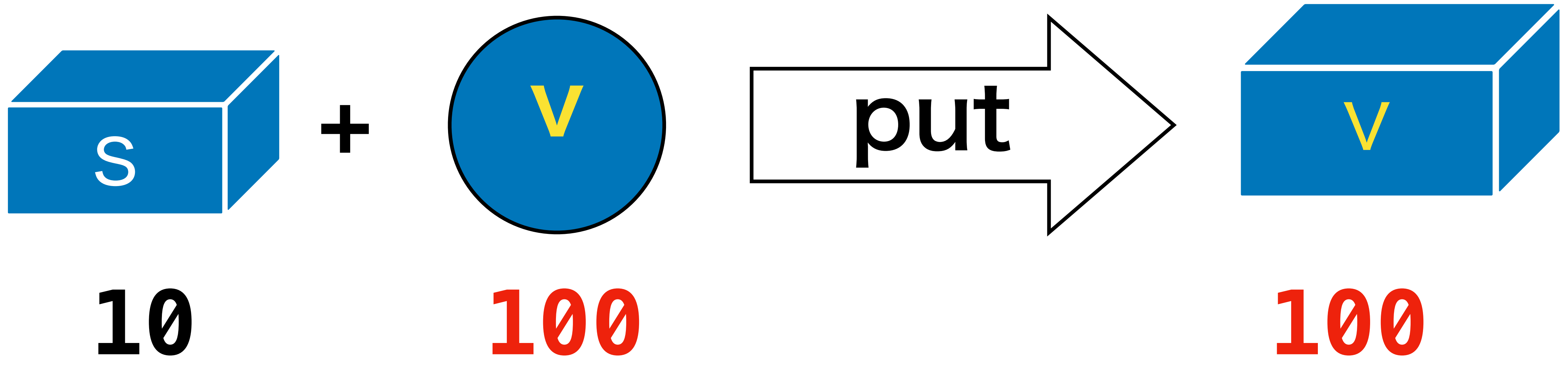
```
get :: BiGUL s v -> s -> Maybe v
put :: BiGUL s v -> s -> v -> Maybe s
```

BiGUL Primitives : Replace

```
Replace :: BiGUL s s
```

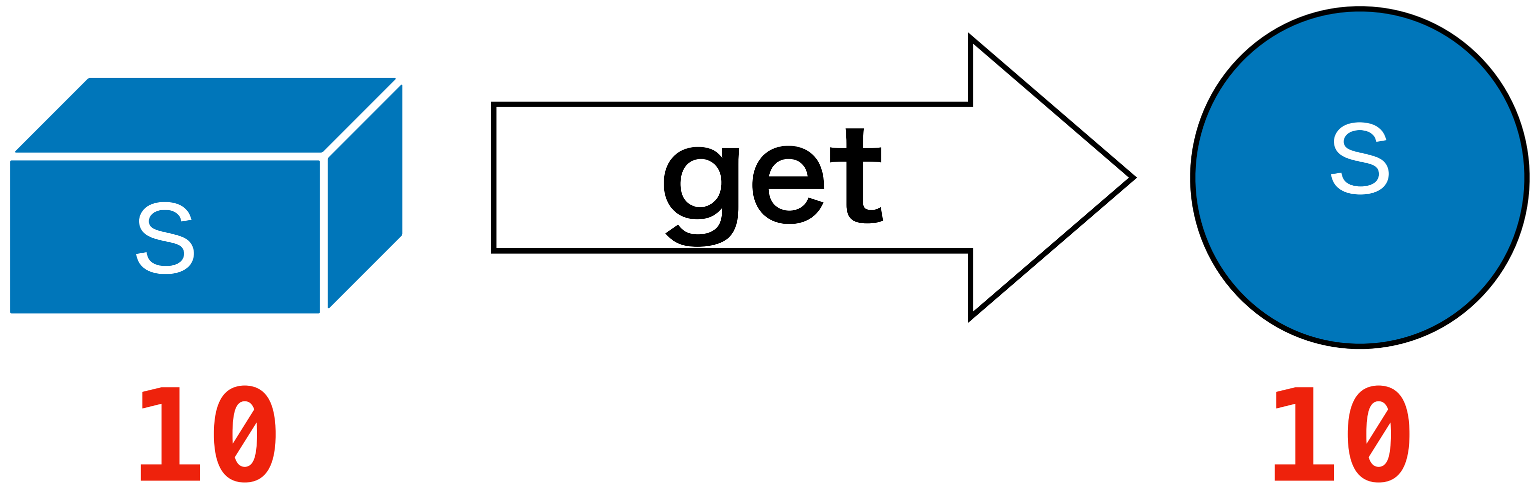
Completely replace the source with the view

```
put  Replace
```



```
> put Replace 10 100
Just 100
```

```
get  Replace
```



```
> get Replace 10
Just 10
```

BiGUL Primitives : Skip

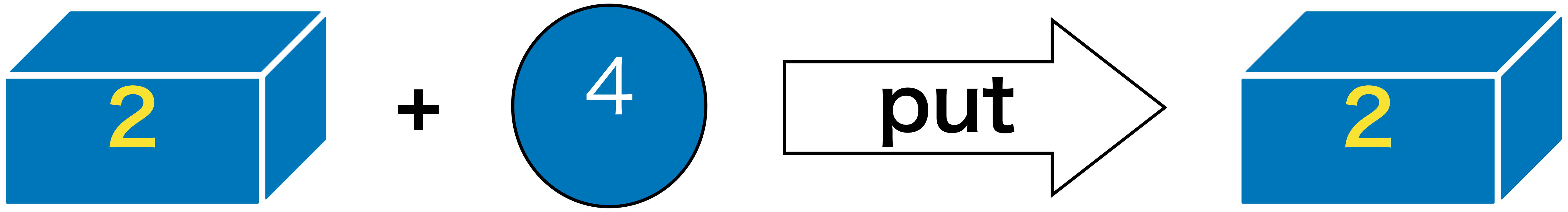
```
Skip :: (s → v) → BiGUL s v
```

Does nothing to the source.

```
square x = x * x
b = Skip square
```

`put (Skip square)`

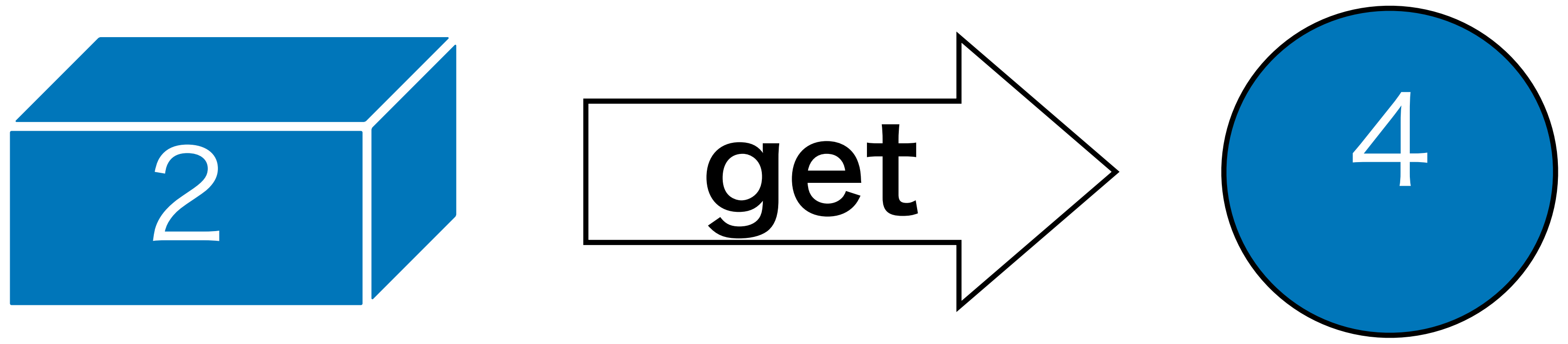
succeed with no source update
only if (square s == v)



square 2 == 4

```
> put b 2 4
Just 2
```

`get (Skip square)`

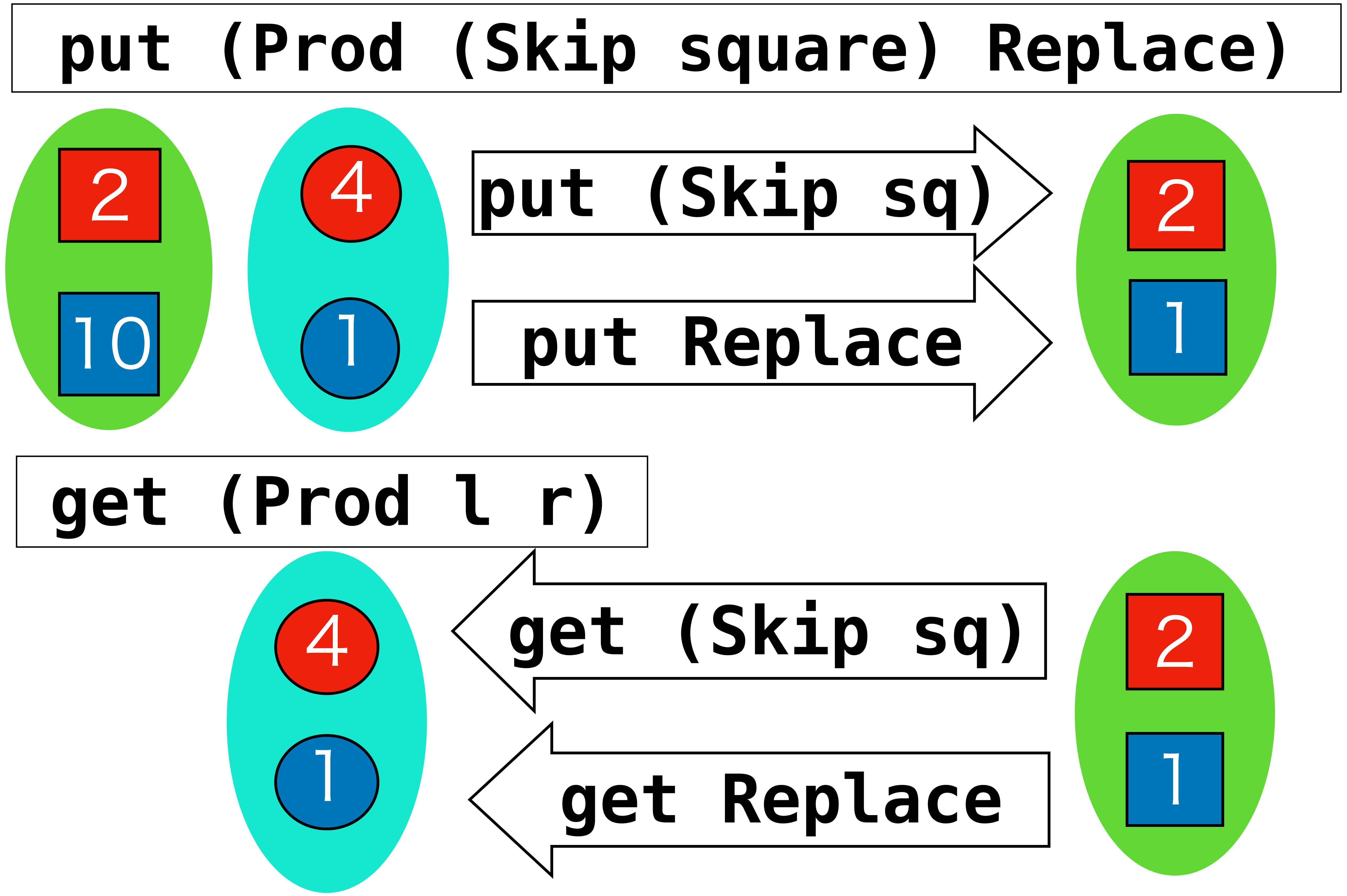


```
> get b 2
Just 4
```

BiGUL Primitives: Production

```
Prod :: BiGUL s1 v1 → BiGUL s2 v2 → BiGUL (s1,s2) (v1,v2)
```

Product two transformations to deal with source pair and view pair



put (Prod (Skip square) Replace)

get (Prod l r)

```
b = Prod (Skip square) Replace
```

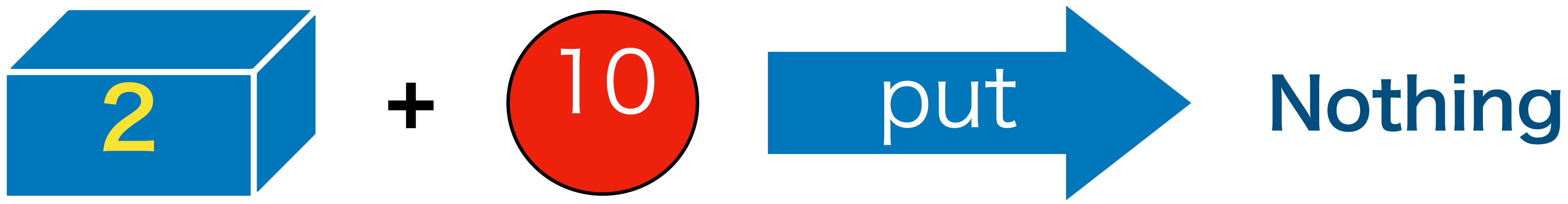
```
> put b (2 , 10) (4 , 1)
Just (2 , 1)
```

```
> get b (2 , 1)
Just (4 , 1)
```

BiGUL’s runtime check

```
data Maybe a =
   Nothing |
   Just a
```

`get` & `put` may fail
when they violate well-behavedness



Check fails	square 2 != 10

```
 > put (Skip square) 2 10
 Nothing
```

`10 != square 2`

==> The consistency which the **`(Skip square)`** guarantees is broken

Contents

- Introduction to Bidirectional Programming
- Introduction to BiGUL
- Underlying Logics of BiGUL
- Future Work

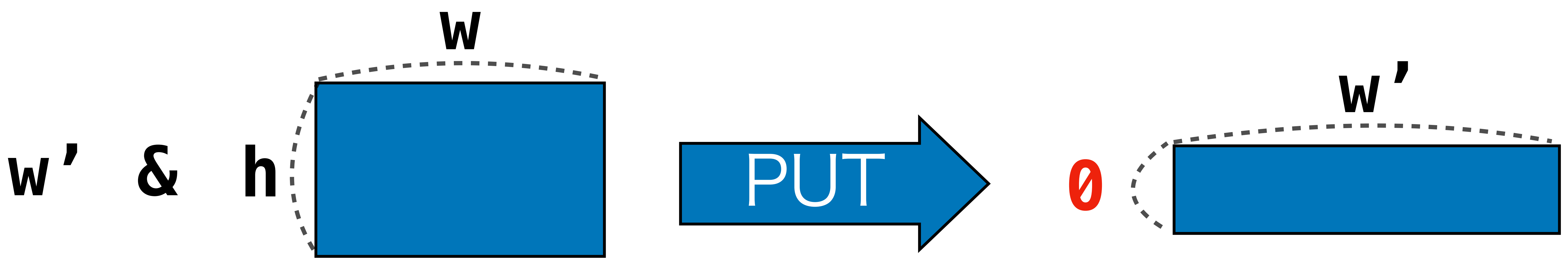
Infer the behavior of `put` & derived `get`[Ko18]

- What kind of input does `put` succeed for ?
- How does the updated source retain the original information?

- What kind of input does `get` succeed for ?
- How does `get` produce a view from source information?

Ex.) ill-behaved get to be detected

alwaysResetHeight



```
> get alwaysResetHeight (1 , 2)
Nothing
```

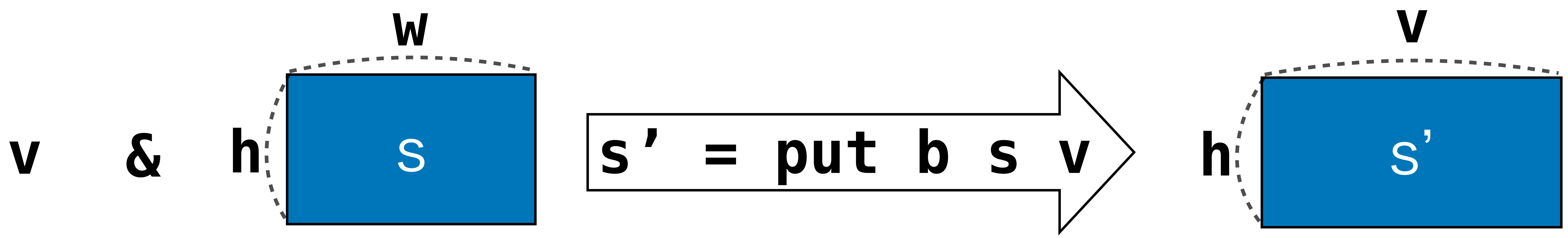
ill-behaved

if get **(1 , 2)** = Just 1
then put (1 , 2) 1 = **(1 , 0)**

We would like to find **successful input range** of **get** without writing tests.

Inference of behavior **put**

Putback-triple: $\{R\}\ \ b\ \ \{R'\}$



$\{\_\ \_ \mid \mathrm{True}\}$ $\qquad b \qquad$ $\{(w', h')\ (w, h)\ v \mid w' = v \wedge h' = h\}$

Conditions to be satisfied in
old source and view

Conditions to be satisfied in
on new source and old ones

Deriving example

Inference Rules

$$\frac{}{\{s\ v \mid f\ s = v\}\quad \text{Skip}\ f\quad \{s'\ s\ \_ \mid s' = s\}}$$

$$\frac{}{\{\_\ \_ \mid \text{True}\}\quad \text{Replace}\quad \{s'\ \_\ v \mid s' = v\}}$$

$$\frac{\{L\}\ \ l\ \ \{L'\} \qquad \{R\}\ \ r\ \ \{R'\}}{\{L * R\}\quad \text{Prod}\ l\ r\quad \{L' * R'\}}$$

Deriving tree of (`Prod (Skip square) Replace)`

$$\dfrac{\dfrac{}{\begin{array}{c}\{s\ v \mid \text{square}\ s = v\}\\ \text{Skip square}\\ \{s'\ s\ \_ \mid s' = s\}\end{array}} \qquad \dfrac{}{\begin{array}{c}\{\_\ \_ \mid \text{True}\}\\ \text{Replace}\\ \{s'\ \_\ v \mid s' = v\}\end{array}}}{\begin{array}{c}\{\mathbf{(s1\ ,\ \_)\ (v1\ ,\ \_) \mid square\ s1 = v1}\}\\ \mathbf{Prod\ (Skip\ square)\ Replace}\\ \{\ \mathbf{(s'1\ ,\ s'2)\ (s1\ ,\ \_)\ (\_\ ,\ v2) \mid s'1 = s1 \wedge s'2 = v2}\ \}\end{array}}$$

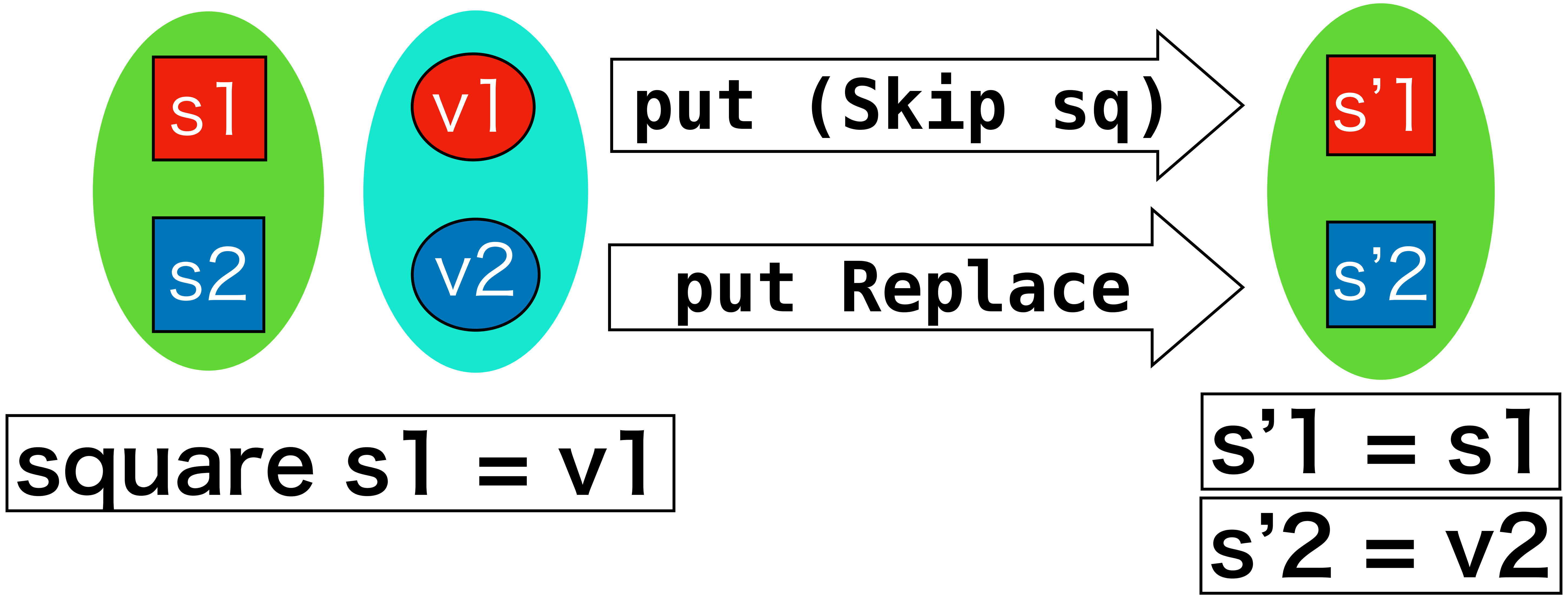
Deriving result

we derived:

{(s1 , _) (v1 , _) | square s1 = v1}

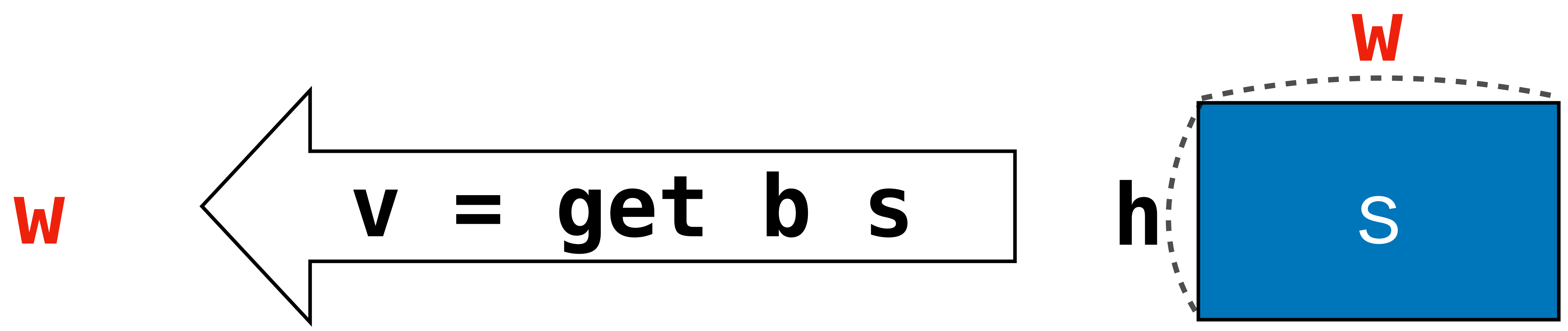
Prod (Skip square) Replace

{(s'1 , s'2) (s1 , _) (_ , v2) | s'1 = s1 ∧ s'2 = v2 }



Inference of behavior **get**

Range-triple: $\{\{R\}\}$ **b** $\{\{R'\}\}$



$\{\{\ (w\ ,h)\ v \mid w = v\ \}\}$ **b** $\{\{\_ \mid \mathrm{True}\}\}$

Conditions to be satisfied in input source and produced view

Conditions to be satisfied in on input source

Range-triple: Inference of **get** behavior

Inference Rules

$$\frac{}{\{\{s\ v \mid f\ s = v\}\}\quad \text{Skip } f \quad \{\{\_\ \_ \mid \text{True}\}\}}$$

$$\frac{}{\{\{s\ v \mid s = v\}\}\quad \text{Replace} \quad \{\{\_\ \_ \mid \text{True}\}\}}$$

$$\frac{\{\{L\}\}\quad l \quad \{\{L'\}\} \qquad \{\{R\}\}\quad r \quad \{\{R'\}\}}{\{\{L * R\}\}\quad \text{Prod } l\ r \quad \{\{L' * R'\}\}}$$

Deriving tree of `(Prod (Skip square) Replace)`

$$\frac{\dfrac{}{\begin{array}{l}\{\{s\ v \mid \text{square } s = v\}\}\\ \text{Skip square}\\ \{\{\_\ \_ \mid \text{True}\}\}\end{array}} \qquad \dfrac{}{\begin{array}{l}\{\{s\ v \mid s = v\}\}\\ \text{Replace}\\ \{\{\_\ \_ \mid \text{True}\}\}\end{array}}}{\begin{array}{l}\{\{(s1\ ,\ s2)\ (v1\ ,\ v2) \mid \text{square } s1 = v1 \wedge s2 = v2\ \}\}\\ \text{Prod (Skip square) Replace}\\ \{\{\_\ \_ \mid \text{True}\}\}\end{array}}$$

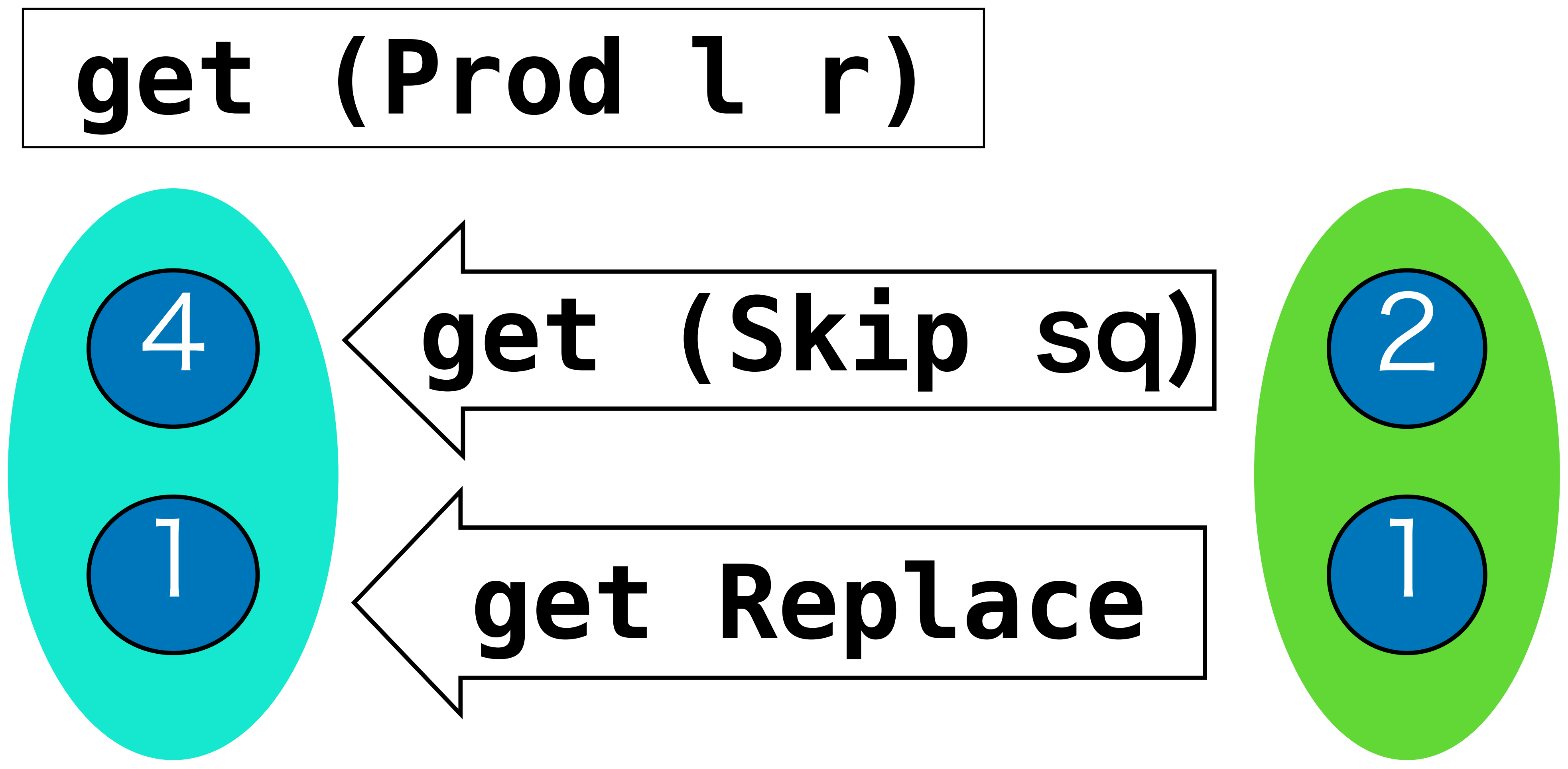
Deriving result

we derived:

{{(s1 , s2) (v1 , v2) | square s1 = v1 ∧ s2 = v2 }}

Prod

{{_ _ | True}}



Therefore,

```
get (Prod (Skip square) Replace)
```

succeed for all input source

and the produced view (v1 , v2) is equal

to (square s1 , s2)

Summary

- The motivation: Synchronization Problem
- Bidirectional Programming achieves safe consistency maintenance
  - source and view
  - **Well-behaved** get & put
- BiGUL adopts **Putback-based** approach
  - Derive **unique** `get` that satisfies well-behavedness from `put`
  - Infer the behavior of **get** and **put** by **putback-triple**, **range-triple**

Contents

- Introduction to Bidirectional Programming
- Introduction to BiGUL
- Underlying Logics of BiGUL
- Future Work

Future Work

- Runtime check of well-behavedness can incur serious performance overheads.
- BiGUL programs can quickly become awkward to write and hard to read.
- It is not easy to develop reusable libraries.

  —> Design new language constructs ?

- Graph data structure is hard for functional languages, but many applications require it.
- asymmetric lenses are less expressive.

  —> New programmable formalisms? or Implementing existing formalisms?

I would like to

- Explore other formalization of Bidirectional Programming
- Think about the way of writing put behavior imperatively.
  - BiGUL into Monad, do block?
- Can it be static checked?
  - refinement types or dependant types
- Lightweight dynamic checks

References

- Hsiang-Shang Ko and Zhenjiang Hu. 2018. An Axiomatic Basis for Bidirectional Programming. https://doi.org/10.1145/3158129

- Zhenjiang Hu and Hsiang-Shang Ko. Principles and Practice of Bidirectional Programming in BiGUL https://josh-hs-ko.github.io/manuscripts/SSBX16.pdf