Principle and Practice of Putback-based Bidirectional Programming in BiGUL

Zhenjiang Hu, Hsiang-Shang Ko
National Institute of Informatics, Japan

BX Summer School, Oxford
July 25-29, 2016

Overview

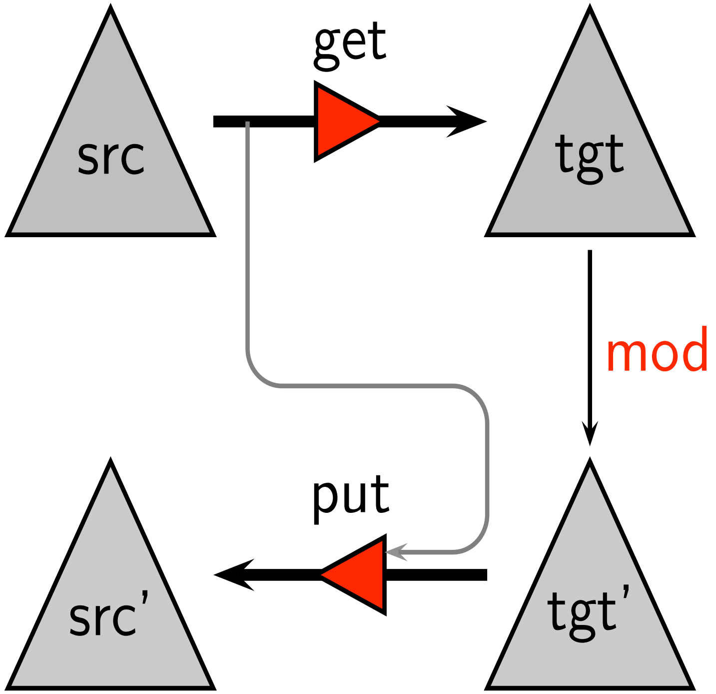
- Lecture 1: Introduction to BiGUL
  - Why is putback-based BX?
  - What is its foundation?
  - How to program in BiGUL?
- Lecture 2: Into BiGUL's Bidirectionality
  - How is BiGUL implemented?
- Lecture 3: Three Applications using BiGUL
  - Matching/delta lenses in BiGUL
  - Bidirectionalize relational queries with BiGUL
  - Parsing and reflective printing (BiYacc)

https://goo.gl/MdJeyk: lecture notes and codes

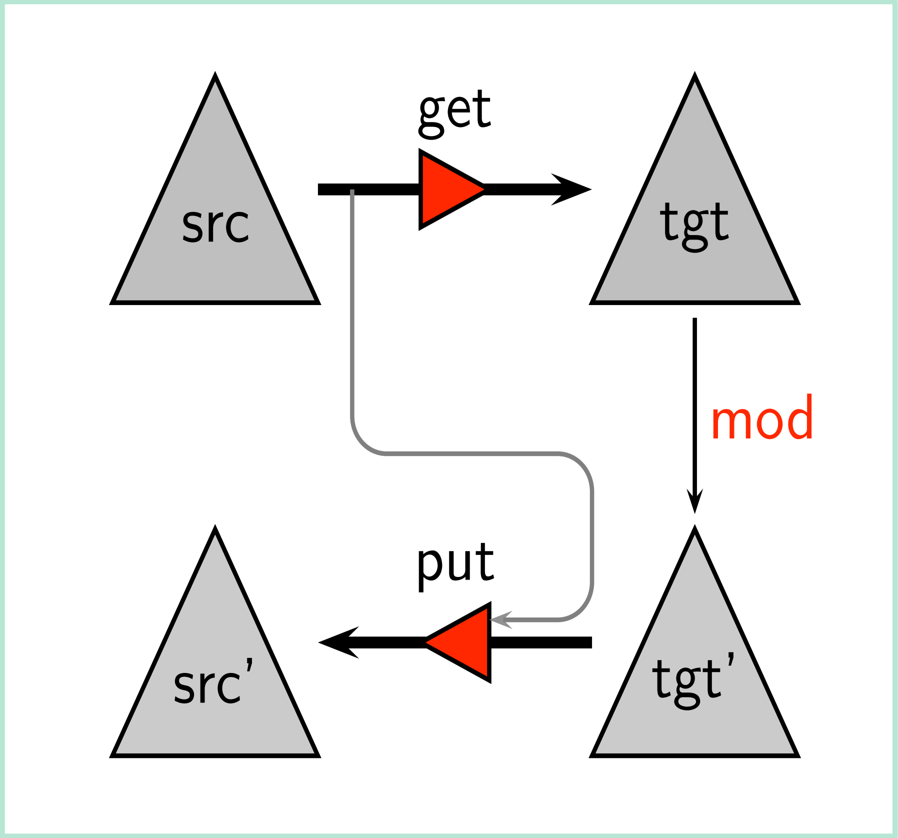
Bidirectional Programming (using Functions)

Bidirectional Transformation (BX)

[Nate Foster, et al: POPL 2005]



Roundtrip Properties



Get-Put:
put s (get s) = s

Put-Get:
get (put s t) = t

Challenges

- One may “solve” the problem just by sticking together two arbitrary functions in any programming language you like.

- Tricky to get right... and even trickier to maintain

- Need to find a way of deriving both functions from a single description.

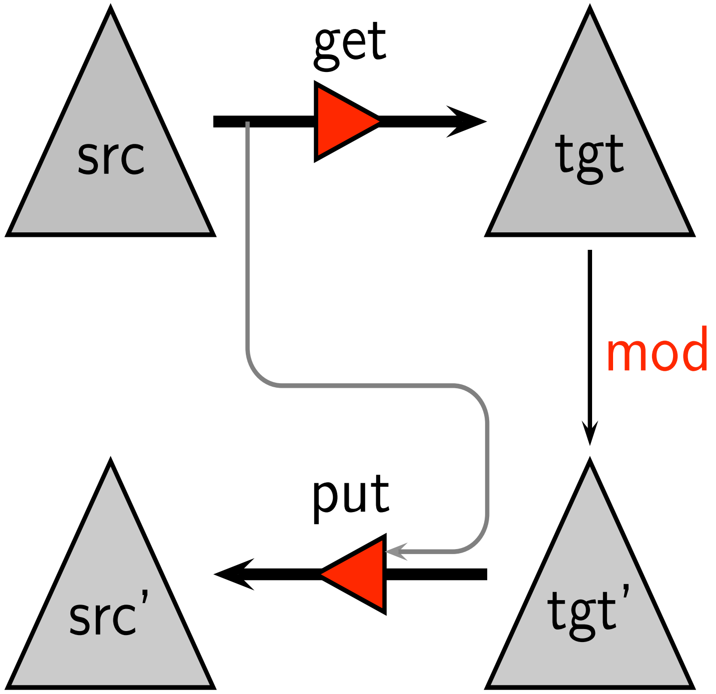
Get-based Approach

"get" (forward transformation)

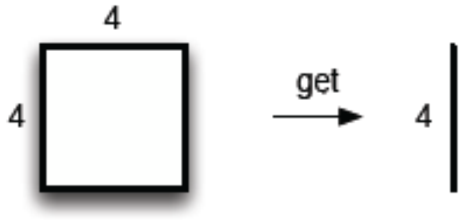
"put" (backward transformation)

- Domain Specific Bidirectional Languages (lens1, lens2, ...)
- Automatic Bidirectionization of ATL, XQuery, UnQL

Ambiguity of “Put”

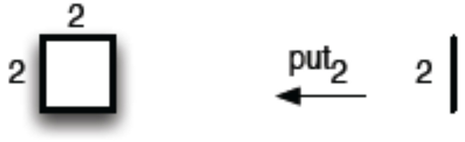


Since get is generally non-injective, many suitable puts correspond to one get, each being useful in different context.

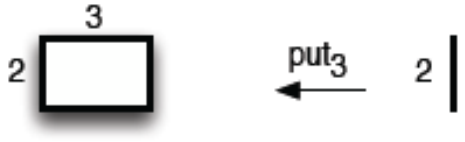


```
getHeight (w,h) = h
```

```
putHeight1 (w,h) h' = (w,h')
```



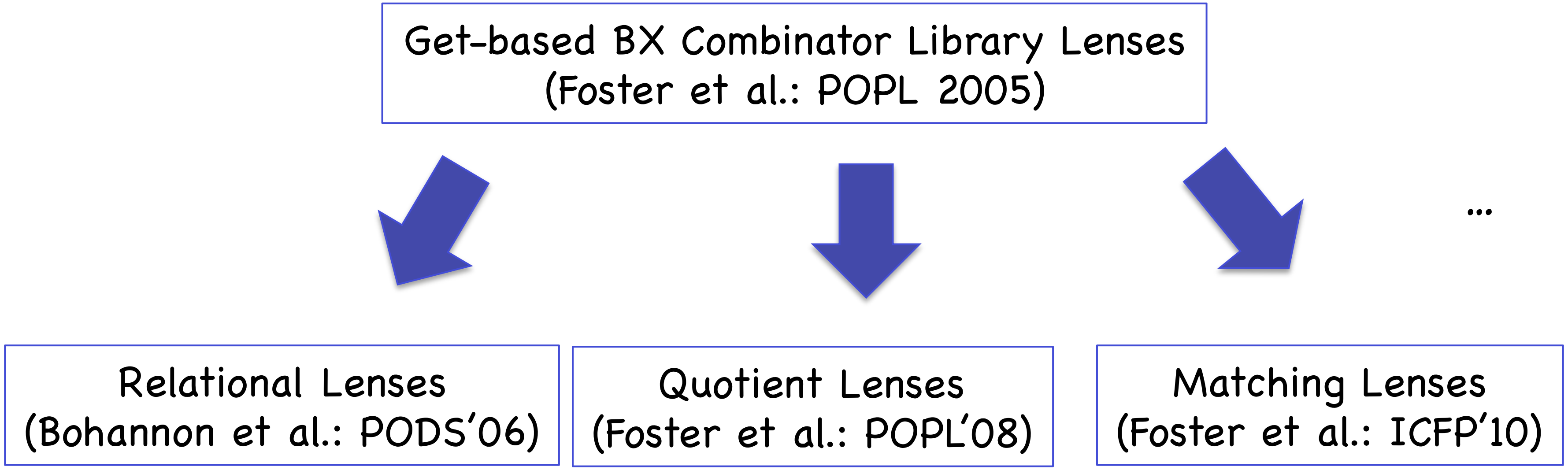
```
PutHeight2 (w,h) h' = (w*h'/h, h')
```



```
putHeight3 (w,h) h' | h==h' = (w,h)
                    | otherwise = (3,h')
```

One Solution: Enriching "get"

Enrich "get" with more and more control over "put"



Get-based BX Combinator Library Lenses
(Foster et al.: POPL 2005)

...

Relational Lenses
(Bohannon et al.: PODS'06)

Quotient Lenses
(Foster et al.: POPL'08)

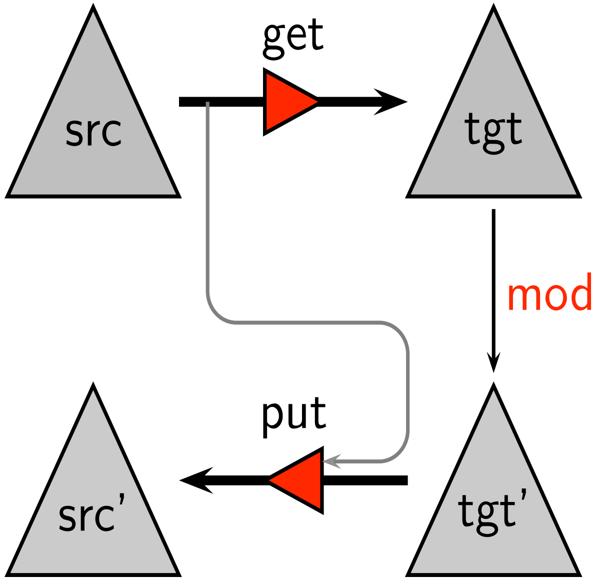
Matching Lenses
(Foster et al.: ICFP'10)

➔ We will have too many versions of "get" ...

Foundation of Putback-based Bidirectional Programming

Putback is the essence of BX!

- An important but little-known fact:



put uniquely determines get

Derived “get”: Uniqueness

Lemma: Given a put function, there exists at most one get function such that GetPut and PutGet hold.

Proof:

Suppose we have two get functions, say get and get′.

get s

get′ s

Derived "get": Uniqueness

Lemma: Given a put function, there exists at most one get function such that GetPut and PutGet hold.

Proof:

Suppose we have two get functions, say get and get′.

get s

=> { GetPut }

get (put s (get′ s))

== { PutGet }

get′ s

Derived “get”: Existence

Lemma: Given a surjective put function (for any s, there exist s′, v, such that s = put s′ v), the get function defined by

get s = v such that put s v = s

satisfies GetPut and PutGet.

Proof:

get (put s v)

v

put s (get s)

s

Derived “get”: Existence

Lemma: Given a surjective put function (for any s, there exist s′, v, such that s = put s′ v), the get function defined by

get s = v **such that** put s v = s

satisfies GetPut and PutGet.

Proof:

get (put s v)
= { condition for put }
get s
= { definition of get}
v

put s (get s)
= { definition of get }
put s v
= { condition for put }
s

Remark

Lemma: if there exists a v satisfying put s’ v = s, then so does put s v = s.

Proof:

```
    put s’ v = s
=>   { put }
     put (put s’ v) v = put s v
==   { PutTwice: forall s v. put (put s v) v = put s v }
     put s’ v = put s v
==   { Assumtion: put s’ v = s }
     s = put s v
```

Well-behaved “put”

Definition: A “put” function is said to be well-behaved, if there exists a (unique) “get” function such that GetPut and PutGet hold.

Exercise

Which of the following puts are well-behaved?

① put1 s v = s
② put2 s v = v
③ put2 s v = v+1

Well-behaved “put”

Lemma:

put is well-behaved, **iff**

1. View-deterministic

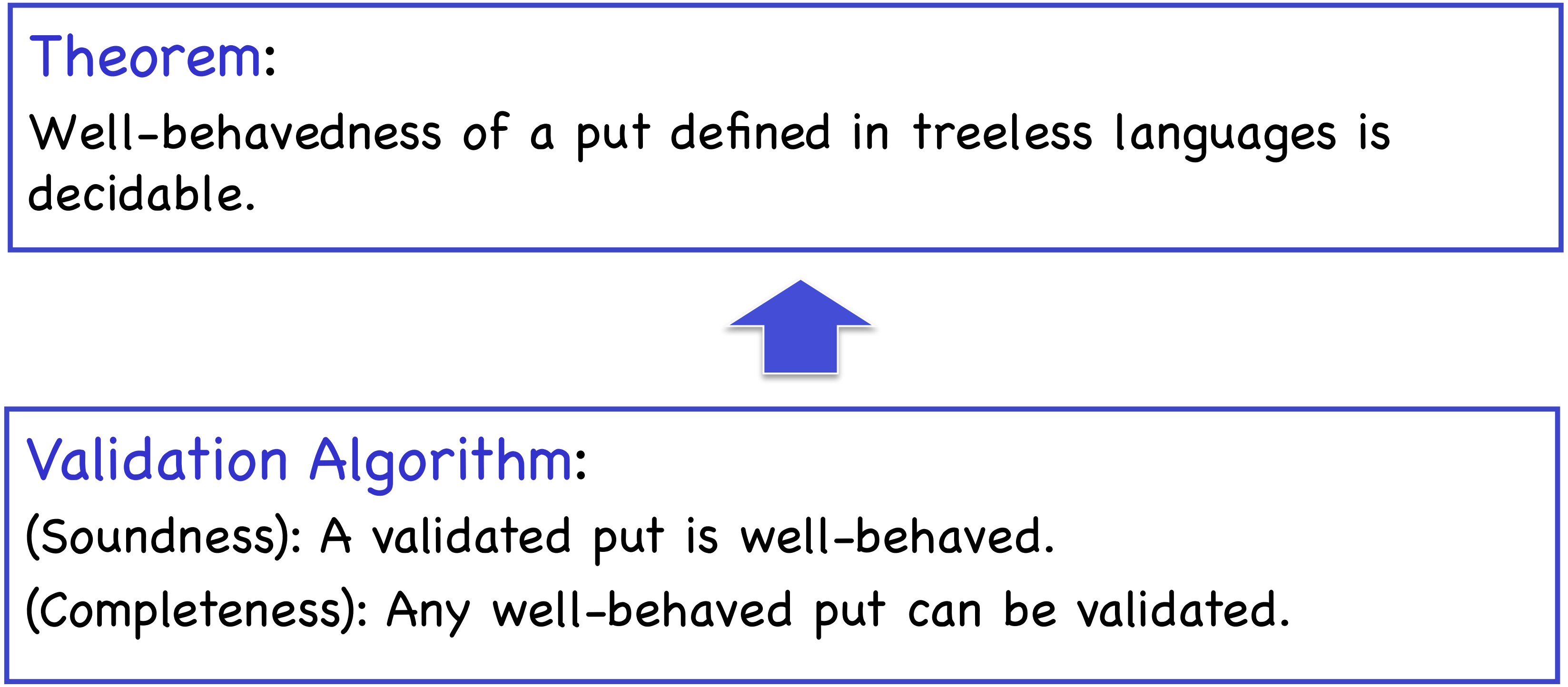
   put s1 v1 = put s2 v2 ➔ v1 = v2

2. View-stable

   for any s, there exists a v, such that put s v = s

Sebastian Fischer, Zhenjiang Hu, Hugo Pacheco, **Pearl: A Clear Picture of Lenses**, MPC 2015.

Validity Check of "Put"



Theorem:

Well-behavedness of a put defined in treeless languages is decidable.

Validation Algorithm:

(Soundness): A validated put is well-behaved.

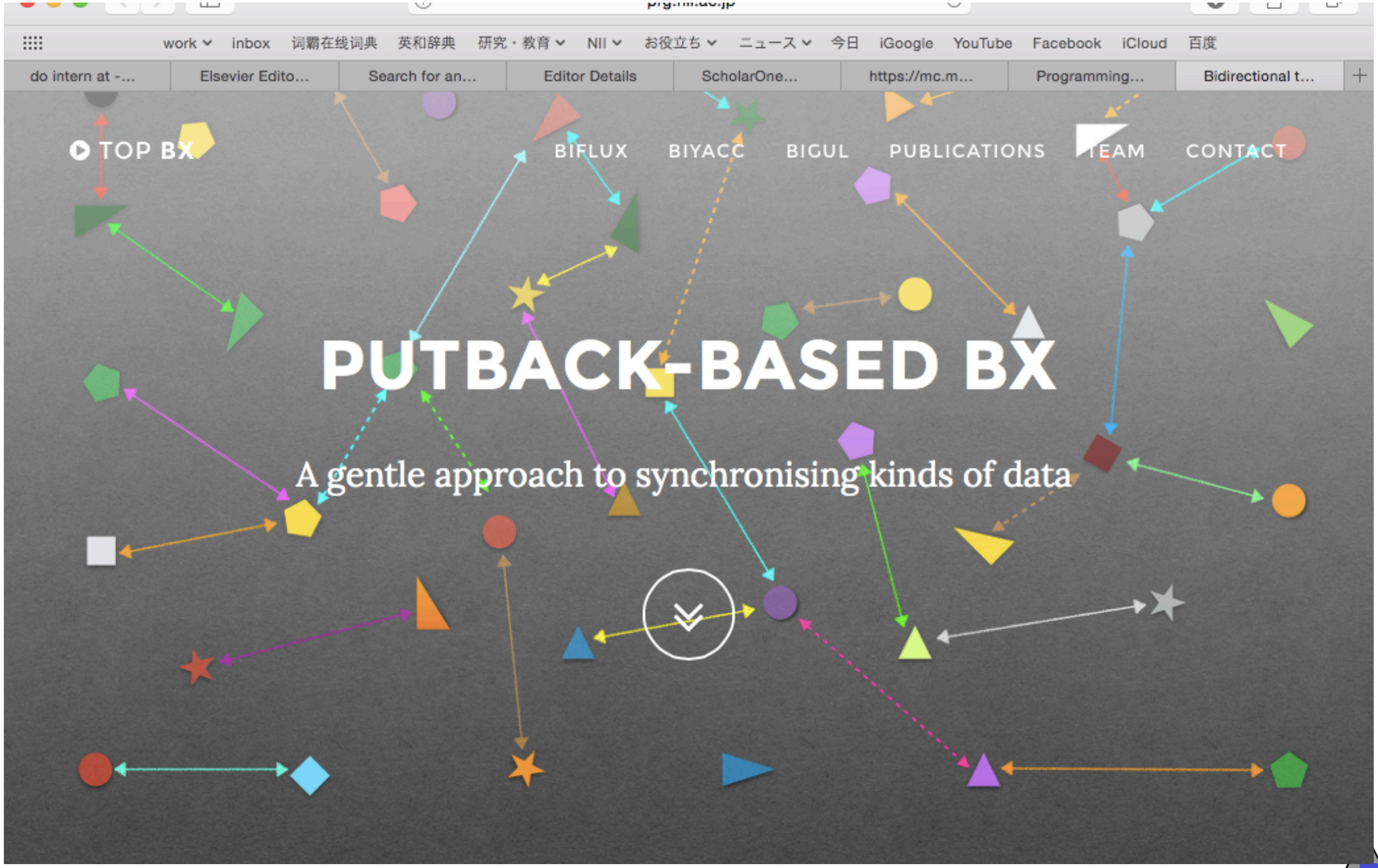
(Completeness): Any well-behaved put can be validated.

Zhenjiang Hu, Hugo Pacheco, Sebastian Fischer, **Validity Verification of Putback Transformations in Treeless Languages in Bidirectional Programming**, FM 2014.

Putback-based Bidirectional Programming in BiGUL

- Full control of bidirectional behavior
- Put is not that difficult to write

http://www.prg.nii.ac.jp/bx/



PUTBACK-BASED BX

A gentle approach to synchronising kinds of data

Installing BiGUL

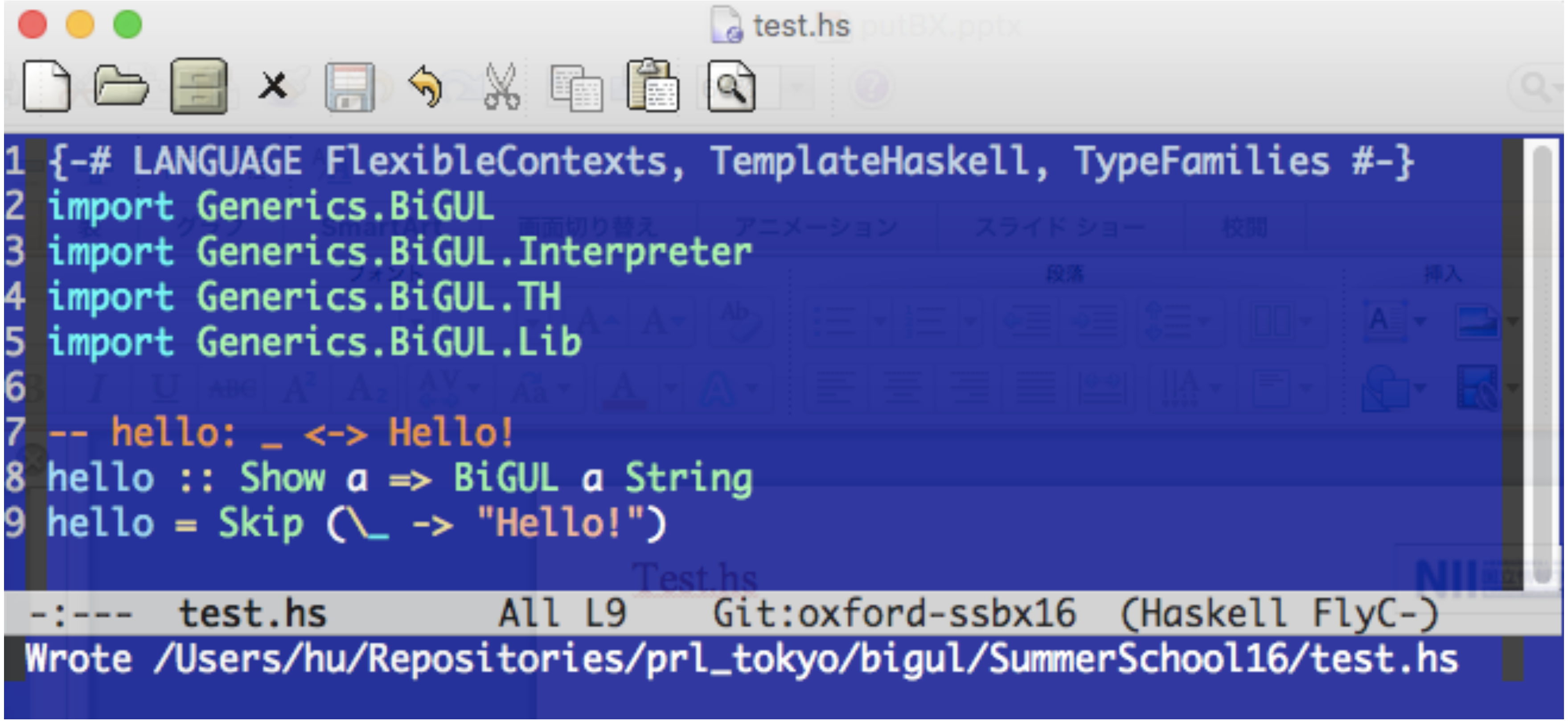
- 1. Get the Glasgow Haskell Compiler (GHC) version 7.10.3. The easiest way is to install the Haskell Platform:

  https://www.haskell.org/platform/

- Use Haskell's default build system "cabal" to install BiGUL 1.0 (as a library). Start your terminal and run

```
>  cabal update
>  cabal install BiGUL
```

Test.hs



```
{-# LANGUAGE FlexibleContexts, TemplateHaskell, TypeFamilies #-}
import Generics.BiGUL
import Generics.BiGUL.Interpreter
import Generics.BiGUL.TH
import Generics.BiGUL.Lib

-- hello: _ <-> Hello!
hello :: Show a => BiGUL a String
hello = Skip (\_ -> "Hello!")
```

```
-:---  test.hs          All L9     Git:oxford-ssbx16  (Haskell FlyC-)
Wrote /Users/hu/Repositories/prl_tokyo/bigul/SummerSchool16/test.hs
```

Put and Its Bidirectional Interpretation

- A putback function:

px :: BiGUL s v

desrcibes how to use the view to update the source.

- Bidirectional Interpretation:

get px :: s → Maybe v
put px :: s → v → Maybe s

```
-- hello: _ <-> Hello!
hello :: Show a => BiGUL a String
hello = Skip (\_ -> "Hello!")
```

```
*Main> get hello 1
Just "Hello!”

*Main> get hello 2
Just "Hello!”

*Main> put hello 2 "Hello!"
Just 2

*Main> put hello 2 "Hello!!!"
Nothing
```

A Quick Tour of BiGUL

1. Skip
2. Replace
3. Product
4. Source/View Rearragement
5. Case

1. Skip

Disallow any change on the view

```
Skip :: (s → v) → BiGUL s v
```

```
*Main> put (Skip square) 10 100
Just 10

*Main> put (Skip square) 10 250
Nothing

*Main> get (Skip square) 5
Just 25
```

```
skip1 :: BiGUL s ()
skip1 = Skip (const ())
```

2. Replace

Use the view to completely replace the source

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

```
*Main> put Replace 1 100
Just 100

*Main> put Replace (1,1) (100,200)
Just (100,200)
```

3. Prod: Production of two puts

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

```
*Main> put (skip1 `Prod` Replace) (5,1) ((),100)
Just (5,100)

*Main> put ((skip1 `Prod` Replace) `Prod` Replace) ((5,1),2) (((),100),200)
Just ((5,100),200)
```

4. Source/View Rearrangements

Rearrange the source/view through a natural transformation tau to make the view and the source have the same structure.

```
$(rearrS [| tau :: s1 -> s2 |]) :: BiGUL s2 v -> BiGUL s1 v

$(rearrV [| tau :: v1 -> v2 |]) :: BiGUL s v2 -> BiGUL s v1
```

```
putPairOverNPair :: (Show s1, Show s2) => BiGUL ((s0,s1),s2) (s1,s2)
putPairOverNPair = $(rearrS [| \((s0,s1),s2) -> (s1,s2) |]) Replace

putPairOverNPair' :: (Show s0, Show s1, Show s2) => BiGUL ((s0,s1),s2) (s1,s2)
putPairOverNPair' =  $(rearrV [| \(v1,v2) -> (((),v1),v2) |]) $
                               (skip1 `Prod` Replace) `Prod` Replace
```

A syntactic sugar:

```
$(update [p| source-pattern |]
         [p| view-pattern |]
         [d| updating-strategy |])
```

```
putPairOverNPair” :: (Show s1, Show s2) => BiGUL ((s0,s1),s2) (s1,s2)
putPairOverNPair” = $(update [p| ((_,s1),s2) |]
                             [p| (s1,s2) |]
                             [d| s1 = Replace; s2 = Replace |])
```

Exercise:
Define pHead to use the view to replace to first element of the source list.

```
pHead :: BiGUL [s] s
```

```
pHead :: Show s => BiGUL [s] s
pHead = $(rearrS [| \(s:_) -> s |]) Replace
```

```
pHead :: Show s => BiGUL [s] s
pHead = $(update [p| s:_ |] [| s |] [d| s = Replace |]
```

Exercise:

Define pNth to use the view to replace to the ith element of the source list.

```
pNth :: Int → BiGUL [s] s
```

```haskell
pNth :: Show s => Int -> BiGUL [s] s
pNth i =  if i == 0 then pHead
          else  $(rearrS [| \(x:xs) -> (x,xs) |]) $
                   $(rearrV [| \v -> ((), v) |]) $
                      skip1 `Prod` pNth (i-1)
```

```
*PBasic> put (pNth 3) [1..10] 100
Just [1,2,3,100,5,6,7,8,9,10]

*PBasic> get (pNth 3) [1..10]
Just 4
```

5. Case

```
Case  [  $(normal   [| enteringCond1  :: s -> v -> Bool |] [|exitCond1 :: s -> Bool |])
           ==> (bx1 :: BiGUL s v)
       ,  $(adaptive [| enteringCond1' :: s -> v -> Bool |])
            ==> (f1 :: s -> v -> s)
       ,  ...
       ,  $(normal   [| enteringCondn  :: s -> v -> Bool |] [|exitCond1 :: s -> Bool |])
            ==> (bxn :: BiGUL s v)
       ,  ...
       ,  $(adaptive [| enteringCondm' :: s -> v -> Bool |])
            ==> (fm :: s -> v -> s)
       ]
    :: BiGUL s v
```

```
pHead :: Show s => BiGUL [s] s
pHead = $(rearrS [| \(s:_) -> s |]) Replace
```

```
repHead :: BiGUL [Int] Int
repHead = Case [
  $(normal [| \s v -> length s > 0 |] [| \s -> length s > 0 |])
    ==> $(rearrS [| \(x:_) -> x |]) Replace,
  $(adaptive [| \s v -> length s == 0 |])
    ==> \s v -> [0]
 ]
```

Exercise

Define a safe embedding of a pair of well-behaved get and put as a putback transformation.

```
emb :: Eq v => (s -> v) -> (s -> v -> s) -> BiGUL s v
emb g p = Case [ $(normal [| \s v -> g s == v |] [p| _ |])
                       ==> Skip g
                  , $(adaptive [| \s v -> {- g s /= v -} True |])
                       ==> p
                  ]
```

```
distSum :: BiGUL (Int, Int) Int
distSum = emb g p
  where  g (x,y) = x+y;  p (x,y) v = (v-y,y)
```