Semantic Bidirectionalization and the Constant-Complement Perspective

Janis Voigtländer

University of Bonn

BT-in-ABC’10
Bidirectional Transformation

$source \xrightarrow{get} view
s \xrightarrow{get} v$
Bidirectional Transformation

source $s$ \quad \xrightarrow{\text{get}} \quad \text{view} \quad v \quad \xrightarrow{\text{update}} \quad v'$
Bidirectional Transformation

source

\[ s \]

\[ s' \]

get

view

\[ v \]

\[ v' \]

update

put
Bidirectional Transformation

source

\( s \)

\( s' \)

view

\( v \)

\( v' \)

get

put

update
Bidirectional Transformation

source

\[ s \]

\[ \text{get} \]

\[ \rightarrow \]

view

\[ v \]

Acceptability / GetPut
Bidirectional Transformation

Acceptability / GetPut
Bidirectional Transformation

source $s$ \hspace{3cm} get \hspace{3cm} view $v$

$\downarrow$ update $\uparrow$

$s'$ \hspace{3cm} put \hspace{3cm} $v'$

Consistency / PutGet
Bidirectional Transformation

source

view

Consistency / PutGet
Bidirectional Transformation

source

view

s

get

put

s′

v

update

ν′
Bidirectional Transformation

source

\[ s \]

view

\[ v \]

update

\[ v' \]

\[ s' \]

get

put

Lenses, DSLs

[Foster et al., ACM TOPLAS'07, …]
Bidirectional Transformation

Bidirectionalization

[Matsuda et al., ICFP’07]
Bidirectional Transformation

source

view

\( s \)

\( v \)

\( s' \)

\( v' \)

get

update

put

Syntactic Bidirectionalization

[Matsuda et al., ICFP’07]
Bidirectional Transformation

source

\[ s \]

\[ s' \]

view

\[ v \]

\[ v' \]

Semantic Bidirectionalization

get

put

update

?
Semantic Bidirectionalization

[V., POPL’09]
Semantic Bidirectionalization

**Aim:** Write a higher-order function $\texttt{bff}$ such that any $\texttt{get}$ and $\texttt{bff get}$ satisfy $\texttt{GetPut}$, $\texttt{PutGet}$, . . . .
Semantic Bidirectionalization

Aim: Write a higher-order function $\texttt{bff}^\dagger$ such that any $\texttt{get}$ and $\texttt{bff get}$ satisfy GetPut, PutGet, 

$^\dagger$ “Bidirectionalization for free!”
Semantic Bidirectionalization

Aim: Write a higher-order function \texttt{bff} such that any \texttt{get} and \texttt{bff get} satisfy GetPut, PutGet, ....

Examples:

```
"abc"  tail  "bc"
```
Semantic Bidirectionalization

**Aim:** Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ....

**Examples:**

```
"abc"  tail  "bc"
```

```
"ade"  "de"
```

† “Bidirectionalization for free!”
Semantic Bidirectionalization

Aim: Write a higher-order function $bff^\dagger$ such that any $get$ and $bff\ get$ satisfy $GetPut$, $PutGet$, ....

Examples:

```
"abc" $\xrightarrow{\text{tail}}$ "bc"
"ade" $\xrightarrow{\text{bff\ tail}}$ "de"
```

$^\dagger$ “Bidirectionalization for free!”
Semantic Bidirectionalization

**Aim:** Write a higher-order function \( \texttt{bff} \uparrow \) such that any \texttt{get} and \texttt{bff get} satisfy GetPut, PutGet, . . . .

**Examples:**

```
'a' 'b' 'a' 'c'
```

flatten

```
'abac'
```

† “Bidirectionalization for free!”
Semantic Bidirectionalization

Aim: Write a higher-order function \( \text{bff}^\dagger \) such that any get and \( \text{bff get} \) satisfy GetPut, PutGet, . . . .

Examples:

\[ \text{\texttt{a}} \quad \text{\texttt{b}} \quad \text{\texttt{a}} \quad \text{\texttt{c}} \]

\( \text{\texttt{abac}} \)

\[ \text{\texttt{a}} \quad \text{\texttt{b}} \quad \text{\texttt{x}} \quad \text{\texttt{c}} \]

\( \text{\texttt{abxc}} \)

\( \text{\texttt{flatten}} \)

\( \text{\texttt{abac}} \)

\( \text{\texttt{update}} \)

\( \text{\texttt{abxc}} \)

\( \dagger \) “Bidirectionalization for free!”
Semantic Bidirectionalization

Aim: Write a higher-order function $\text{bff}^+$ such that any `get` and $\text{bff get}$ satisfy GetPut, PutGet, ....

Examples:

```
'a' 'b' 'a' 'c'
```

$\text{flatten}$

```
'abac'
```

$\text{update}$

```
'a' 'b' 'x' 'c'
```

$\text{bff flatten}$

```
'abxc'
```

† “Bidirectionalization for free!”
Semantic Bidirectionalization

**Aim:** Write a higher-order function $\text{bff}^\dagger$ such that any $\text{get}$ and $\text{bff get}$ satisfy GetPut, PutGet, ....

**Examples:**

```
     'a' 'b' 'a' 'c'
    /   \
   /     \
'nub ◦ flatten' → "abc"
```

$^\dagger$ “Bidirectionalization for free!”
Aim: Write a higher-order function $\text{bff}^\dagger$ such that any $\text{get}$ and $\text{bff get}$ satisfy GetPut, PutGet, . . . .

Examples:

$$
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{a} \\
\text{c}
\end{array}
\xrightarrow{\text{nub} \circ \text{flatten}}
\text{"abc"}
\xrightarrow{\text{update}}
\text{"xbc"}
$$
Semantic Bidirectionalization

**Aim:** Write a higher-order function $\text{bff}^\dagger$ such that any $\text{get}$ and $\text{bff get}$ satisfy $\text{GetPut}$, $\text{PutGet}$, . . . .

**Examples:**

```
'ab' 'ba' 'cb'
\(\text{nub} \circ \text{flatten}\)
\(\text{bff} (\text{nub} \circ \text{flatten})\)
```

† “Bidirectionalization for free!”
Overview of the Bidirectionalization Method

`b' 'a' 'c' 'a'
0 → 'b'
1 → 'a'
2 → 'c'
3 → 'a'

"aca"
0 → 'b'
1 → 'x'
2 → 'c'
3 → 'a'
1 → 'x'
2 → 'c'
3 → 'a'

? "xca"
Overview of the Bidirectionalization Method

'b' 'a' 'c' 'a'
0 → 'b'
1 → 'a'
2 → 'c'
3 → 'a'

'x' 'c' 'a'
0 → 'b'
1 → 'x'
2 → 'c'
3 → 'a'
1 → 'x'
2 → 'c'
3 → 'a'

? "xca"bff (\text{tail} \circ \text{flatten})
v′

3
Overview of the Bidirectionalization Method

```
s
  'b' 'a' 'c' 'a'

0 1 2 3

```

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
'b' & 'a' & 'c' & 'a' \\
\end{array} \]
Overview of the Bidirectionalization Method

- The diagram shows a tree structure with nodes labeled 'b', 'a', 'c', and 'a'.
- The tree has a root labeled 't' and another labeled 's'.
- The labels '0', '1', '2', and '3' are associated with each node, indicating the direction of traversal.
- The table on the right side lists the mappings:
  - 0 → 'b'
  - 1 → 'a'
  - 2 → 'c'
  - 3 → 'a'
- The label 'g' is associated with the mapping.
- The text 'xa' is highlighted.
Overview of the Bidirectionalization Method

t
0 1 2 3
\[v']
\]
get t

tail \circ flatten

\[\{1, 2, 3\}\]

g
\[
\begin{array}{c}
0 \rightarrow 'b' \\
1 \rightarrow 'a' \\
2 \rightarrow 'c' \\
3 \rightarrow 'a'
\end{array}
\]

"xca"
Overview of the Bidirectionalization Method

\( \text{tail} \circ \text{flatten} \)

\[
\begin{array}{c}
0 \rightarrow 'b' \\
1 \rightarrow 'a' \\
2 \rightarrow 'c' \\
3 \rightarrow 'a'
\end{array}
\]

\[
\begin{array}{c}
1 \rightarrow 'x' \\
2 \rightarrow 'c' \\
3 \rightarrow 'a'
\end{array}
\]

\[
\begin{array}{c}
'x' \text{ ca}
\end{array}
\]
Overview of the Bidirectionalization Method

\[ t \]

\[ s \]

The diagram illustrates the bidirectionalization method, showing the transformation of a tree structure into a flattened list. The tree is labeled with the sequence 'b' 'a' 'c' 'a', and the flattened list is [1, 2, 3] 'aca'. The process involves applying operations such as `tail ◦ flatten` to the tree structure, resulting in the desired flattened list.
Overview of the Bidirectionalization Method

\[ t \]

\[ \text{tail} \circ \text{flatten} \]

\[ [1,2,3] \]

\[ \text{get } t \]

\[ \text{s} \]

\[ \text{get } t' \]

\[ h' \]

\[ h \]

\[ \text{v'} \]

\[ \text{v} \]

\[ \text{aca} \]

\[ \text{b} \quad \text{x} \quad \text{c} \quad \text{a} \]

\[ \text{b} \quad \text{x} \quad \text{c} \quad \text{a} \]

\[ \text{b} \quad \text{x} \quad \text{c} \quad \text{a} \]
The Constant-Complement Approach
[Bancilhon & Spyratos, ACM TODS’81]

In general, given

\[
\text{get} :: S \rightarrow V
\]
The Constant-Complement Approach
[Bancilhon & Spyrtatos, ACM TODS’81]

In general, given

\[\text{get} :: S \rightarrow V\]

define a \( V^C \) and

\[\text{compl} :: S \rightarrow V^C\]

Important: \( \text{compl} \) should "collapse" as much as possible.
The Constant-Complement Approach
[Bancilhon & Spyraitos, ACM TODS’81]

In general, given

\[ \text{get} :: S \rightarrow V \]

define a \( V^c \) and

\[ \text{compl} :: S \rightarrow V^c \]

such that

\[ \lambda s \rightarrow (\text{get } s, \text{compl } s) \]

is injective

Important: \( \text{compl} \) should "collapse" as much as possible.
The Constant-Complement Approach
[Bancilhon & Spyropoulos, ACM TODS’81]

In general, given
\[
\text{get} :: S \rightarrow V
\]
define a \( V^c \) and
\[
\text{compl} :: S \rightarrow V^c
\]
such that
\[
\lambda s \rightarrow (\text{get } s, \text{compl } s)
\]
is injective and has an inverse
\[
\text{inv} :: (V, V^c) \rightarrow S
\]
Important: \( \text{compl} \) should "collapse" as much as possible.
The Constant-Complement Approach
[Bancilhon & Spyrouatos, ACM TODS’81]

In general, given

\[
g\text{et} :: S \rightarrow V
\]

define a \( V^C \) and

\[
\text{compl} :: S \rightarrow V^C
\]

such that

\[
\lambda s \rightarrow (\text{get } s, \text{compl } s)
\]

is injective and has an inverse

\[
\text{inv} :: (V, V^C) \rightarrow S
\]

Then:

\[
\text{put} :: S \rightarrow V \rightarrow S
\]

\[
\text{put } s \nu' = \text{inv } (\nu', \text{compl } s)
\]
The Constant-Complement Approach
[Bancilhon & Spyridos, ACM TODS’81]

In general, given

\[ \text{get} :: S \rightarrow V \]

define a \( V^C \) and

\[ \text{compl} :: S \rightarrow V^C \]

such that

\[ \lambda s \rightarrow (\text{get } s, \text{compl } s) \]

is injective and has an inverse

\[ \text{inv} :: (V, V^C) \rightarrow S \]

Then:

\[ \text{put} :: S \rightarrow V \rightarrow S \]
\[ \text{put } s \; v' = \text{inv } (v', \text{compl } s) \]

Important: \text{compl} should “collapse” as much as possible.
The Constant-Complement Approach

For a very simple setting,

\[
\text{get} :: [\alpha] \rightarrow [\alpha],
\]

what should be \( V^C \) and

\[
\text{compl} :: [\alpha] \rightarrow V^C
\]
The Constant-Complement Approach

For a very simple setting,

\[ \text{get} :: [\alpha] \rightarrow [\alpha], \]

what should be \( V^C \) and

\[ \text{compl} :: [\alpha] \rightarrow V^C \]

To make

\[ \lambda s \rightarrow (\text{get } s, \text{compl } s) \]

injective, need to record information discarded by \text{get}. 
The Constant-Complement Approach

For a very simple setting,

\[
\text{get} :: [\alpha] \rightarrow [\alpha],
\]

what should be \(V^C\) and

\[
\text{compl} :: [\alpha] \rightarrow V^C \quad ???
\]

To make

\[
\lambda s \rightarrow (\text{get } s, \text{compl } s)
\]

injective, need to record information discarded by get.

Candidates:

1. length of the source list
The Constant-Complement Approach

For a very simple setting,

\[ \text{get} :: [\alpha] \rightarrow [\alpha], \]

what should be \( V^C \) and

\[ \text{compl} :: [\alpha] \rightarrow V^C \]

To make

\[ \lambda s \rightarrow (\text{get} \ s, \text{compl} \ s) \]

injective, need to record information discarded by \text{get}.

Candidates:

1. length of the source list
2. discarded list elements
The Constant-Complement Approach

For a very simple setting,

\[
\text{get} :: [\alpha] \rightarrow [\alpha],
\]

what should be \( V^C \) and

\[
\text{compl} :: [\alpha] \rightarrow V^C \quad ???
\]

To make

\[
\lambda s \rightarrow (\text{get } s, \text{compl } s)
\]

injective, need to record information discarded by \( \text{get} \).

Candidates:

1. length of the source list
2. discarded list elements

For the moment, be maximally conservative.
The Complement Function

\[
\text{compl} :: [\alpha] \rightarrow (\text{Int}, [\alpha])
\]

\[
\text{compl} \ s = \begin{aligned}
\text{let} \ n &= (\text{length} \ s) - 1 \\
t &= [0..n] \\
g &= \text{zip} \ t \ s \\
g' &= \text{filter} \ (\lambda(i, \_ \rightarrow \text{notElem} \ i \ (\text{get} \ t)) \ g \\
\text{in} \ (n + 1, \text{map} \ \text{snd} \ g')
\end{aligned}
\]

For example:

\[
\text{get} = \text{tail} \Rightarrow \text{compl} \ "abcde" = (5, ['a'])
\]

\[
\text{get} = \text{take} 3 \Rightarrow \text{compl} \ "abcde" = (5, ['d', 'e'])
\]

\[
\text{get} = \text{reverse} \Rightarrow \text{compl} \ "abcde" = (5, [])
\]
The Complement Function

\[
\text{compl} :: [\alpha] \rightarrow (\text{Int}, [\alpha])
\]
\[
\text{compl} \ s = \text{let } n = (\text{length } s) - 1
\]
\[
t = [0..n]
\]
\[
g = \text{zip } t \ s
\]
\[
g' = \text{filter } (\lambda(i, _) \rightarrow \text{notElem } i \ (\text{get } t)) \ g
\]
\[
\text{in } (n + 1, \text{map snd } g')
\]

For example:

\[
\text{get } = \text{tail} \quad \leadsto \quad \text{compl } "abcde" = (5, ['a'])
\]
The Complement Function

\[ \text{compl} :: [\alpha] \rightarrow (\text{Int}, [\alpha]) \]

\[
\text{compl } s = \text{let } n = (\text{length } s) - 1 \\
\quad t = [0..n] \\
\quad g = \text{zip } t \ s \\
\quad g' = \text{filter } (\lambda(i, _) \rightarrow \text{notElem } i \ (\text{get } t)) \ g \\
\quad \text{in } (n + 1, \text{map } \text{snd} \ g')
\]

For example:

\[
\text{get } = \text{tail} \quad \rightsquigarrow \quad \text{compl} \ "abcde" = (5, ['a'])
\]

\[
\text{get } = \text{take } 3 \quad \rightsquigarrow \quad \text{compl} \ "abcde" = (5, ['d', 'e'])
\]
The Complement Function

\[
\text{compl} :: [\alpha] \rightarrow (\text{Int}, [\alpha])
\]

\[
\text{compl} \ s = \text{let} \ n = (\text{length} \ s) - 1
\]

\[
t = [0..n]
\]

\[
g = \text{zip} \ t \ s
\]

\[
g' = \text{filter} (\lambda(i,_) \rightarrow \text{notElem} \ i \ (\text{get} \ t)) \ g
\]

\[
\text{in} \ (n + 1, \text{map} \ \text{snd} \ g')
\]

For example:

\[
\text{get} = \text{tail} \quad \leadsto \quad \text{compl} \ "abcde" = (5, ['a'])
\]

\[
\text{get} = \text{take} 3 \quad \leadsto \quad \text{compl} \ "abcde" = (5, ['d', 'e'])
\]

\[
\text{get} = \text{reverse} \quad \leadsto \quad \text{compl} \ "abcde" = (5, [])
\]
An Inverse of $\lambda s \rightarrow (get\ s, compl\ s)$

\[
\text{inv} :: ([\alpha], (Int, [\alpha])) \rightarrow [\alpha]
\]

\[
\text{inv} ([], (0, _)) = []
\]

\[
\text{inv} (v', (n + 1, as)) =
\]

\[
\text{let } t = [0..n]
\]

\[
h = \text{assoc} (get\ t) v'
\]

\[
g' = \text{zip} (\text{filter} (\lambda i \rightarrow \text{notElem}\ i (\text{get}\ t)) t)\ as
\]

\[
h' = h ++ g'
\]

\[
\text{in } \text{map} (\lambda i \rightarrow \text{fromJust} (\text{lookup}\ i\ h'))\ t
\]
An Inverse of $\lambda s \to \langle \text{get } s, \text{compl } s \rangle$

\[
\text{inv} :: ([\alpha], (\text{Int, } [\alpha])) \to [\alpha]
\]
\[
\text{inv} ([], (0, _)) = []
\]
\[
\text{inv} (v', (n + 1, as)) =
\]
\[
\text{let } t = [0..n]
\]
\[
\phantom{=} h = \text{assoc}^\dagger (\text{get } t) v'
\]
\[
\phantom{=} g' = \text{zip} (\text{filter} (\lambda i \to \text{notElem } i (\text{get } t)) t) \text{ as }
\]
\[
\phantom{=} h' = h ++ g'
\]
\[
\text{in } \text{map} (\lambda i \to \text{fromJust } (\text{lookup } i h')) t
\]

\[^\dagger\] Can be thought of as \text{zip} for the moment.
An Inverse of $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

\[
\text{inv} :: ([\alpha], (\text{Int}, [\alpha])) \rightarrow [\alpha] \\
\text{inv} ([], (0, \_)) = [] \\
\text{inv} (v', (n + 1, as)) = \\
\quad \text{let } t = [0..n] \\
\quad \quad h = \text{assoc}^\dagger (\text{get } t) v' \\
\quad \quad g' = \text{zip} (\text{filter} (\lambda i \rightarrow \text{notElem } i (\text{get } t)) t) \text{ as} \\
\quad \quad h' = h ++ g' \\
\quad \text{in } map (\lambda i \rightarrow \text{fromJust} (\text{lookup } i h')) t
\]

For example:

\[
\text{get} = \text{tail} \quad \leadsto \quad \text{inv} ("bcde", (5, [\text{'a'}])) = "abcde"
\]

\dagger Can be thought of as \text{zip} for the moment.
An Inverse of $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

\[
\text{inv} :: ([\alpha], (\text{Int}, [\alpha])) \rightarrow [\alpha]\\
\text{inv} ([], (0, _)) = []\\
\text{inv} (v', (n + 1, as)) =\\
\quad \text{let } t = [0..n]\\
\quad \quad h = \text{assoc}^\dagger (\text{get } t) v'\\
\quad \quad g' = \text{zip} (\text{filter} (\lambda i \rightarrow \text{notElem } i (\text{get } t)) t) \text{ as } h' = h ++ g'\\
\quad \text{in } \text{map} (\lambda i \rightarrow \text{fromJust} (\text{lookup } i h')) t
\]

For example:

\[
\text{get } = \text{tail} \quad \implies \quad \text{inv} ("bcde", (5, ['a'])) = "abcde" \\
\text{get } = \text{take} 3 \quad \implies \quad \text{inv} ("xyz", (5, ['d', 'e'])) = "xyzde"
\]

\(^\dagger\) Can be thought of as \text{zip} for the moment.
Correctness

To prove formally:

- \( \text{inv} (\text{get } s, \text{compl } s) = s \)
- if \( \text{inv} (v, c) \) defined, then \( \text{get} (\text{inv} (v, c)) = v \)
- if \( \text{inv} (v, c) \) defined, then \( \text{compl} (\text{inv} (v, c)) = c \)
Correctness

To prove formally:

- \( \text{inv} \left( \text{get} \ s, \text{compl} \ s \right) = s \)
- if \( \text{inv} \ (v, c) \) defined, then \( \text{get} \ (\text{inv} \ (v, c)) = v \)
- if \( \text{inv} \ (v, c) \) defined, then \( \text{compl} \ (\text{inv} \ (v, c)) = c \)

Use a free theorem [Wadler, FPCA’89], namely that for every

\[ \text{get} :: [\alpha] \rightarrow [\alpha] \]

we have, for arbitrary \( f \) and \( l \),

\[ \text{map} \ f \ (\text{get} \ l) = \text{get} \ (\text{map} \ f \ l). \]
Correctness

To prove formally:

- \( \text{inv} (\text{get } s, \text{compl } s) = s \)
- if \( \text{inv} (v, c) \) defined, then \( \text{get} (\text{inv} (v, c)) = v \)
- if \( \text{inv} (v, c) \) defined, then \( \text{compl} (\text{inv} (v, c)) = c \)

Use a free theorem [Wadler, FPCA'89], namely that for every

\[
\text{get} :: [\alpha] \rightarrow [\alpha]
\]

we have, for arbitrary \( f \) and \( l \),

\[
\text{map } f \ (\text{get } l) = \text{get} \ (\text{map } f \ l).
\]

Given an arbitrary list \( s \) of length \( n + 1 \), set \( l = [0..n] \), \( f = (s !!) \), leading to:

\[
\text{map} \ (s !!) \ (\text{get } [0..n]) = \text{get} \ (\text{map} \ (s !!) \ [0..n])
\]
Correctness

To prove formally:

- $\text{inv} \ (\text{get} \ s, \ \text{compl} \ s) = s$
- if $\text{inv} \ (v, c)$ defined, then $\text{get} \ (\text{inv} \ (v, c)) = v$
- if $\text{inv} \ (v, c)$ defined, then $\text{compl} \ (\text{inv} \ (v, c)) = c$

Use a free theorem [Wadler, FPCA’89], namely that for every

$\text{get} :: [\alpha] \rightarrow [\alpha]$

we have, for arbitrary $f$ and $l$,

$\text{map} \ f \ (\text{get} \ l) = \text{get} \ (\text{map} \ f \ l)$.

Given an arbitrary list $s$ of length $n + 1$, set $l = [0..n]$, $f = (s !!)$, leading to:

$\text{map} \ (s !!) \ (\text{get} \ [0..n]) = \text{get} \ (\text{map} \ (s !!) \ [0..n])$

$= \text{get} \ s$
Correctness

To prove formally:

- $\text{inv } (\text{get } s, \text{compl } s) = s$
- If $\text{inv } (v, c)$ defined, then $\text{get } (\text{inv } (v, c)) = v$
- If $\text{inv } (v, c)$ defined, then $\text{compl } (\text{inv } (v, c)) = c$

Use a free theorem [Wadler, FPCA’89], namely that for every

$$\text{get} :: [\alpha] \rightarrow [\alpha]$$

we have, for arbitrary $f$ and $l$,

$$\text{map } f \ (\text{get } l) = \text{get } (\text{map } f \ l) .$$

Given an arbitrary list $s$ of length $n + 1$,

$$\text{map } (s!!) \ (\text{get } [0..n]) = \text{get } s$$
Correctness

To prove formally:

- $\text{inv} (\text{get } s, \text{compl } s) = s$
- If $\text{inv} (v, c)$ defined, then $\text{get} (\text{inv } (v, c)) = v$
- If $\text{inv} (v, c)$ defined, then $\text{compl} (\text{inv } (v, c)) = c$

Use a free theorem [Wadler, FPCA’89], namely that for every

$$\text{get} :: [\alpha] \rightarrow [\alpha]$$

we have, for arbitrary $f$ and $l$,

$$\text{map } f (\text{get } l) = \text{get} (\text{map } f l).$$

Given an arbitrary list $s$ of length $n + 1$,

$$\text{get } s = \text{map } (s!!) (\text{get } [0..n])$$
Altogether, So Far:

\[
\text{compl} :: [\alpha] \to (\text{Int}, [\alpha])
\]

\[
\text{compl} \ s = \text{let} \ n = (\text{length} \ s) - 1 \\
\quad t = [0..n] \\
\quad g = \text{zip} \ t \ s \\
\quad g' = \text{filter} (\lambda(i, _) \to \text{notElem} \ i \ (\text{get} \ t)) \ g \\
\text{in} \ (n + 1, \text{map} \ \text{snd} \ g')
\]

\[
\text{inv} :: ([\alpha], (\text{Int}, [\alpha])) \to [\alpha]
\]

\[
\text{inv} ([\ ], (0, _)) = [\ ] \\
\text{inv} (v', (n + 1, as)) = \\
\quad \text{let} \ t = [0..n] \\
\quad h = \text{assoc} \ (\text{get} \ t) \ v' \\
\quad g' = \text{zip} \ (\text{filter} (\lambda i \to \text{notElem} \ i \ (\text{get} \ t)) \ t) \ as \\
\quad h' = h ++ g' \\
\text{in} \ \text{map} \ (\lambda i \to \text{fromJust} \ (\text{lookup} \ i \ h')) \ t
\]
“Fusion”

Inlining `compl` and `inv` into `put`, plus some clever rewriting:

\[
\begin{align*}
\text{put} \; [] \; [] &= [] \\
\text{put} \; s \; v' &= \text{let} \; n = (\text{length} \; s) - 1 \\
&\quad \, t = [0..n] \\
&\quad \, g = \text{zip} \; t \; s \\
&\quad \, g' = \text{filter} \; (\lambda (i, _) \rightarrow \text{notElem} \; i \; (\text{get} \; t)) \; g \\
&\quad \, h = \text{assoc} \; (\text{get} \; t) \; v' \\
&\quad \, h' = h ++ g' \\
\text{in} \quad \text{seq} \; h \; (\text{map} \; (\lambda i \rightarrow \text{fromJust} \; (\text{lookup} \; i \; h')) \; t)
\end{align*}
\]
“Fusion”

Inlining `compl` and `inv` into `put`, plus some clever rewriting:

```haskell
put [] [] = []
put s v' = let n = (length s) - 1
            t = [0..n]
            g = zip t s
            g' = filter (λ(i, _) → notElem i (get t)) g
            h = assoc (get t) v'
            h' = h ++ g'
            in seq h (map (λi → fromJust (lookup i h'))) t)

assoc [] [] = []
assoc (i : is) (b : bs) = let m = assoc is bs
                          in case lookup i m of
                              Nothing        → (i, b) : m
                              Just c | b == c → m
```

Actual code only slightly more elaborate!
“Fusion”

Inlining `compl` and `inv` into `put`, plus some clever rewriting:

```
bff get [] [] = []
bff get s v' = let n = (length s) − 1
                    t = [0..n]
                    g = zip t s
                    g' = filter (λ(i, _) → notElem i (get t)) g
                    h = assoc (get t) v'
                    h' = h ++ g'
                    in seq h (map (λi → fromJust (lookup i h')) t)
```

```
assoc [] [] = []
assoc (i : is) (b : bs) = let m = assoc is bs
                          in case lookup i m of
                              Nothing → (i, b) : m
                              Just c | b == c → m
```

Actual code only slightly more elaborate!
“Fusion”

Inlining \texttt{compl} and \texttt{inv} into \texttt{put}, plus some clever rewriting:

\begin{verbatim}
bff get [] [] = []
bff get s v' = let n = (length s) - 1
    t = [0..n]
    g = zip t s
    g' = filter (\lambda (i,_) -> notElem i (get t)) g
    h = assoc (get t) v'
    h' = h ++ g'
    in seq h (map (\lambda i -> fromJust (lookup i h')) t)

assoc [] [] = []
assoc (i : is) (b : bs) = let m = assoc is bs
    in case lookup i m of
        Nothing -> (i, b) : m
        Just c | b == c -> m
\end{verbatim}

Actual code only slightly more elaborate!
Overview of the Bidirectionalization Method

\[ t \]

\[ s \]

\[ 0 \rightarrow 'b' \]
\[ 1 \rightarrow 'a' \]
\[ 2 \rightarrow 'c' \]
\[ 3 \rightarrow 'a' \]

\[ [1,2,3] \]

\[ \text{get } t \]

\[ \text{tail } \circ \text{ flatten} \]

\[ 0 \rightarrow 'b' \]
\[ 1 \rightarrow 'a' \]
\[ 2 \rightarrow 'c' \]
\[ 3 \rightarrow 'a' \]

\[ 'b' 'x' 'c' 'a' \]

\[ 'b' 'a' 'c' 'a' \]

\[ 'b' 'x' 'c' 'a' \]

\[ 'xca' \]

\[ \text{tail } \circ \text{ flatten} \]

\[ v' \]

\[ g \]

\[ h' \]

\[ h \]

\[ h' \]

\[ g \]

\[ v' \]
Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.

For example, \texttt{bff tail "abcde" "xyz"}...

Analysis as to Why:

- Our approach to making $\lambda$s $\rightarrow$ (\texttt{get}, \texttt{compl}) injective was to record, via \texttt{compl}, the following information:
  1. length of the source list
  2. discarded list elements

- Being maximally conservative this way often does not "collapse enough".

- For example: \texttt{get} $\rightarrow$ \texttt{tail} $\Rightarrow$ \texttt{put} "abcde" "xyz" fails precisely because \texttt{compl} "abcde" = (5, ['a'])
Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.
- For example, `bff tail “abcde” “xyz”` ...
Extending the Technique

Major Problem:
- Shape-affecting updates lead to failure.
- For example, \texttt{bff tail “abcde” “xyz”} ...

Analysis as to Why:
- Our approach to making \( \lambda s \rightarrow (\text{get } s, \text{compl } s) \) injective was to record, via \text{compl}, the following information:
  1. length of the source list
  2. discarded list elements
Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.
- For example, \texttt{bff tail "abcde" "xyz" ...}

Analysis as to Why:

- Our approach to making \( \lambda s \rightarrow (\text{get } s, \text{compl } s) \) injective was to record, via \texttt{compl}, the following information:
  1. length of the source list
  2. discarded list elements
- Being maximally conservative this way often does not “collapse enough”.
Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.
- For example, \texttt{bff tail “abcde” “xyz”} ...

Analysis as to Why:

- Our approach to making \( \lambda s \rightarrow (\textit{get} s, \textit{compl} s) \) injective was to record, via \textit{compl}, the following information:
  1. length of the source list
  2. discarded list elements

- Being maximally conservative this way often does not “collapse enough”.

- For example:
  \[
  \texttt{get = tail} \implies \texttt{put “abcde” “xyz”} \text{ fails precisely because } \texttt{compl “abcde” = (5, ’a’) }
  \]
Assuming Shape-Injectivity

So assume there is a function

\[ \text{shapeInv} :: \text{Int} \rightarrow \text{Int} \]

with, for every source list \( s \),

\[ \text{length } s = \text{shapeInv} \left( \text{length} \left( \text{get} \ s \right) \right) \]
Assuming Shape-Injectivity

So assume there is a function

\[ \text{shapeInv} :: \text{Int} \rightarrow \text{Int} \]

with, for every source list \( s \),

\[ \text{length } s = \text{shapeInv} \left( \text{length } (\text{get } s) \right) \]

Then:

\[ \text{compl} :: [\alpha] \rightarrow (\text{Int}, [\alpha]) \]

\[ \text{compl } s = \text{let } n = (\text{length } s) - 1 \]
\[ t = [0..n] \]
\[ g = \text{zip } t \ s \]
\[ g' = \text{filter} \left( \lambda (i, \_ ) \rightarrow \text{notElem } i \ (\text{get } t) \right) g \]
\[ \text{in } (n + 1, \text{map } \text{snd } g') \]
Assuming Shape-Injectivity

So assume there is a function

\[
\text{shapeInv} :: \text{Int} \to \text{Int}
\]

with, for every source list \(s\),

\[
\text{length } s = \text{shapeInv} (\text{length } (\text{get } s))
\]

Then:

\[
\text{compl} :: [\alpha] \to [\alpha]
\]

\[
\text{compl } s = \text{let } n = (\text{length } s) - 1
\]

\[
t = [0..n]
\]

\[
g = \text{zip } t s
\]

\[
g' = \text{filter } (\lambda(i, \_ ) \to \text{notElem } i (\text{get } t)) g
\]

\[
\text{in } \quad \text{map } \text{snd } g'
\]
Assuming Shape-Injectivity

\[
\begin{align*}
\text{inv} & : ([\alpha], (\text{Int}, [\alpha])) \to [\alpha] \\
\text{inv} ([], (0, _)) & = [] \\
\text{inv} (\nu', (n + 1, as)) & = \\
& \quad \text{let } t = [0..n] \\
& \quad \quad h = \text{assoc (get } t \text{) } \nu' \\
& \quad \quad g' = \text{zip (filter (}\lambda i \to \text{notElem } i \text{ (get } t\text{)) } t\text{) as} \\
& \quad \quad h' = h \uplus g' \\
& \quad \text{in } \text{map (}\lambda i \to \text{fromJust (lookup } i \text{ h'})\text{)} t
\end{align*}
\]
Assuming Shape-Injectivity

\[
\text{inv} :: ([\alpha], [\alpha]) \rightarrow [\alpha]
\]

\[
\text{inv}([], \_ ) = []
\]

\[
\text{inv}(\nu', \_ as) =
\]

\[
\begin{align*}
\text{let } n &= (\text{shapeInv } (\text{length } \nu')) - 1 \\
t &= [0..n] \\
h &= \text{assoc } (\text{get } t) \nu' \\
g' &= \text{zip } (\text{filter } (\lambda i \rightarrow \text{notElem } i (\text{get } t)) t) \text{ as} \\
h' &= h ++ g' \\
\text{in } \text{map } (\lambda i \rightarrow \text{fromJust } (\text{lookup } i h')) t
\end{align*}
\]
Assuming Shape-Injectivity

\[ \text{inv} :: ([\alpha], [\alpha]) \rightarrow [\alpha] \]

\[ \text{inv} ([], -) = [] \]

\[ \text{inv} (v', as) = \]

\[ \text{let } n = (\text{shapeInv (length } v')) - 1 \]

\[ t = [0..n] \]

\[ h = \text{assoc (get } t) v' \]

\[ g' = \text{zip (filter (}\lambda i \rightarrow \text{notElem } i \text{(get } t)) t) \text{ as} \]

\[ h' = h ++ g' \]

\[ \text{in } \text{map (}\lambda i \rightarrow \text{fromJust (lookup } i h')) t \]

But how to obtain \text{shapeInv} ???
Assuming Shape-Injectivity

\[ \text{inv} : ([\alpha], [\alpha]) \rightarrow [\alpha] \]
\[ \text{inv} ([], _) = [] \]
\[ \text{inv} (\nu', as) = \]
\[ \begin{align*}
\text{let } & n = (\text{shapeInv} \ (\text{length} \ \nu')) - 1 \\
n & = [0..n] \\
t & = [0..n] \\
h & = \text{assoc} \ (\text{get} \ t) \ \nu'
\end{align*} \]
\[ \begin{align*}
g' & = \text{zip} \ (\text{filter} \ (\lambda i \rightarrow \text{notElem} \ i \ (\text{get} \ t)) \ t) \ as \\
h' & = h ++ g'
\end{align*} \]
\[ \text{in } \text{map} \ (\lambda i \rightarrow \text{fromJust} \ (\text{lookup} \ i \ h')) \ t \]

But how to obtain \( \text{shapeInv} \) ???

Just for experimentation:

\[ \text{shapeInv} :: \text{Int} \rightarrow \text{Int} \]
\[ \text{shapeInv} \ l_v = \text{head} \ [n + 1 | n \leftarrow [0..], \ (\text{length} \ (\text{get} \ [0..n])) == l_v] \]
Some Tests

Works quite nicely in some cases:

\[
\text{get} = \text{tail} \quad \rightsquigarrow \quad \text{put} \quad \text{"abcde" "xyz" = "axyz"}, \text{ using}
\]
\[
\text{compl} \quad \text{"abcde" = ["a"]}
\]
Some Tests

Works quite nicely in some cases:

\[
\begin{align*}
\text{get} = \text{tail} \quad &\rightarrow \quad \text{put} \ "\text{abcde}" \ "\text{xyz}" = "\text{axyz}" , \text{ using} \\
&\text{ compl } \ "\text{abcde}" = [\'a\'] \\
\text{get} = \text{init} \quad &\rightarrow \quad \text{put} \ "\text{abcde}" \ "\text{xyz}" = "\text{xyze}" , \text{ using} \\
&\text{ compl } \ "\text{abcde}" = [\'e\']
\end{align*}
\]
Some Tests

Works quite nicely in some cases:

\[
get = \text{tail} \quad \leadsto \quad \text{put} \quad \text{“abcde” “xyz”} = \text{“axyz”}, \quad \text{using}
\]
\[
\text{compl} \quad \text{“abcde”} = \text{[’a’]}
\]

\[
get = \text{init} \quad \leadsto \quad \text{put} \quad \text{“abcde” “xyz”} = \text{“xyze”}, \quad \text{using}
\]
\[
\text{compl} \quad \text{“abcde”} = \text{[’e’]}
\]

But not so in others:

\[
get = \text{take 3} \quad \leadsto \quad \text{put} \quad \text{“abcde” “abc”} = \text{“abc”}
\]
Some Tests

Works quite nicely in some cases:

\[
\text{get} = \text{tail} \quad \leadsto \quad \text{put} \quad \text{"abcde" "xyz" = "axyz"}, \quad \text{using} \\
\quad \text{compl} \quad \text{"abcde" = ['a']}
\]

\[
\text{get} = \text{init} \quad \leadsto \quad \text{put} \quad \text{"abcde" "xyz" = "xyze"}, \quad \text{using} \\
\quad \text{compl} \quad \text{"abcde" = ['e']} \]

But not so in others:

\[
\text{get} = \text{take\ 3} \quad \leadsto \quad \text{put} \quad \text{"abcde" "abc" = "abc"}
\]

The problem: have forgotten to take the original source length into account.
Some Tests

Works quite nicely in some cases:

\[
\begin{align*}
\text{get} &= \text{tail} \quad \leadsto \quad \text{put} \; \text{"abcde"} \; \text{"xyz"} = \text{"axyz"}, \text{ using} \\
&\quad \text{compl} \; \text{"abcde"} = [\text{'a'}] \\
\text{get} &= \text{init} \quad \leadsto \quad \text{put} \; \text{"abcde"} \; \text{"xyz"} = \text{"xyze"}, \text{ using} \\
&\quad \text{compl} \; \text{"abcde"} = [\text{'e'}]
\end{align*}
\]

But not so in others:

\[
\begin{align*}
\text{get} &= \text{take} \; 3 \quad \leadsto \quad \text{put} \; \text{"abcde"} \; \text{"abc"} = \text{"abc"}
\end{align*}
\]

The problem: have forgotten to take the original source length into account.

Better:

\[
\begin{align*}
\text{shapeInv} &:: \text{Int} \to \text{Int} \to \text{Int} \\
\text{shapeInv} \; l_s \; l_v &= \text{head} \; [n + 1 \mid n \leftarrow (l_s - 1) : [0..], \\
&\quad (\text{length} \; (\text{get} \; [0..n])) == l_v]
\end{align*}
\]
Conclusion

[V., POPL’09]:
- very lightweight, easy access to bidirectionality
- full treatment of equality and ordering constraints
- proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Here:
- a constant-complement perspective on the method

Outlook:
- . . . could also be a way to inject/exploit “user knowledge”
- . . . combination with syntactic bidirectionalization ` a la [Matsuda et al., ICFP’07] is work in progress
- efficiency issues untackled so far, . . .
Conclusion

[V., POPL’09]:
▶ very lightweight, easy access to bidirectionality
▶ full treatment of equality and ordering constraints
▶ proofs, using free theorems and equational reasoning
▶ a datatype-generic account of the whole story

Here:
▶ a constant-complement perspective on the method
▶ . . . helps expanding its scope to updates that affect shape

Outlook:
▶ . . . could also be a way to inject/exploit “user knowledge”
▶ combination with syntactic bidirectionalization ` a la [Matsuda et al., ICFP’07] is work in progress
▶ efficiency issues untackled so far, . . .
Conclusion

[V., POPL’09]:
- very lightweight, easy access to bidirectionality
- full treatment of equality and ordering constraints
- proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Here:
- a constant-complement perspective on the method
- ... helps expanding its scope to updates that affect shape

Outlook:
- ... could also be a way to inject/exploit “user knowledge”
Conclusion

[V., POPL’09]:
▶ very lightweight, easy access to bidirectionality
▶ full treatment of equality and ordering constraints
▶ proofs, using free theorems and equational reasoning
▶ a datatype-generic account of the whole story

Here:
▶ a constant-complement perspective on the method
▶ . . . helps expanding its scope to updates that affect shape

Outlook:
▶ . . . could also be a way to inject/exploit “user knowledge”
▶ combination with syntactic bidirectionalization à la [Matsuda et al., ICFP’07] is work in progress
Conclusion

[V., POPL’09]:
▶ very lightweight, easy access to bidirectionality
▶ full treatment of equality and ordering constraints
▶ proofs, using free theorems and equational reasoning
▶ a datatype-generic account of the whole story

Here:
▶ a constant-complement perspective on the method
▶ . . . helps expanding its scope to updates that affect shape

Outlook:
▶ . . . could also be a way to inject/exploit “user knowledge”
▶ combination with syntactic bidirectionalization à la [Matsuda et al., ICFP’07] is work in progress
▶ efficiency issues untackled so far, . . .
References I

F. Bancilhon and N. Spyrratos.
Update semantics of relational views.

J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt.
Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem.

Bidirectionalization transformation based on automatic derivation of view complement functions.
References II

J. Voigtländer.  
Bidirectionalization for free!  

P. Wadler.  
Theorems for free!  