Principle and Practice of Putback-based Bidirectional Programming in BiGUL

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

BX Summer School, Oxford
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:
\[ \text{put } s \ (\text{get } s) = s \]

Put-Get:
\[ \text{get } (\text{put } s \ t) = t \]
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

- 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 \((w,h) h' \mid h = h' = (w,h) \mid \text{otherwise } = (3,h')\)
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:

```
src → tgt → src'
```

```
tgt → src ← tgt'
```

**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'.

\[
\text{get s} \\
\Rightarrow \{ \text{GetPut} \} \\
\text{get (put s (get' s))} \\
\Rightarrow \{ \text{PutGet} \} \\
\text{get' s}
\]
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

\[
\text{get } s = v \text{ such that } \text{put } s v = s
\]

satisfies GetPut and PutGet.

Proof:
\[
\begin{align*}
\text{get (put } s v) &= \text{condition for put} \\
\text{put } s (\text{get } s) &= \text{definition of get}
\end{align*}
\]
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

\[
\text{get } s = v \text{ such that } \text{put } s \ v = s
\]

satisfies GetPut and PutGet.

**Proof**:

\[
\begin{align*}
\text{get } (\text{put } s \ v) &= \{ \text{condition for put} \} \text{get } s \\
&= \{ \text{definition of get} \} v \\
\end{align*}
\]

\[
\begin{align*}
\text{put } s \ (\text{get } s) &= \{ \text{definition of get} \} \text{put } s \ v \\
&= \{ \text{condition for put} \} s \\
\end{align*}
\]
Remark

**Lemma**: if there exists a $v$ satisfying $\text{put} \ s' \ v = s$, then so does $\text{put} \ s \ v = s$.

**Proof**: 

$$\text{put} \ s' \ v = s$$

$$\Rightarrow \ \{ \text{put} \}$$

$$\text{put} \ (\text{put} \ s' \ v) \ v = \text{put} \ s \ v$$

$$\equiv \ \{ \text{PutTwice: forall } s \ v. \ \text{put} \ (\text{put} \ s \ v) \ v = \text{put} \ s \ v \}$$

$$\text{put} \ s' \ v = \text{put} \ s \ v$$

$$\equiv \ \{ \text{Assumption: put} \ s' \ v = s \}$$

$$s = \text{put} \ s \ v$$
Well-behaved “put”

**Definition:** A “put” function is said to be well-behaved, if there exists a (unique) “get” function such that $\text{Get}(\text{Put}(s, v)) = v$ holds.

**Exercise**
Which of the following puts are well-behaved?

1. $\text{put1}(s, v) = s$
2. $\text{put2}(s, v) = v$
3. $\text{put2}(s, v) = v+1$
Well-behaved “put”

**Lemma:**

put is well-behaved, *iff*

1. View-deterministic
   
   \[
   \text{put} \ s1 \ v1 = \text{put} \ s2 \ v2 \ \Rightarrow \ v1 = v2
   \]

2. View-stable
   
   for any \( s \), there exists a \( v \), such that \( \text{put} \ s \ v = s \)

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.

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/
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/](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
{-# 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!")
Put and Its Bidirectional Interpretation

- A putback function:
  \[ px :: \text{BiGUL} \; s \; v \]
  describes how to use the view to update the source.

- Bidirectional Interpretation:
  \[ \text{get} \; px :: s \rightarrow \text{Maybe} \; v \]
  \[ \text{put} \; px :: s \rightarrow v \rightarrow \text{Maybe} \; s \]
-- 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 Rearrangement
5. Case
1. Skip

Disallow any change on the view

\[
\text{Skip} :: (s \to v) \to \text{BiGUL} \ s \ v
\]

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

*Main> put (Skip square) 10 250
Nothing

*Main> get (Skip square) 5
Just 25

\[
\text{skip1} :: \text{BiGUL} \ s \ ()
\text{skip1} = \text{Skip} \ (\text{const} \ ())
\]
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 \rightarrow BiGUL s2 v2 \\
\rightarrow 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.

\[
\text{putPairOverNPair} :: \text{(Show } s_0, \text{ Show } s_1, \text{ Show } s_2) \Rightarrow \text{BiGUL } ((s_0,s_1),s_2) (s_1,s_2)
\]

\[
\text{putPairOverNPair} = \text{(rearrS [l \((s_0,s_1),s_2\) \(\rightarrow\) \((s_1,s_2)\) l]) Replace}
\]

\[
\text{putPairOverNPair'} :: \text{(Show } s_0, \text{ Show } s_1, \text{ Show } s_2) \Rightarrow \text{BiGUL } ((s_0,s_1),s_2) (s_1,s_2)
\]

\[
\text{putPairOverNPair'} = \text{(rearrV [l \((s_1,v_2)\) \(\rightarrow\) \((((()),v_1),v_2)\) l])}$
\[
\quad \text{(skip1 } \text{`Prod` Replace) `Prod` Replace}
\]
A syntactic sugar:

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

\[
putPairOverNPair'' :: (Show s1, Show s2) \Rightarrow BiGUL ((s0,s1),s2) (s1,s2)
putPairOverNPair'' = $(update \[p| (\_,s1),s2) \mid\]
 [p| (s1,s2) \mid]
 [d| s1 = Replace; s2 = Replace \mid])
\]
Exercise:
Define pHead to use the view to replace to first element of the source list.

\[ pHead :: \text{BiGUL} [s] s \]

\[ pHead :: \text{Show } s \Rightarrow \text{BiGUL} [s] s \]
\[ pHead = \$(\text{rearrS} [l \ \langle s :_\rangle \to s \ ])) \text{ Replace} \]

\[ pHead :: \text{Show } s \Rightarrow \text{BiGUL} [s] s \]
\[ pHead = \$(\text{update} [p| s :_\rangle | l \ s \ ] [d| s = \text{Replace} \ ] ) \]
Exercise:
Define \( \text{pNth} \) to use the view to replace to the \( i \)th element of the source list.

\[
\text{pNth} :: \text{Int} \rightarrow \text{BiGUL } [s] \ s
\]

\[
\text{pNth} :: \text{Show } s \Rightarrow \text{Int} \rightarrow \text{BiGUL } [s] \ s
\]

\[
\text{pNth } i = \text{if } i = 0 \text{ then } \text{pHead}
\]
\[
\text{else } (\text{rearrS } [l \ (x:xs) \rightarrow (x, xs) \ l]) \ (\text{rearrV } [l \ \text{v} \rightarrow () \ v \ l]) \ (\text{skip1 } \text{`Prod`} \ \text{pNth } (i-1))
\]

*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 [ $\text{(normal [l enteringCond1 :: s \to v \to \text{Bool} |] [exitCond1 :: s \to \text{Bool} |])}$

$\Rightarrow (bx_1 :: \text{BiGUL s v})$

, $\text{(adaptive [l enteringCond1' :: s \to v \to \text{Bool} |])}$

$\Rightarrow (f_1 :: s \to v \to s)$

, ...

, $\text{(normal [l enteringCondn :: s \to v \to \text{Bool} |] [exitCond1 :: s \to \text{Bool} |])}$

$\Rightarrow (bx_n :: \text{BiGUL s v})$

, ...

, $\text{(adaptive [l enteringCondn' :: s \to v \to \text{Bool} |])}$

$\Rightarrow (f_m :: s \to v \to s)$

] $:: \text{BiGUL s v}$
pHead :: Show s => BiGUL [s] s
pHead = $(rearrS [l \(s::_) -> s |]) Replace

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

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

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