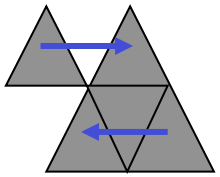
Principle and Practice of Putback-based Bidirectional Programming in BiGUL

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

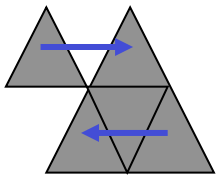
BX Summer School, Oxford
July 28, 2016



Overview

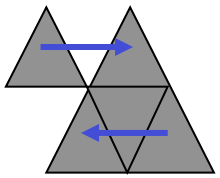
- Lecture 1: Introduction to BiGUL
  - Why is putback-based BX?
  - What is its foundation?
  - How to program in BiGUL?
- Lecture 2: Into BiGUL's Bidirectionality
  - How is BiGUL implemented?
- Lecture 3: Three Applications using BiGUL
  - Matching/delta lenses in BiGUL
  - Bidirectionalize relational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

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



NII 大学共同利用機関法人 情報・システム研究機構 国立情報学研究所 National Institute of Informatics

Bidirectionalization of Functions on Lists

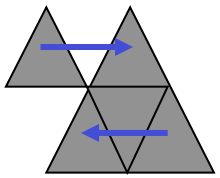


Foldr

- Foldr is one of the important higher order functions for defining useful functions on lists:

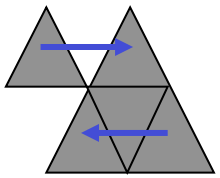
```
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

- Examples:
  - sum = foldr (+) 0
  - map f = foldr (\a r -> f a : r) []
  - reverse p = foldr (\a r -> r ++ [a]) []



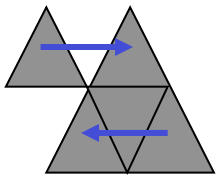
lensFoldr

```
lensFoldr :: (Show a, Show b)
          => BiGUL (a, b) b -> (b->Bool) -> BiGUL ([a], b) b
```

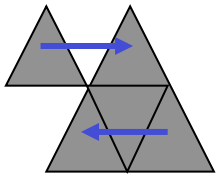
lensFoldr

```
lensFoldr :: (Show a, Show b)
          => BiGUL (a, b) b -> (b->Bool) -> BiGUL ([a], b) b
lensFoldr bx pv =
  Case  [


        ,  $(normal [| \(s,_) v -> null s |] [| \(s,_) -> null s |])


        ,  $(normalSV [p| _ |] [p| _ |] [| \(s,_) -> not (null s) |])


]
```

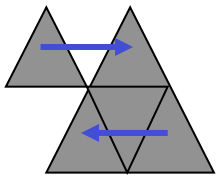
lensFoldr

```
lensFoldr :: (Show a, Show b)
          => BiGUL (a, b) b -> (b->Bool) -> BiGUL ([a], b) b
lensFoldr bx pv =
  Case  [



        ,  $(normal [| \(s,_) v -> null s |] [| \(s,_) -> null s |])
             ==> $(update [p| (_, v) |] [p| v |] [d| v = Replace |])

        ,  $(normalSV [p| _ |] [p| _ |] [| \(s,_) -> not (null s) |])



]
```

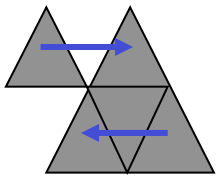
lensFoldr

```
lensFoldr :: (Show a, Show b)
          => BiGUL (a, b) b -> (b->Bool) -> BiGUL ([a], b) b
lensFoldr bx pv =
  Case  [


        ,  $(normal [| \(s,_) v -> null s |] [| \(s,_) -> null s |])
           ==> $(update [p| (_, v) |] [p| v |] [d| v = Replace |])

        ,  $(normalSV [p| _ |] [p| _ |] [| \(s,_) -> not (null s) |])
           ==> $(rearrS [| \((x:xs), e) -> (x, (xs,e))  |])
                  (Replace `Prod` lensFoldr bx pv) `Compose` bx
       ]
```

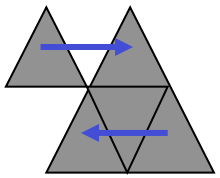
lensFoldr

```
lensFoldr :: (Show a, Show b)
          => BiGUL (a, b) b -> (b->Bool) -> BiGUL ([a], b) b
lensFoldr bx pv =
  Case


        ,  $(normal [| \(s,_) v -> null s |] [| \(s,_) -> null s |])
             ==> $(update [p| (_, v) |] [p| v |] [d| v = Replace |])

        ,  $(normalSV [p| _ |] [p| _ |] [| \(s,_) -> not (null s) |])
             ==> $(rearrS [| \((x:xs), e) -> (x, (xs,e))  |])
                    (Replace `Prod` lensFoldr bx pv) `Compose` bx
      ]
```

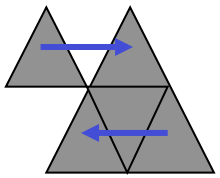
lensFoldr

```
lensFoldr :: (Show a, Show b)
          => BiGUL (a, b) b -> (b->Bool) -> BiGUL ([a], b) b
lensFoldr bx pv =
  Case  [  $(adaptive [| \(s,?) v -> pv v && length s /= 0 |])
           ==> \(s,y) _ -> ([],y)

        ,  $(normal [| \(s,_) v -> null s |] [| \(s,_) -> null s |])
           ==> $(update [p| (_, v) |] [p| v |] [d| v = Replace |])

        ,  $(normalSV [p| _ |] [p| _ |] [| \(s,_) -> not (null s) |])
           ==> $(rearrS [| \((x:xs), e) -> (x, (xs,e))  |])
                 (Replace `Prod` lensFoldr bx pv) `Compose` bx
        ]
```

lensMap

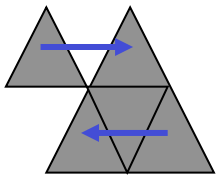
```
lensMap :: (Show a, Show b)
          => BiGUL a b -> BiGUL ([a],[b]) [b]
lensMap bx =  lensFoldr bx' null
   where  bx` =  $(update [p| (a,b) |] [| (a : b) |] [d| a = bx; b = Replace)
```

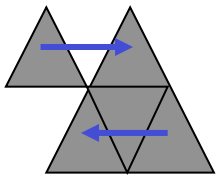
LensReverse

Exercise:
Can you bidirectionalize reverse using lensFoldr?

```
lensReverse :: Show a => BiGUL [a] [a]
lensReverse =  Case  [
                      $(adaptive [| \s v -> length s < length v |])
                        ==> \s v -> v

                    , $(normalSV [p| _ |] [p| _ |] [| \ s -> True |])
                        ==> $(rearrS [| \s -> (s,[]) |]) $
                              lensFoldr (lensSwap `Compose` lensSnoc) null
                  ]
```

Bidirectionalizing relational queries with BiGUL



View Updating for Relational Database

Track	Date	Rating	Album	Quantity
Lullaby	1989	3	Galore	2
Lullaby	1989	3	Show	3
Lovesong	1989	5	Galore	2
Lovesong	1989	5	Paris	4
Trust	1992	4	Wish	5

Q

Track	Rating	Album	Quantity
Lullaby	3	Show	3
Lovesong	5	Paris	4
Trust	4	Wish	5

?

Q

Track	Rating	Album	Quantity
Lullaby	4	Show	3
Lovesong	5	Disintegration	7

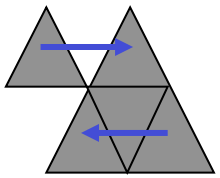
Q: select Track, Rating, Album, Quantity as v
from s where Quantity > 2

Brul [BX 2015]

- Two library functions to describe "update policies" using put:
  - relAlign: dealing with selection/projection
  - unJoin: dealing with join

- Features
  - All SPJ queries can be derived (query-completeness)
  - All update policies discussed in the literature can be described (expressiveness)

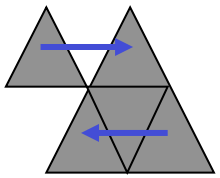
We will focus on relAlign in this tutorial.



Relational Table

```
type RT = [Record]
type Record = [RType]
data RType = RInt Int | RString String | RFloat Float | RDouble Double
      deriving (Show, Eq, Ord)

deriveBiGULGeneric  ''RType
```

```
s  =  [ [RString "Lullaby",  RInt 1989, RInt 3, RString "Galore", RInt 1]
      , [RString "Lullaby",  RInt 1989, RInt 3, RString "Show"   , RInt 3]
      , [RString "Lovesong", RInt 1989, RInt 5, RString "Galore", RInt 1]
      , [RString "Lovesong", RInt 1989, RInt 5, RString "Disintegration" , RInt 4]
      , [RString "Trust",    RInt 1992, RInt 4, RString "Wish"   , RInt 5]
      ]
```

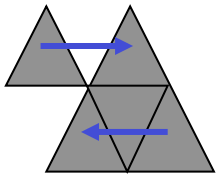



Functional Dependency

Track	Date	Rating	Album	Quantity
Lullaby	1989	3	Galore	2
Lullaby	1989	3	Show	3
Lovesong	1989	5	Galore	2
Lovesong	1989	5	Paris	4
Trust	1992	4	Wish	5

Track → Data, Rating Album → Quantity

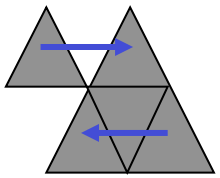
```
type FDMap = Map.Map Int [Int]

sfdMap :: FDMap
sfdMap = Map.fromList [(0,[1,2]), (3,[4])]
```



From keyAlign to relAlign

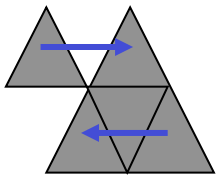
- relAlign is similar to keyAlign except that
  - We need to consider filtering of source elements
  - We need to recover function dependency when updates happen

Idea: keyAlign → pAlign → relAlign



Review: keyAlign

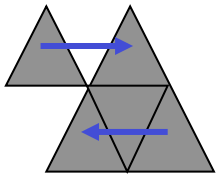
```
keyAlign  ::  forall s v k. (Show s, Show v, Eq k)
          =>  (s -> k) -> (v -> k) -> BiGUL s v -> (v -> s) -> BiGUL [s] [v]
keyAlign ks kv b c = Case
  [ $(normalSV [p| [] |] [p| [] |] [p| [] |])
    ==> $(update [p| [] |] [p| [] |] [d| |])
  , $(normal [| \(s:ss) (v:vs) -> ks s == kv v |] [p| _ : _ |])
    ==> $(update [p| x:xs |]  [p| x:xs |]
                              [d| x = b; xs = keyAlign ks kv b c |])
  , $(adaptiveSV [p| _ : _ |] [p| [] |])
    ==> \ _ _ -> []
  , $(adaptive [| \ss (v:vs) -> kv v `elem` map ks ss |])
    ==> \ss (v : _) -> uncurry (:) (extract (kv v) ss)
  , $(adaptiveSV [p| _ |] [p| _ : _ |])
    ==> \ss (v : _) -> c v : ss
  ]
```

From keyAlign to pAlign


```
pAlign :: forall s v k. (Show s, Show v, Eq k)
        => (s -> Bool) {- predicate -}
           -> (s -> k) -> (v -> k) -> BiGUL s v -> (v -> s)
           -> (s -> Maybe s) {- conceal/hide function -}
           -> BiGUL [s] [v]
pAlign p ks kv b c h = Case
  [ $(normalSV [p| [] |] [p| [] |] [p| [] |])
    ==> $(update [p| [] |] [p| [] |] [d| |])
  , $(normal [| \(s:ss) (v:vs) -> p s && ks s == kv v |] [| \(s:ss) -> p s |])
    ==> $(update [p| x:xs |] [p| x:xs |] [d| x = b; xs = pAlign p ks kv b c h |])
  , $(adaptive [| \(s:ss) v -> p s && null v|])
    ==> \(s:ss) v -> maybe [] (:[]) (h s) ++ ss
  , $(normal [| \(s:ss) v -> not (p s) |] [| \(s:ss) -> not (p s) |])
    ==> $(update [p| _:xs |] [p| xs |] [d| xs = pAlign p ks kv b c h |])
  , $(adaptive [| \ss (v:vs) -> kv v `elem` map ks (filter p ss) |])
    ==> \ss (v:_) -> uncurry (:) (extract (kv v) ss)
  , $(adaptiveSV [p| _ |] [p| _:_ |])
    ==> \ss (v:_) -> filterCheck p (c v) : ss
  ]
```

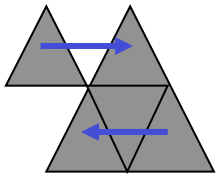
Add 2 new parameters



From keyAlign to pAlign

```
pAlign :: forall s v k. (Show s, Show v, Eq k)
        => (s -> Bool) {- predicate -}
           -> (s -> k) -> (v -> k) -> BiGUL s v -> (v ->
           -> (s -> Maybe s) {- conceal/hide functio
           -> BiGUL [s] [v]
pAlign p ks kv b c h = Case
  [ $(normalSV [p| [] |] [p| [] |] [p| [] |])
    ==> $(update [p| [] |] [p| [] |] [d| |])
  , $(normal [| \(s:ss) (v:vs) -> p s && ks s == kv v |] [| \(s:ss) -> p s |])
    ==> $(update [p| x:xs |] [p| x:xs |] [d| x = b; xs = pAlign p ks kv b c h |])
  , $(adaptive [| \(s:ss) v -> p s && null v|])
    ==> \(s:ss) v -> maybe [] (:[]) (h s) ++ ss
  , $(normal [| \(s:ss) v -> not (p s) |] [| \(s:ss) -> not (p s) |])
    ==> $(update [p| _:xs |] [p| xs |] [d| xs = pAlign p ks kv b c h |])
  , $(adaptive [| \ss (v:vs) -> kv v `elem` map ks (filter p ss) |])
    ==> \ss (v:_) -> uncurry (:) (extract (kv v) ss)
  , $(adaptiveSV [p| _ |] [p| _:_ |])
    ==> \ss (v:_) -> filterCheck p (c v) : ss
  ]
```

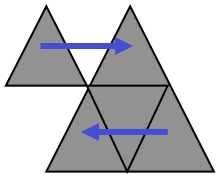
Consider the cases according to “p on source elements”



From keyAlign to pAlign

```
pAlign :: forall s v k. (Show s, Show v, Eq k)
        => (s -> Bool) {- predicate -}
           -> (s -> k) -> (v -> k) -> BiGUL s v -> (v ->
           -> (s -> Maybe s) {- conceal/hide functio
           -> BiGUL [s] [v]
pAlign p ks kv b c h = Case
  [ $(normalSV [p| [] |] [p| [] |] [p| [] |])
    ==> $(update [p| [] |] [p| [] |] [d| |])
  , $(normal [| \(s:ss) (v:vs) -> p s && ks s == kv v |          (s:ss) -> p s |])
    ==> $(update [p| x:xs |] [p| x:xs |] [d| x = b;                pAlign p ks kv b c h |])
  , $(adaptive [| \(s:ss) v -> p s && null v|])
    ==> \(s:ss) v -> maybe [] (:[]) (h s) ++ ss
  , $(normal [| \(s:ss) v -> not (p s) |] [|          ss) -> not (p s) |])
    ==> $(update [p| _:xs |] [p| xs |] [d|      xs = pAlign p ks kv b c h |])
  , $(adaptive [| \ss (v:vs) -> kv v `elem` map ks (filter p ss) |])
    ==> \ss (v:_) -> uncurry (:) (extract (kv v) ss)
  , $(adaptiveSV [p| _ |] [p| _:_ |])
    ==> \ss (v:_) -> filterCheck p (c v) : ss
  ]
```

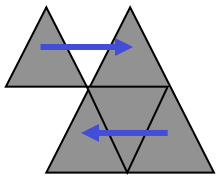
Guarantee the new source elements satisfying p.



From pAlign to relAlign

```
relAlign ::  … -> (s -> s) {- dependency maintaining function -}
             -> BiGUL [s] [v]
relAlign p ks kv b c h fd = Case
  [ …
   , $(adaptive [| \(s:ss) v -> p s && null v|])
     ==> \(s:ss) v -> maybe [] ((:[]) . filterCheck p . fd) (h s) ++ ss
  , $(normal [| \(s:ss) v -> not (p s) |] [| \(s:ss) -> not (p s) |])
     ==> Case
      [ $(adaptive [| \(s:_) _ -> fd s /= s |])
          ==> \(s:ss) _ -> filterCheck (not.p) (fd s) : ss
      , $(normal [|\_ _ -> True |] [| const True |])
          ==> $(update [p| _:xs |] [p| xs |] [d| xs = relAlign p ks kv b c h fd |])
      ]
      …
]
```

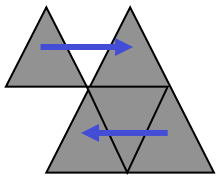
Perform fd on those source elements that does not satisfy p or has no matching element in the view.



Use of relAlign

Define an update policy: if an element is removed from the view, the corresponding source elements should be removed.

```
u0 :: RType -> (Record -> Record) -> BiGUL [Record] [Record]
u0 d =
  relAlign
    (\r -> (r !! 4) > RInt 2)
    (\s -> (s !! 0, s!!3))
    (\v -> (v !! 0, v !! 2))
    $(update [p| (t: _: r: a: q: [])|]
             [p| (t: r: a: q: []) |]
             [d| t = Replace; r = Replace; a = Replace; q = Replace |])
    (\(t: r: a: q: []) -> (t: d: r: a: q: []))
    (\rs -> Nothing)
```

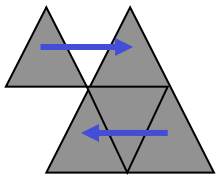



Testing the forward transformation:

```
*Brul> get (u0 (RInt (-1)) id) s
Just
[[RString "Lullaby",RInt 3,RString "Show",RInt 3],
[RString "Lovesong",RInt 5,RString "Disintegration",RInt 4],
[RString "Trust",RInt 4,RString "Wish",RInt 5]]
```

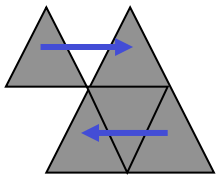
Suppose that the view is changed to:

```
v =  [ [RString "Lullaby" , RInt 4, RString "Show"   , RInt 3]
     , [RString "Lovesong", RInt 5, RString "Disintegration" , RInt 7]
     ]
```



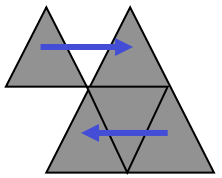
Testing the backward transformation:

```
*Brul> put (u0 (RInt (-1)) (fdFun sfdMap vfdMap svMap v)) s v
Just
[[RString "Lullaby",RInt 1989,RInt 4,RString "Galore",RInt 1],
[RString "Lullaby",RInt 1989,RInt 4,RString "Show",RInt 3],
[RString "Lovesong",RInt 1989,RInt 5,RString "Galore",RInt 1],
[RString "Lovesong",RInt 1989,RInt 5,RString "Disintegration",RInt 7]]
```

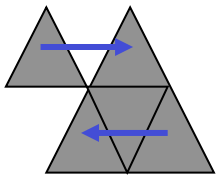
Define a new update policy: if an element is removed from the view, the corresponding source elements will have quantity 0.

```
u0 :: RType -> (Record -> Record) -> BiGUL [Record] [Record]
u0 d =
  relAlign
    (\r -> (r !! 4) > RInt 2)
    (\s -> (s !! 0, s!!3))
    (\v -> (v !! 0, v !! 2))
    $(update [p| (t: _: r: a: q: [])|]
             [p| (t: r: a: q: []) |]
             [d| t = Replace; r = Replace; a = Replace; q = Replace |])
    (\(t: r: a: q: []) -> (t: d: r: a: q: []))
   (\(t: d: r: a: _: []) -> Just (t: d: r: a: Rint 0:[]))
```

Summary

- Putback-based programming is fun and not that difficult.
- Putback-based BX can be served as the basis for developing various domain specific BX languages.
  - BiYacc/BiFluX/Brul …
- Putback-based BX is promising to be used to develop BIG BX systems:
  - Bidirectionalization of pandoc is ongoing



Future Work

- Adding effects to BiGUL
- BiGUL for graphs (GRoundTram ...)
- Static analysis (validatity checking) of BiGUL programs
- Putback-based bidirectional computing machine and compilation