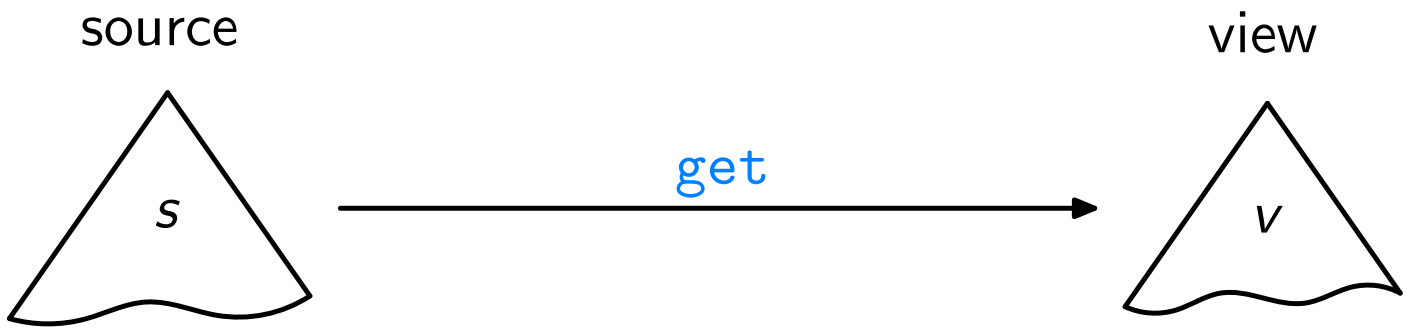
Semantic Bidirectionalization and the Constant-Complement Perspective

Janis Voigtländer

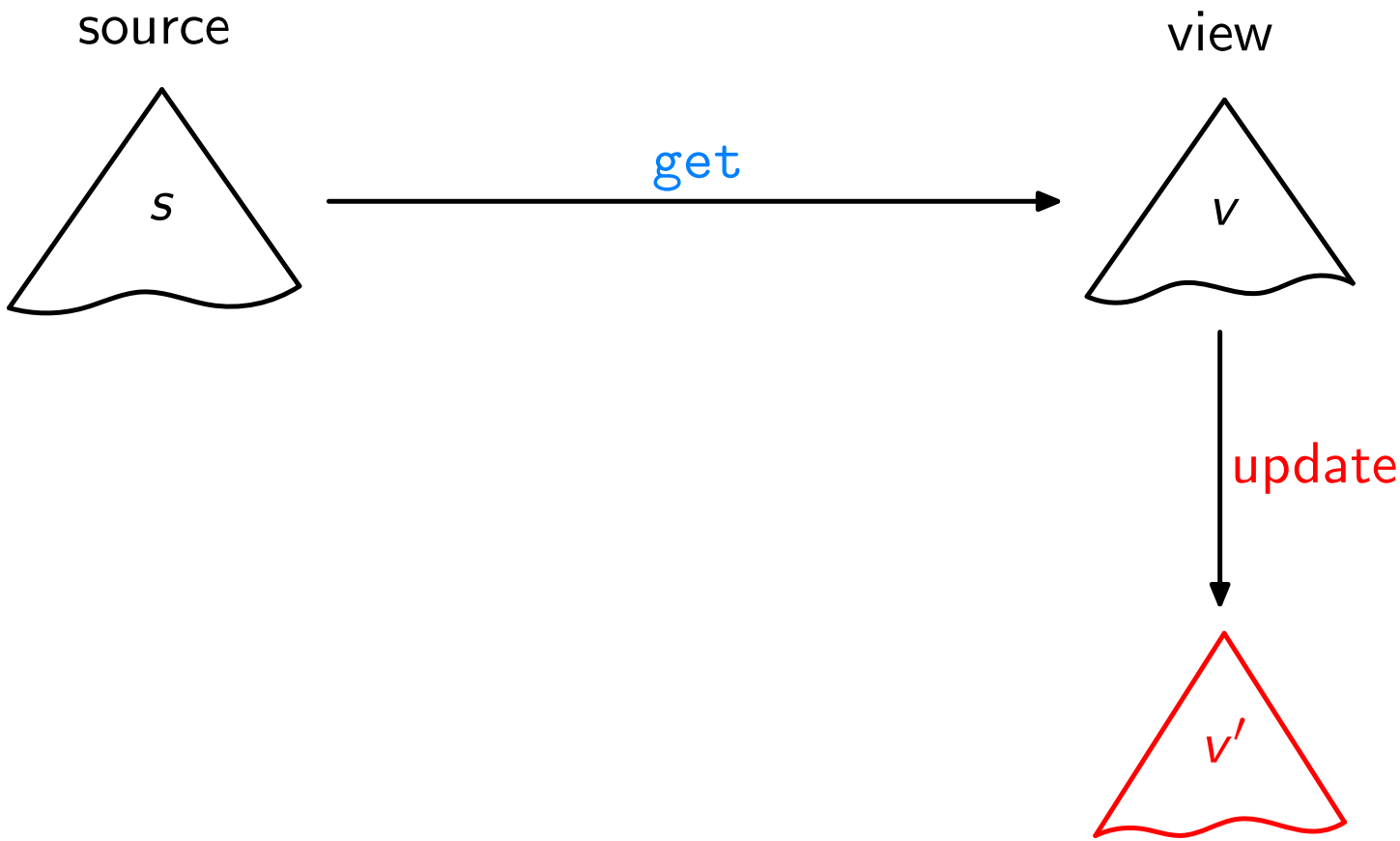
University of Bonn

BT-in-ABC'10

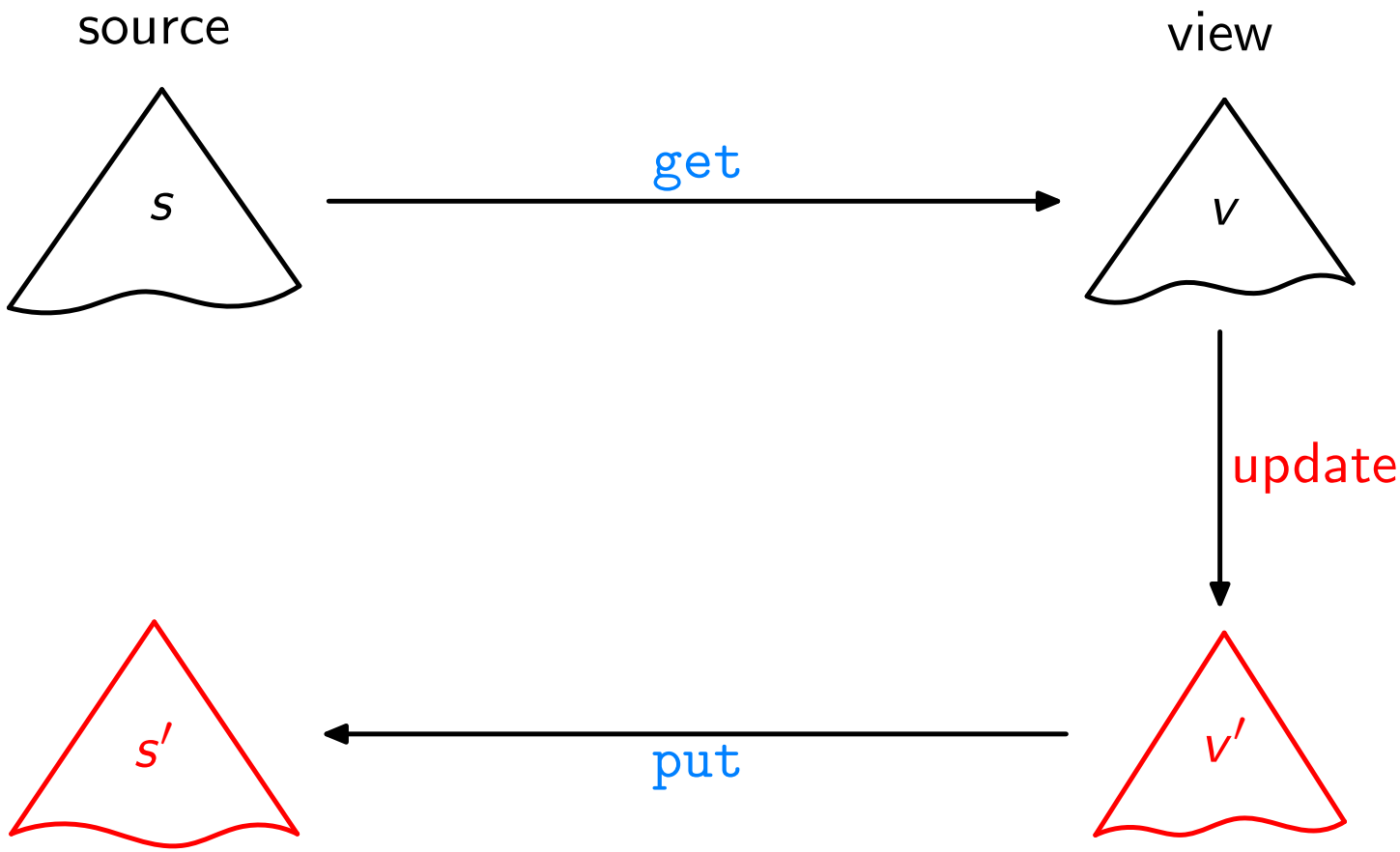
Bidirectional Transformation



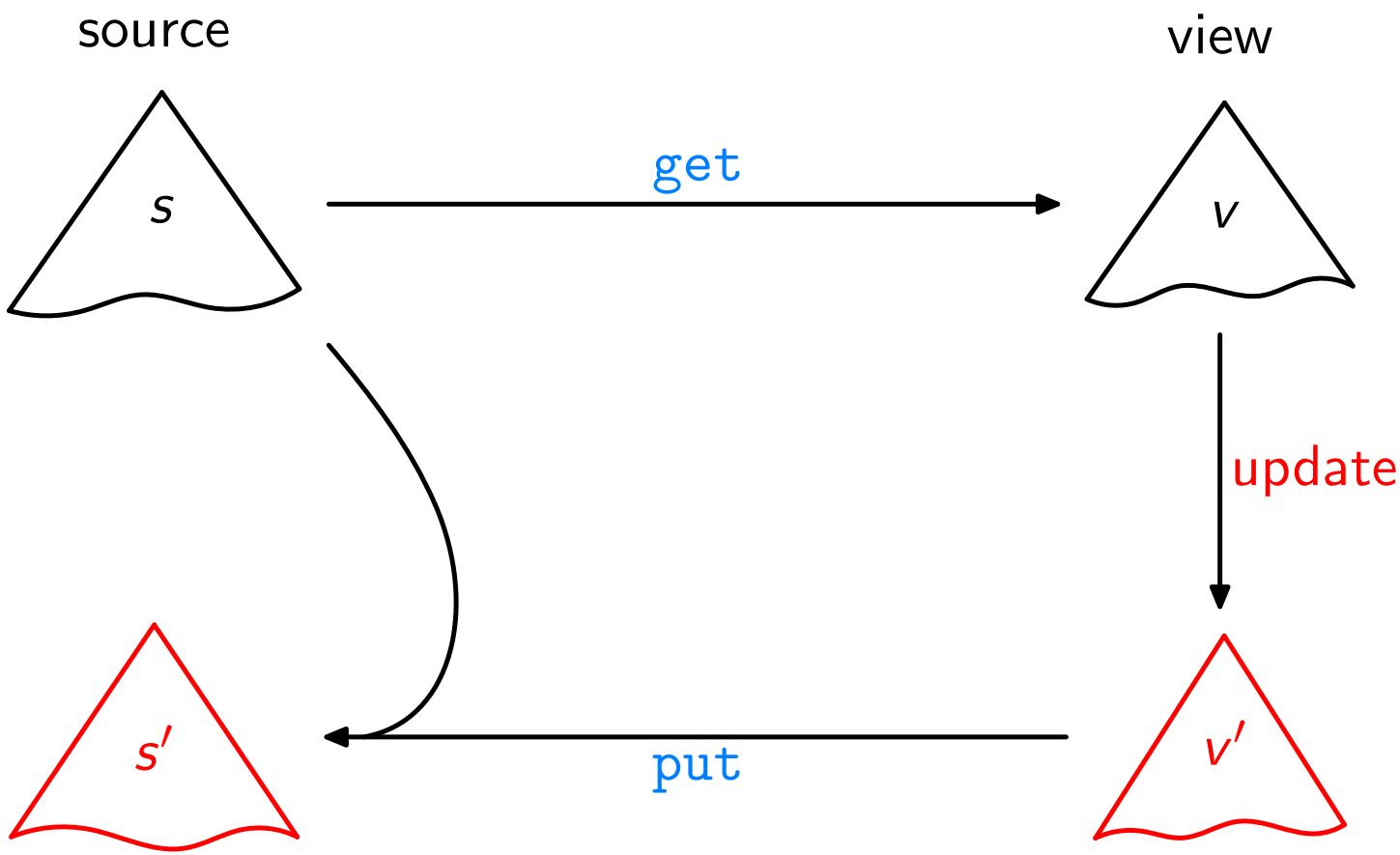
Bidirectional Transformation



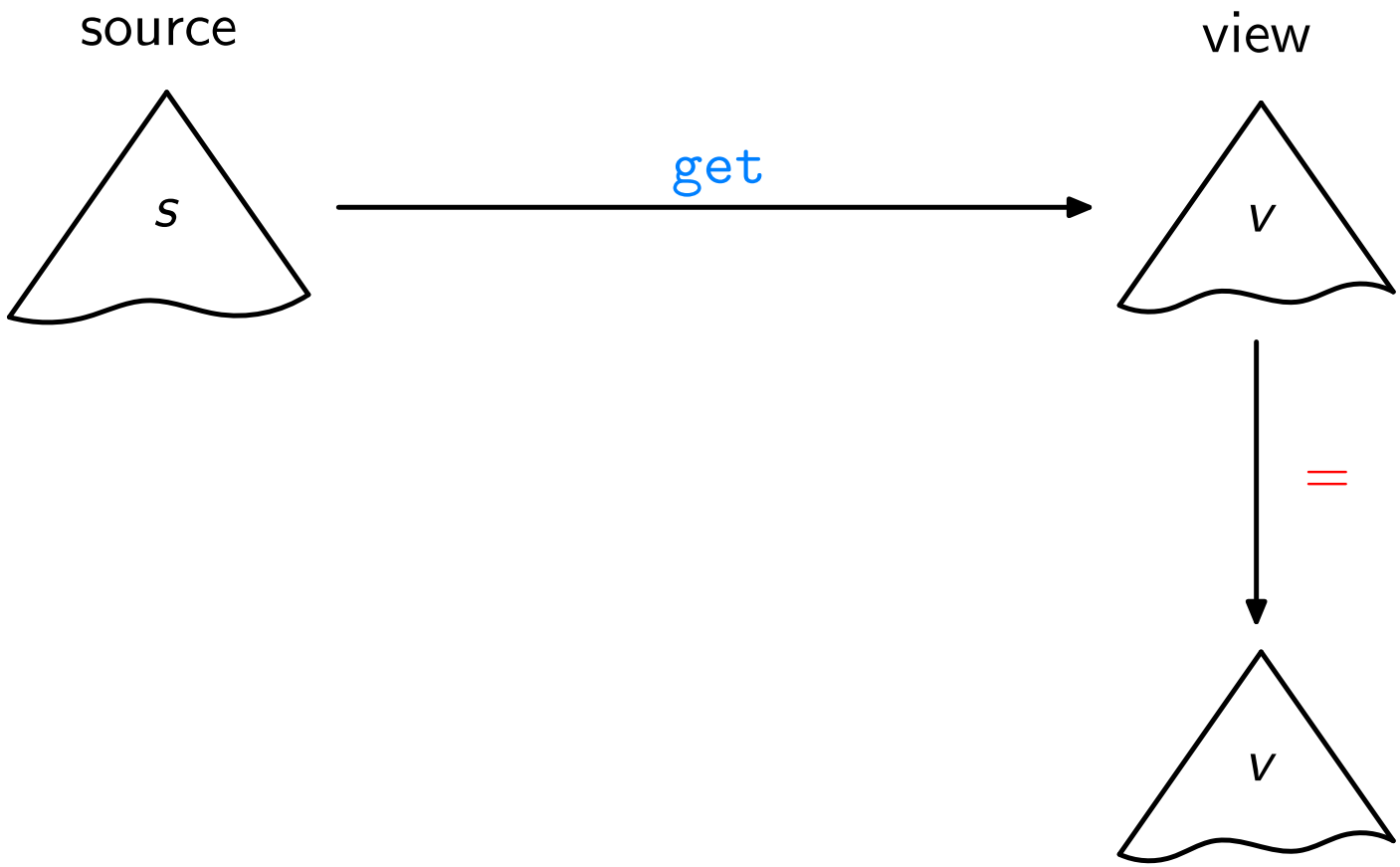
Bidirectional Transformation



Bidirectional Transformation

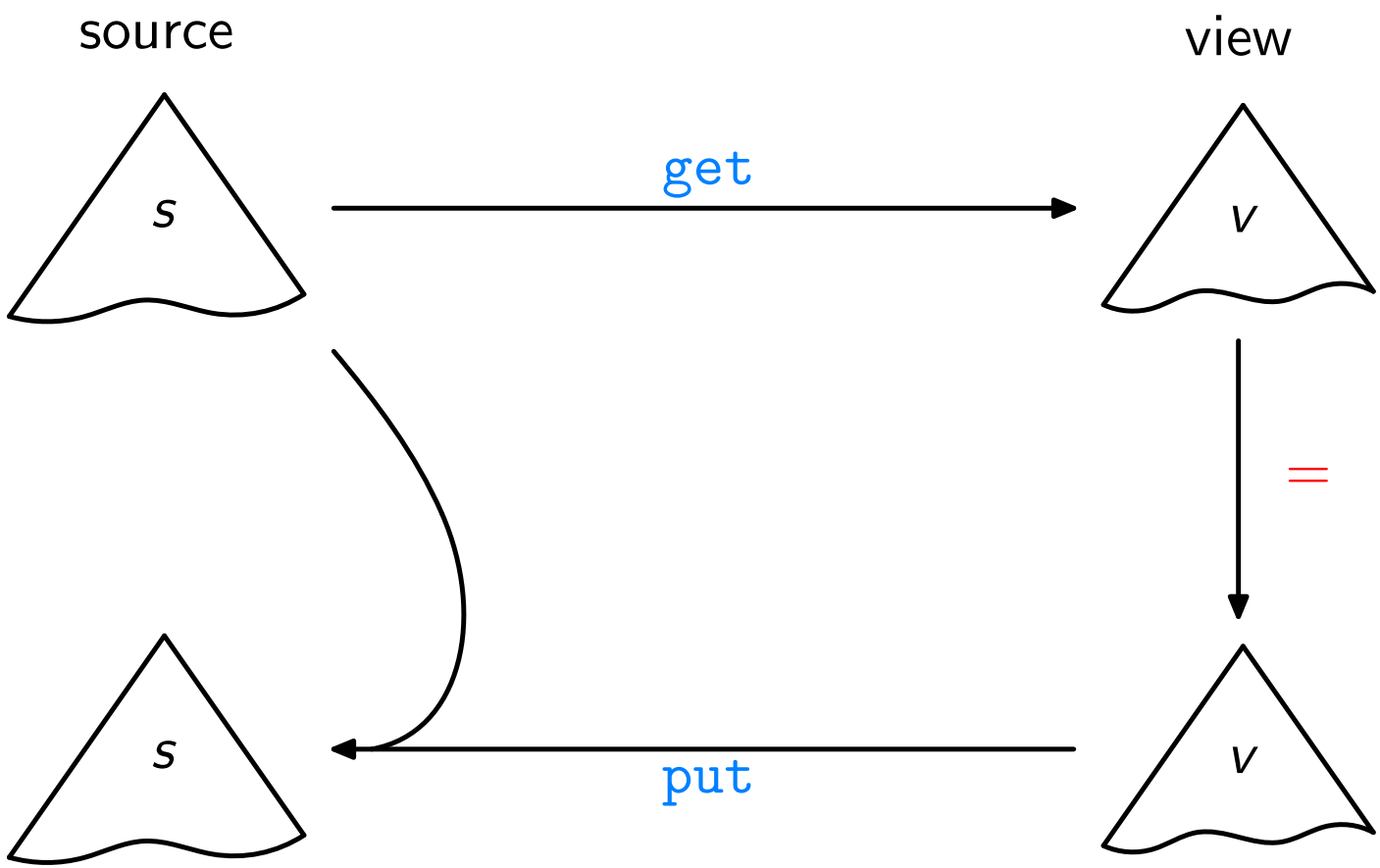


Bidirectional Transformation



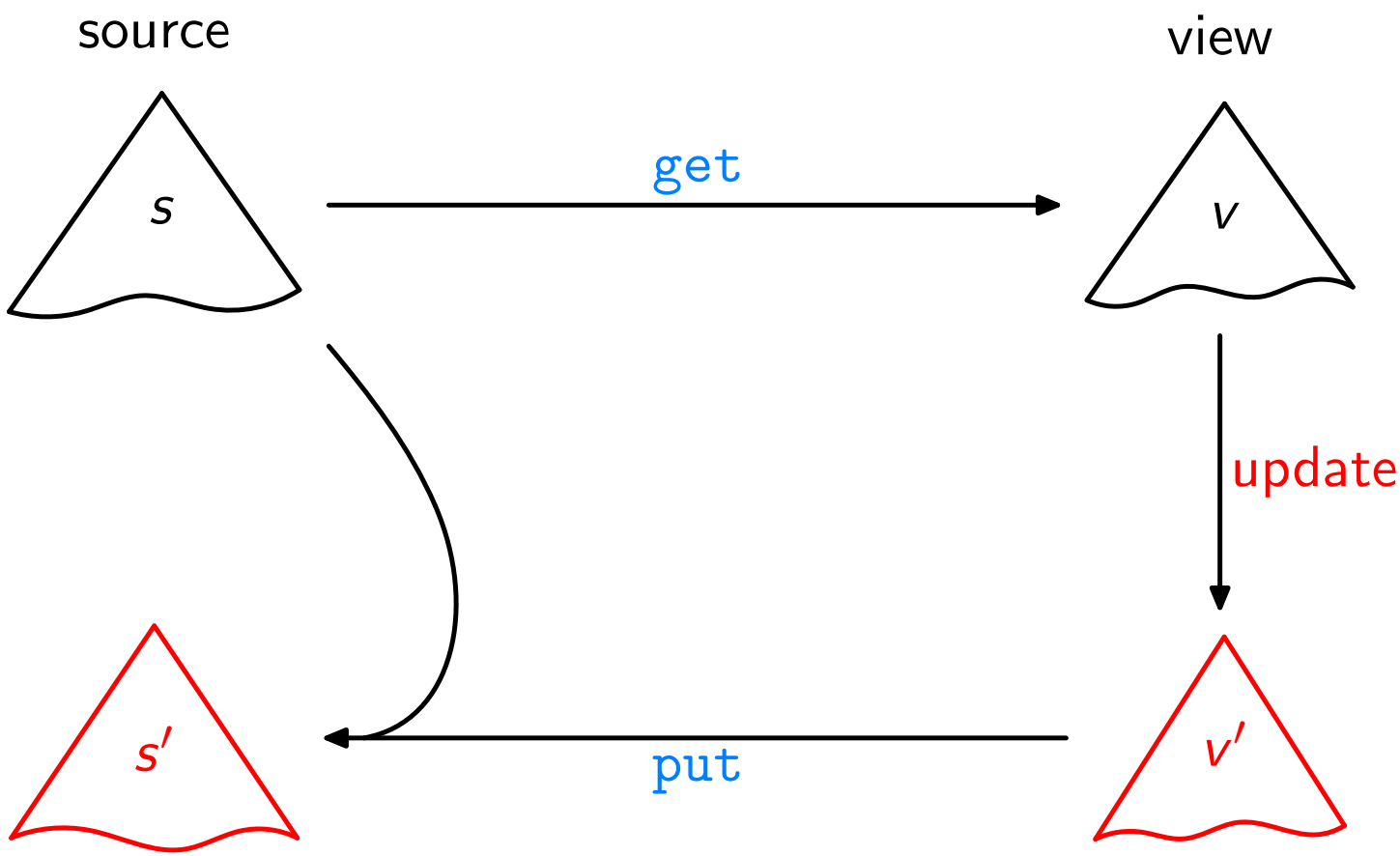
Acceptability / GetPut

Bidirectional Transformation



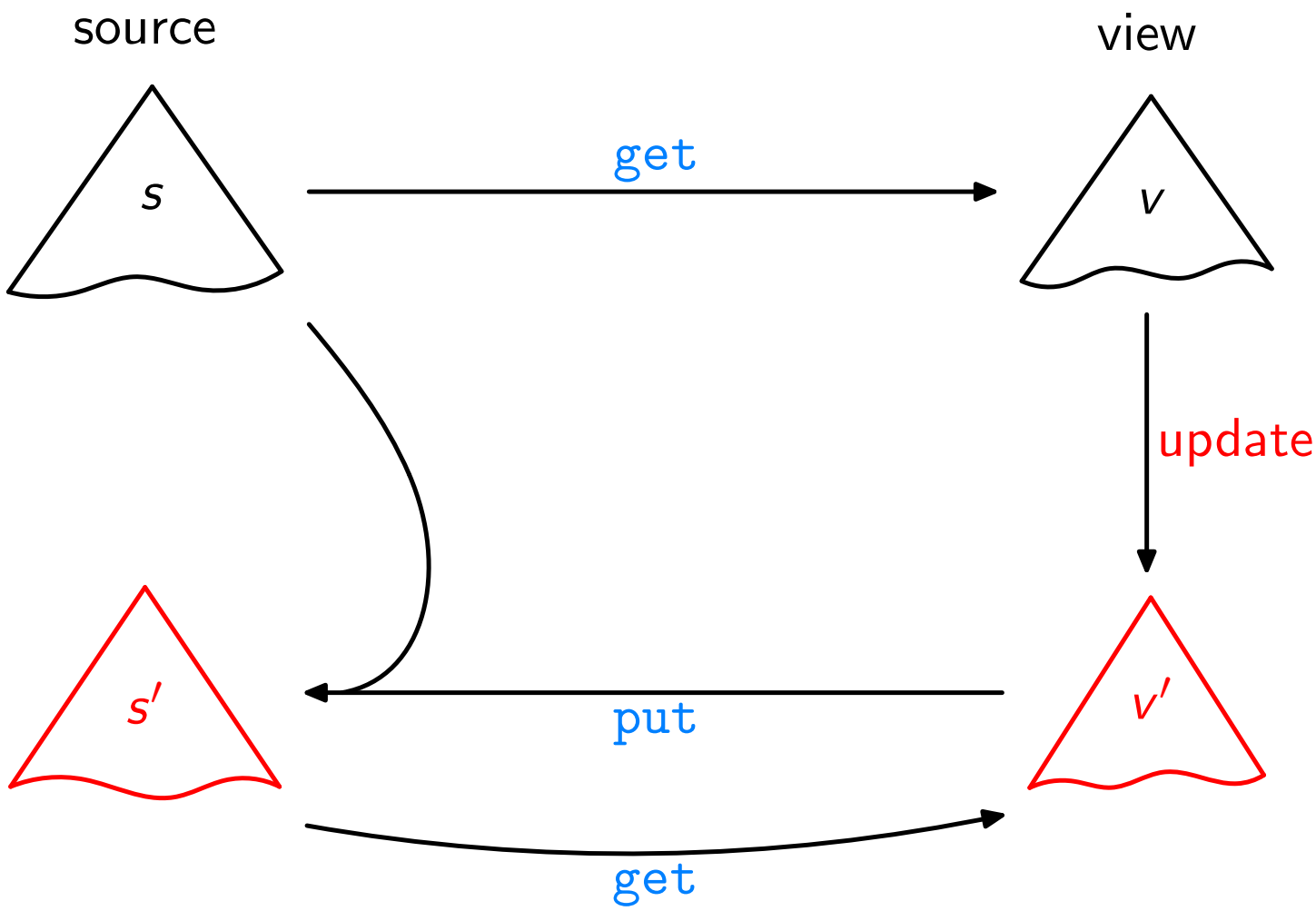
Acceptability / GetPut

Bidirectional Transformation



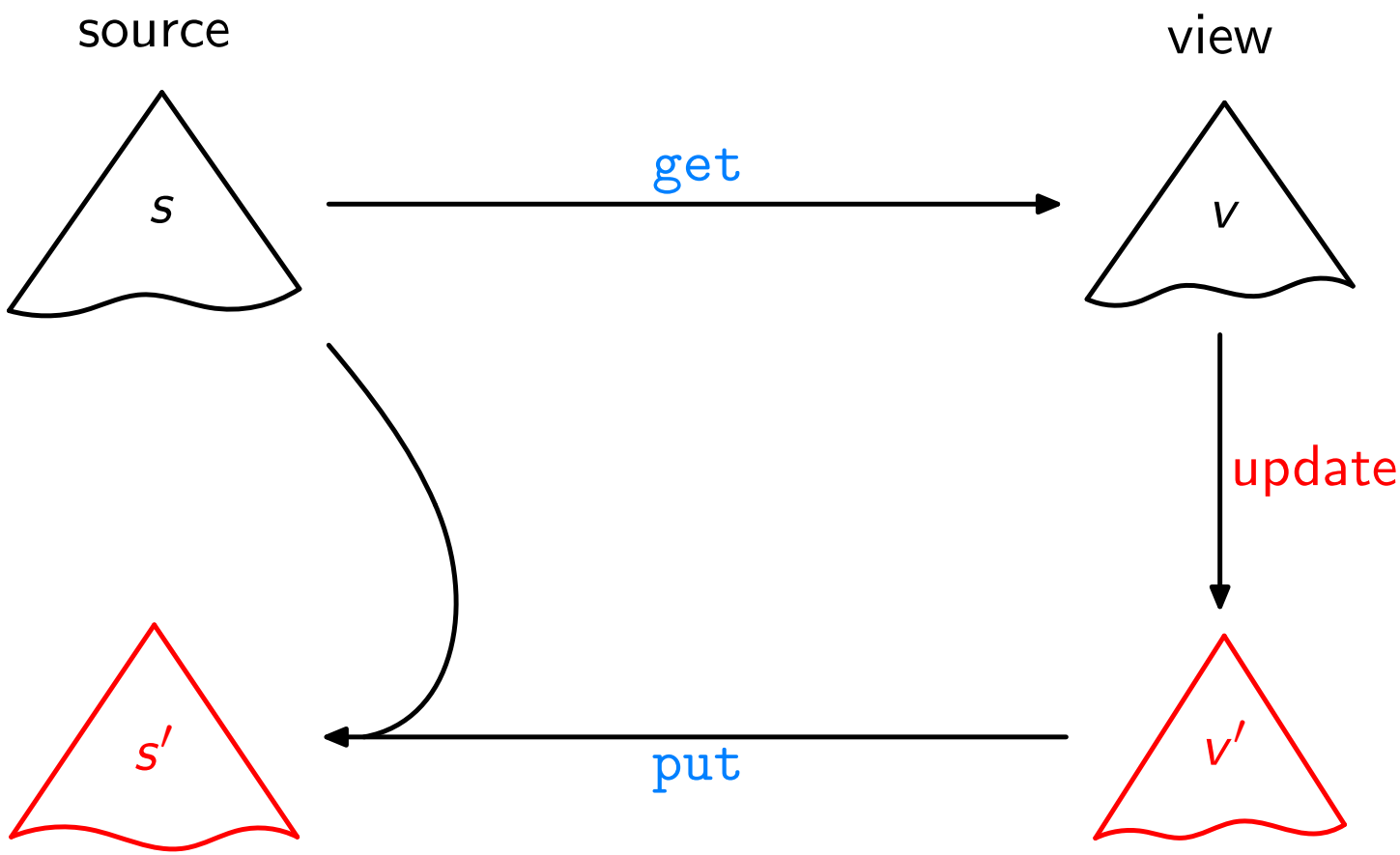
Consistency / PutGet

Bidirectional Transformation

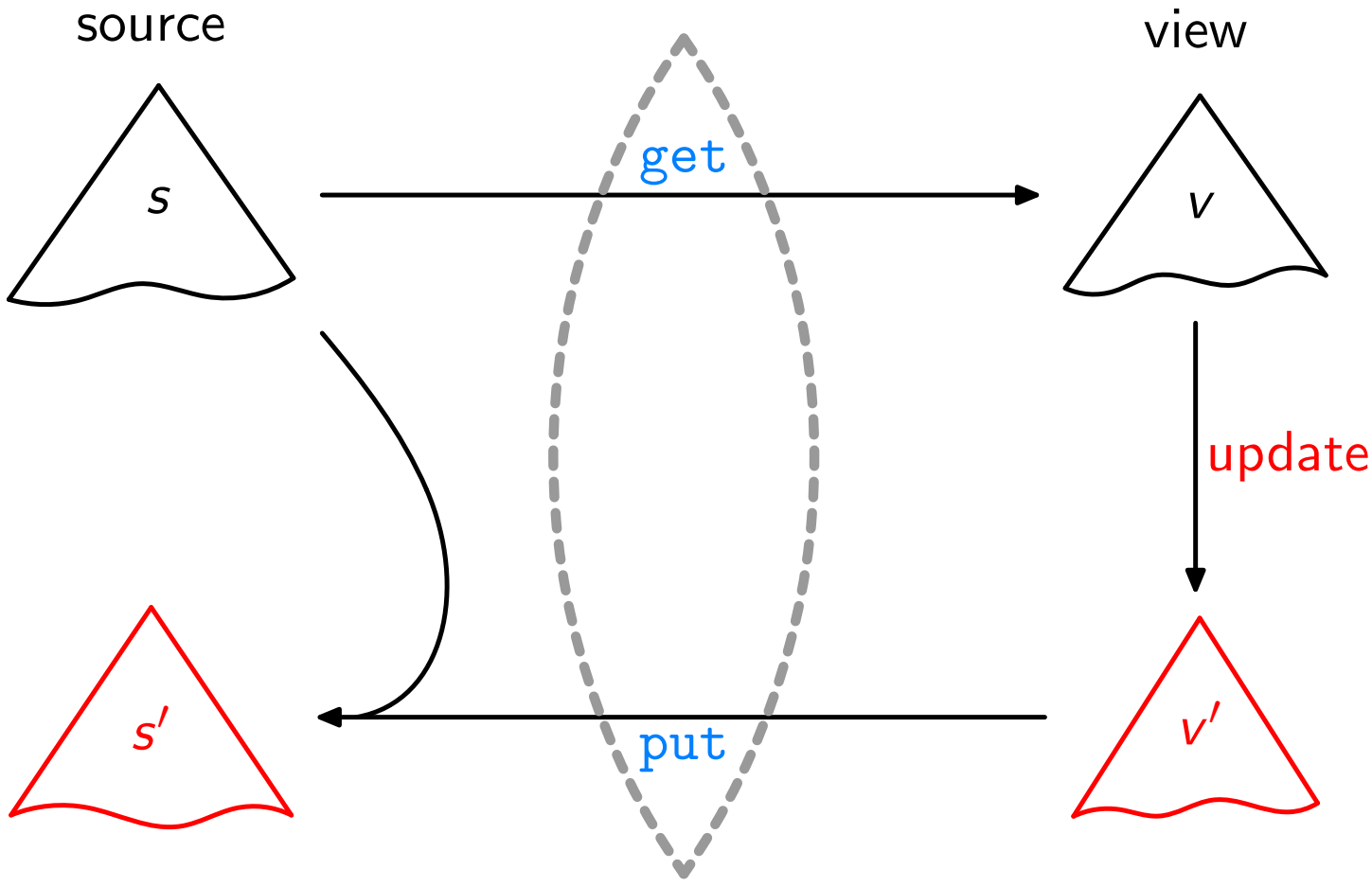


Consistency / PutGet

Bidirectional Transformation



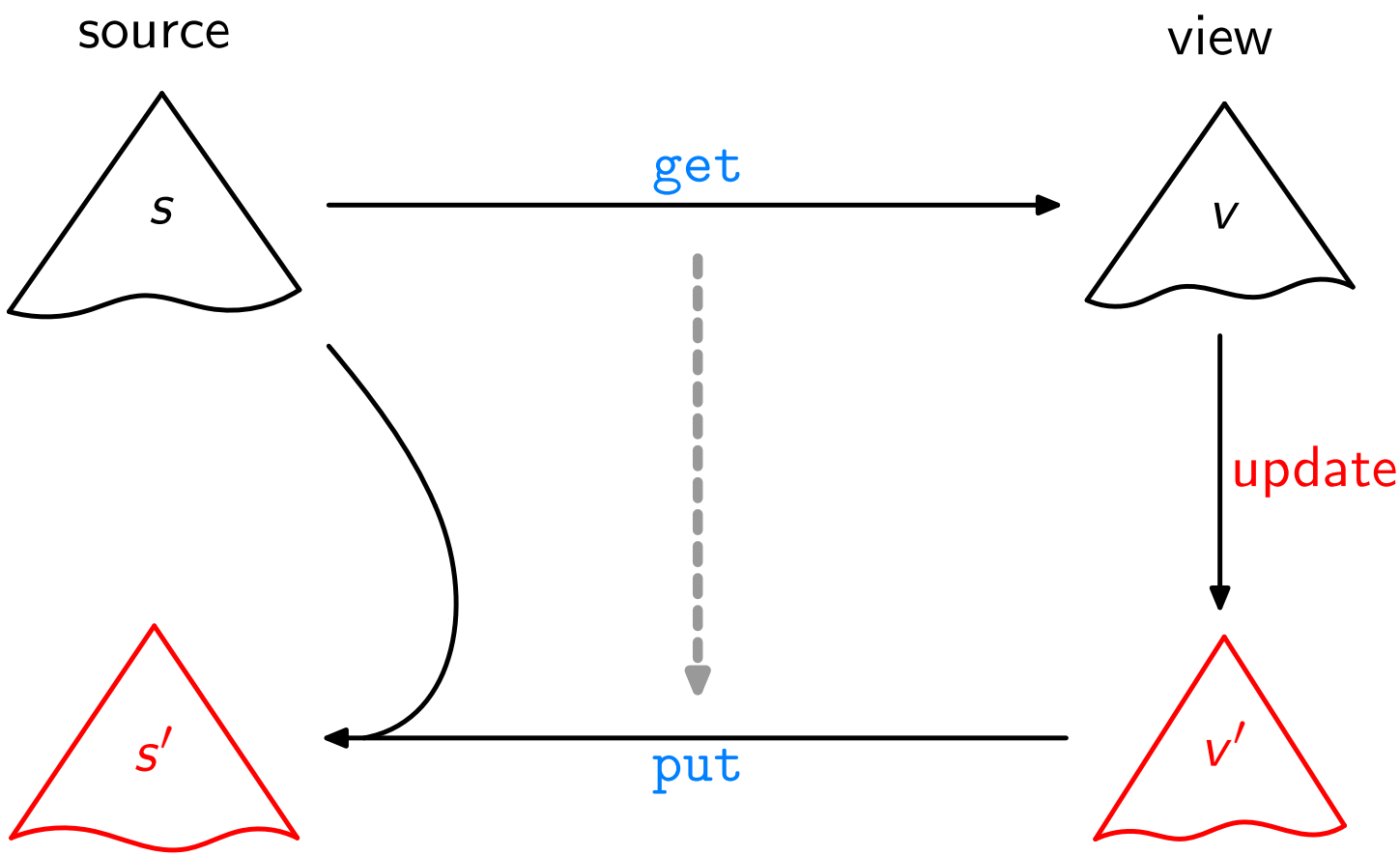
Bidirectional Transformation



Lenses, DSLs

[Foster et al., ACM TOPLAS'07, . . .]

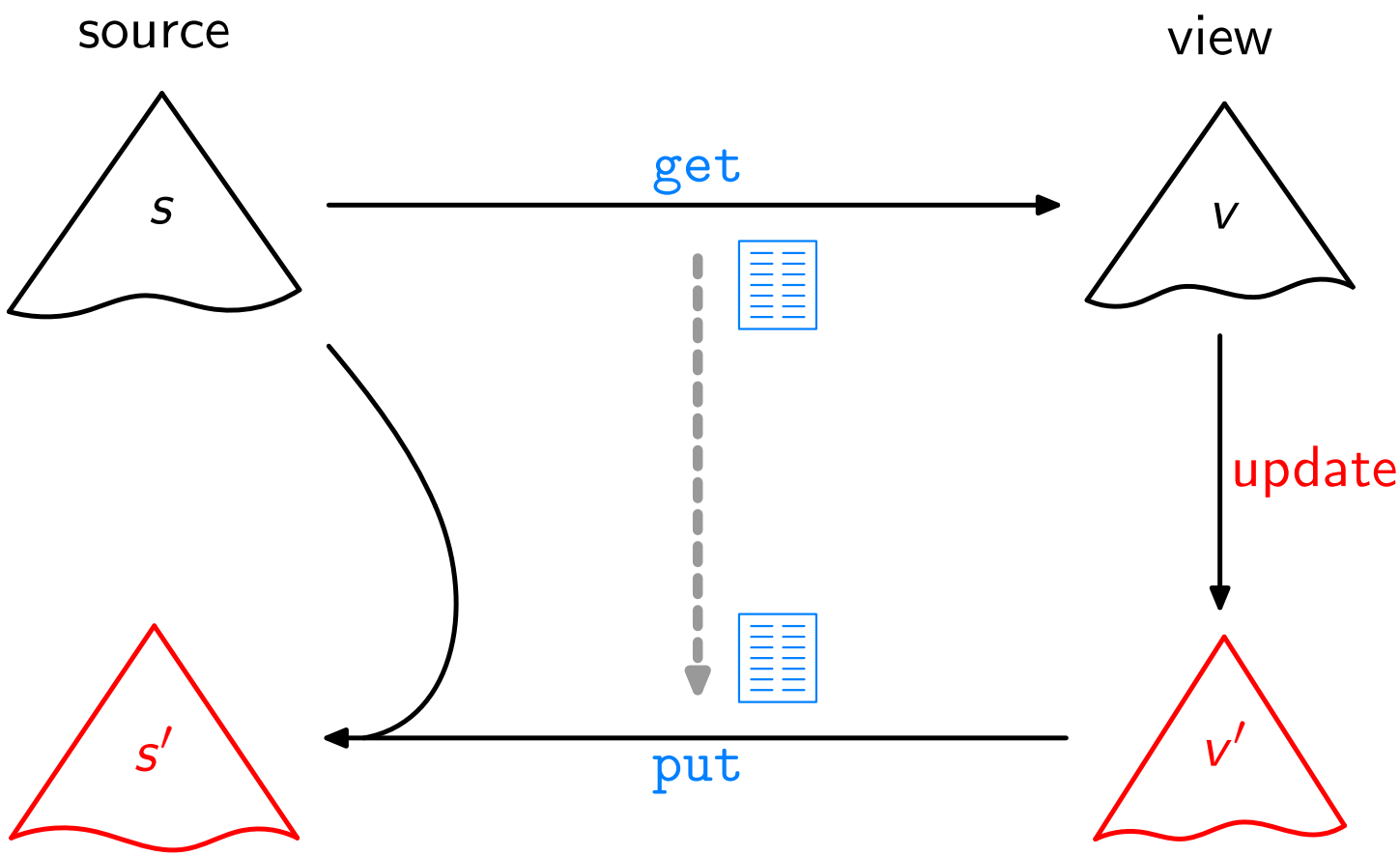
Bidirectional Transformation



Bidirectionalization

[Matsuda et al., ICFP'07]

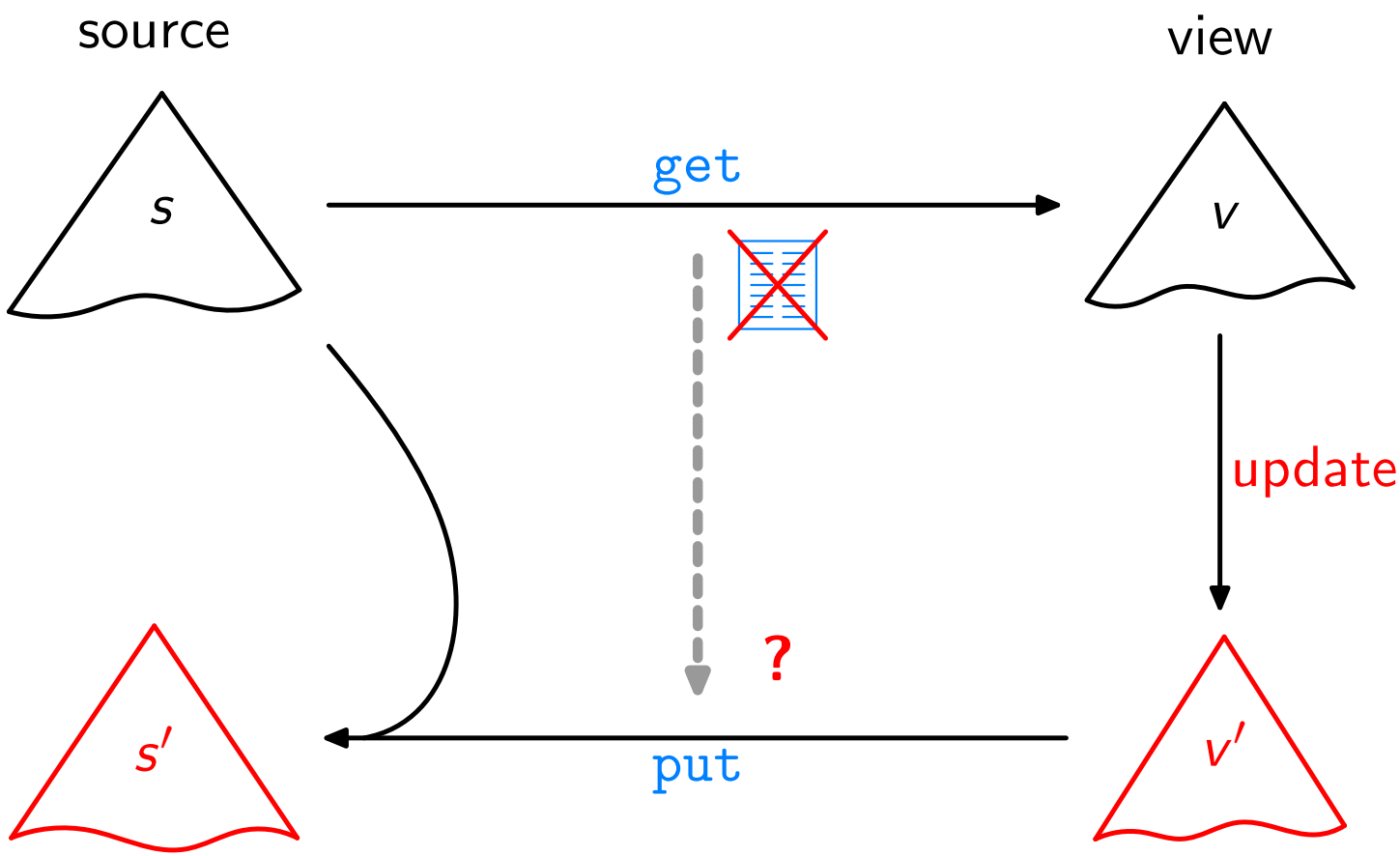
Bidirectional Transformation



Syntactic Bidirectionalization

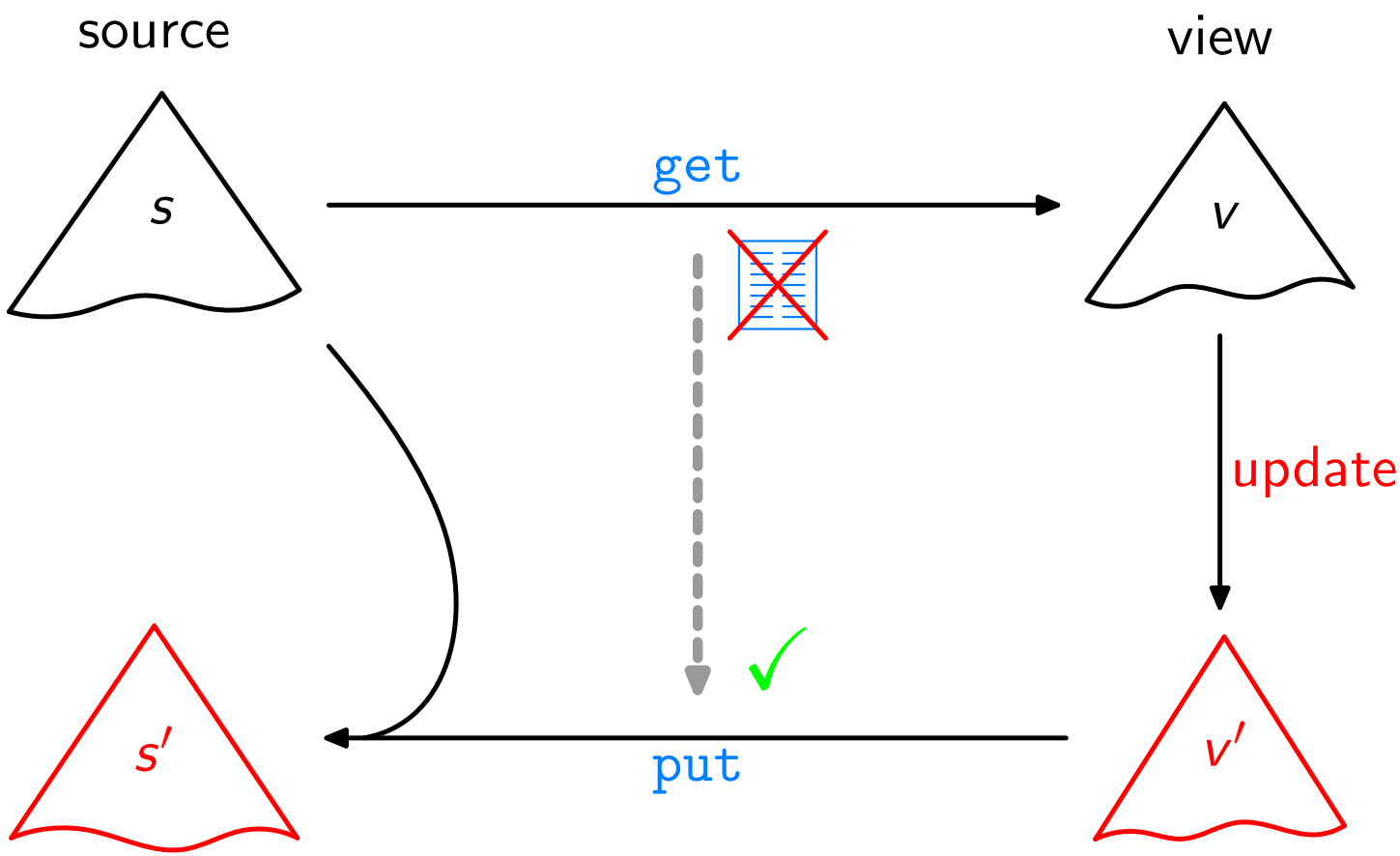
[Matsuda et al., ICFP'07]

Bidirectional Transformation



Semantic Bidirectionalization

Bidirectional Transformation



Semantic Bidirectionalization

[V., POPL'09]

Semantic Bidirectionalization

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

Semantic Bidirectionalization

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

[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

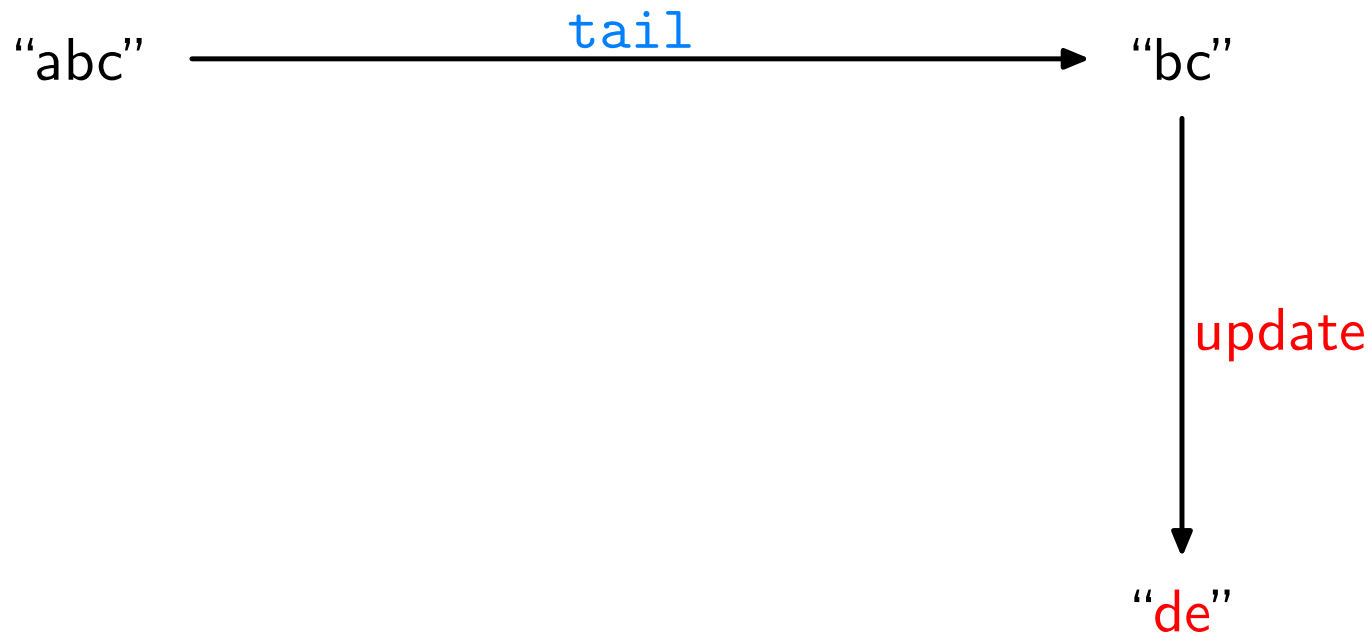
Examples:

[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:

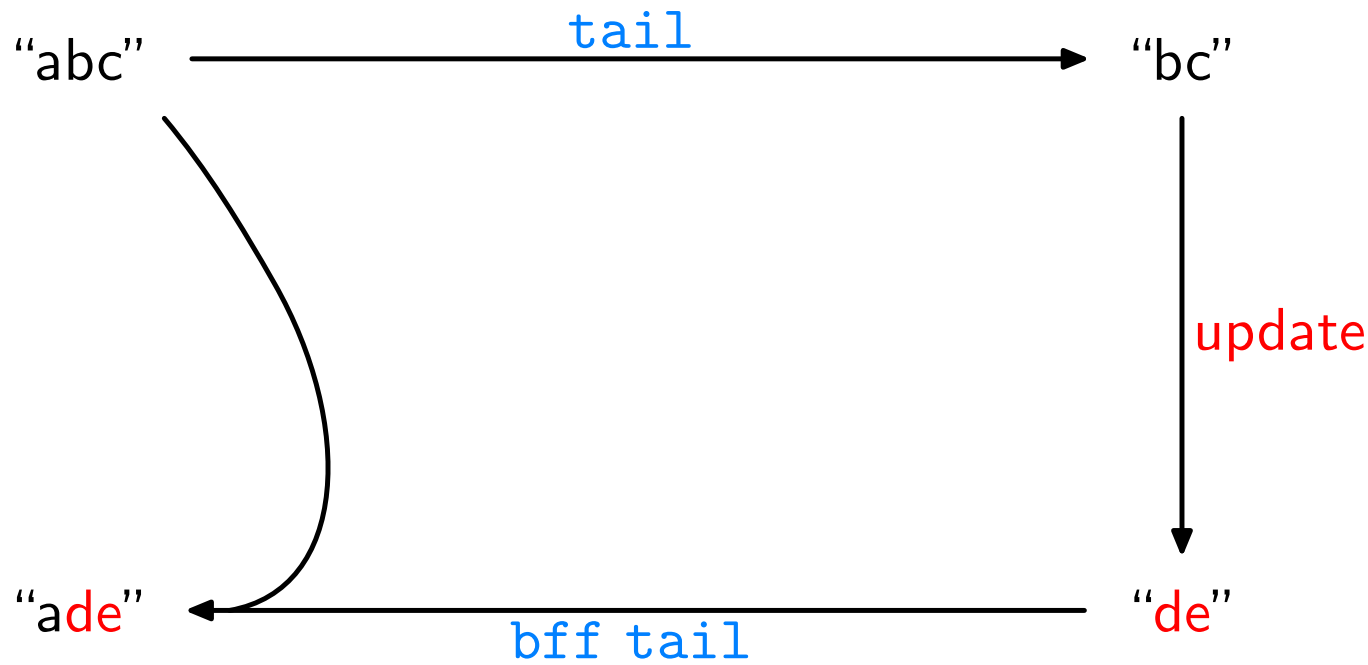


[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:

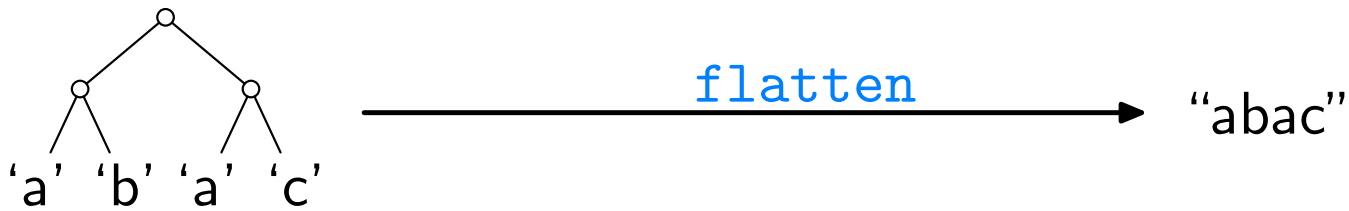


[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:

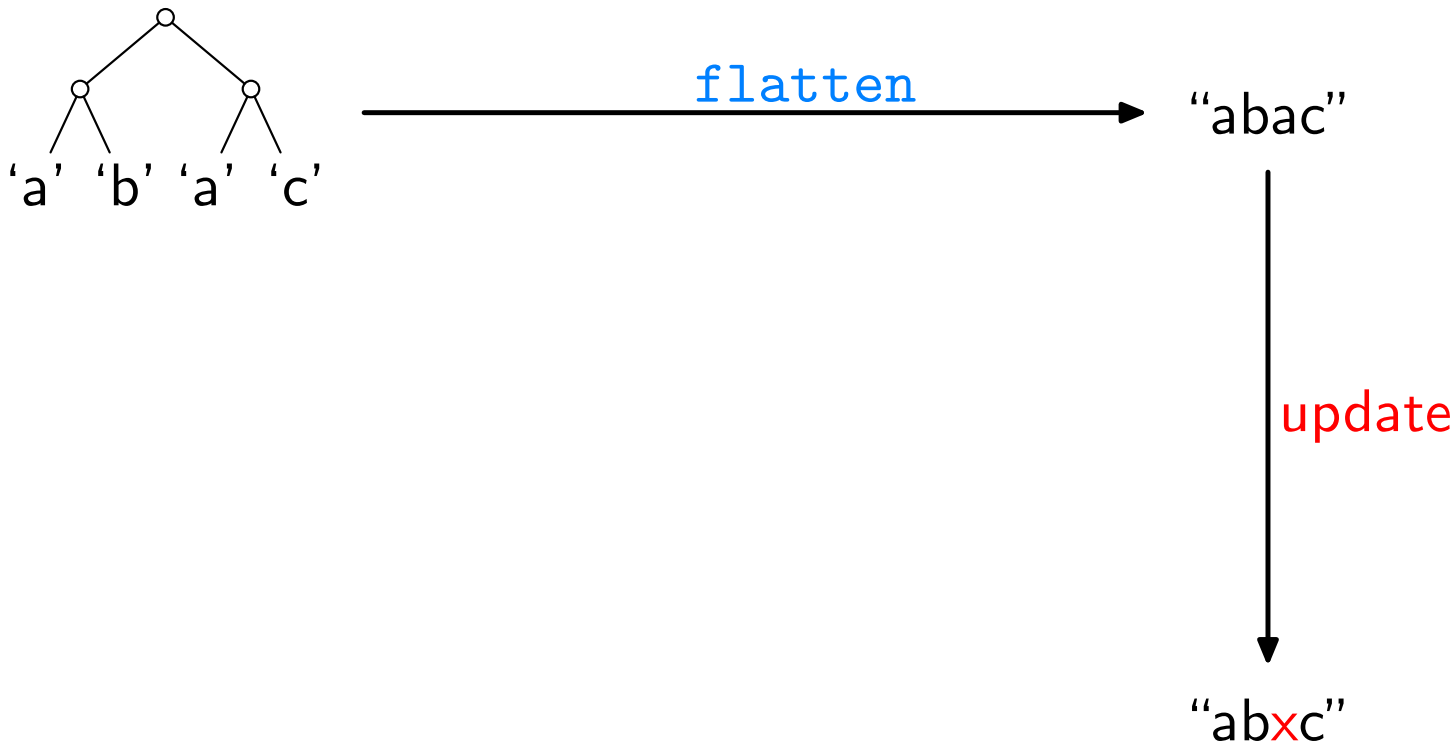


[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:

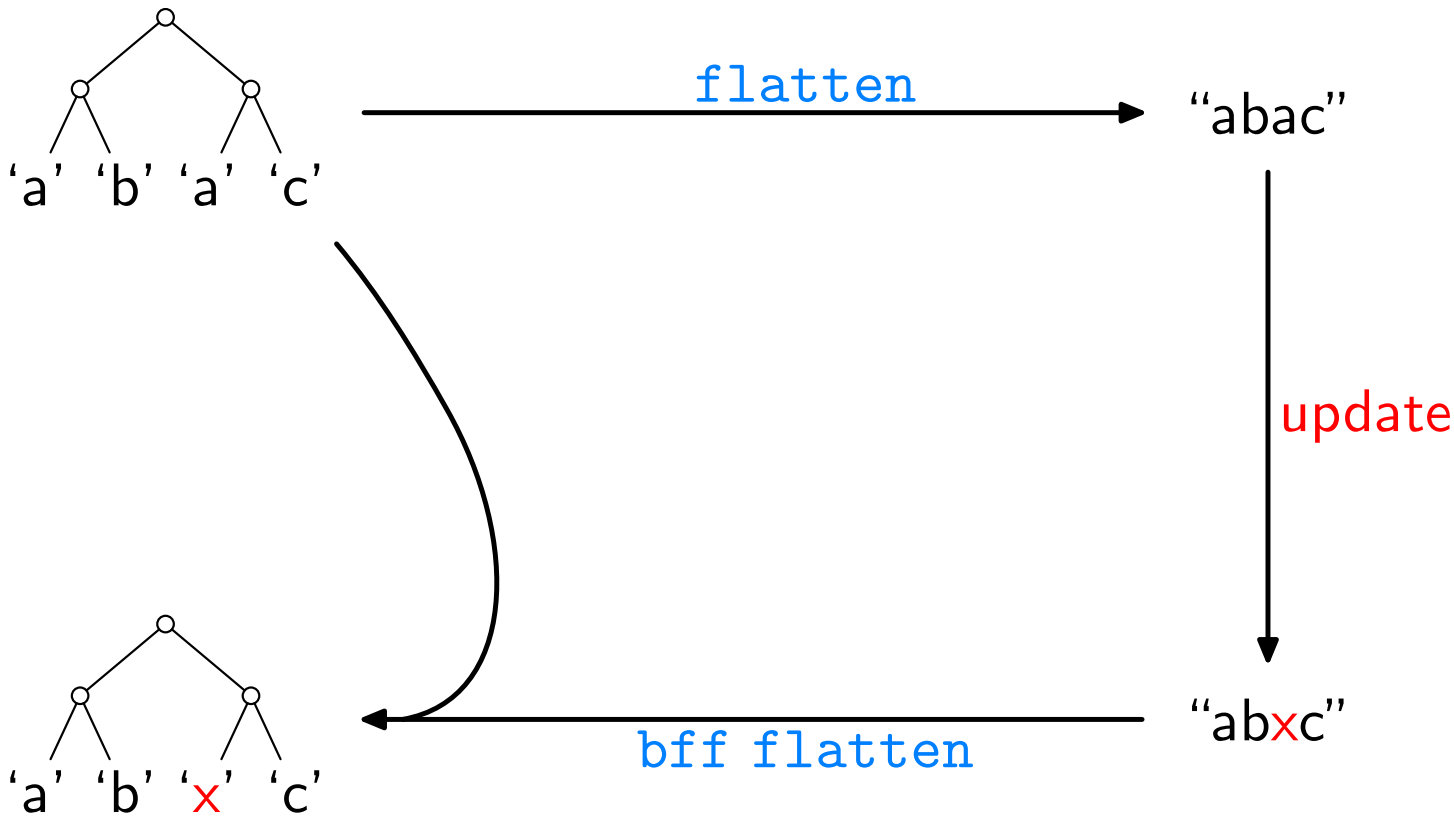


[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:

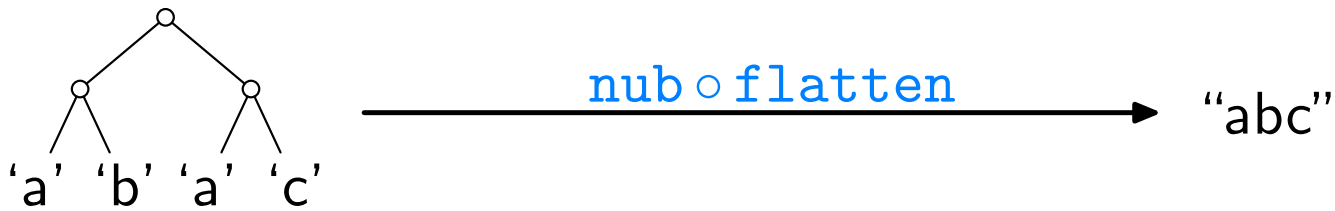


† “Bidirectionalization for free!”

Semantic Bidirectionalization

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

Examples:



[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

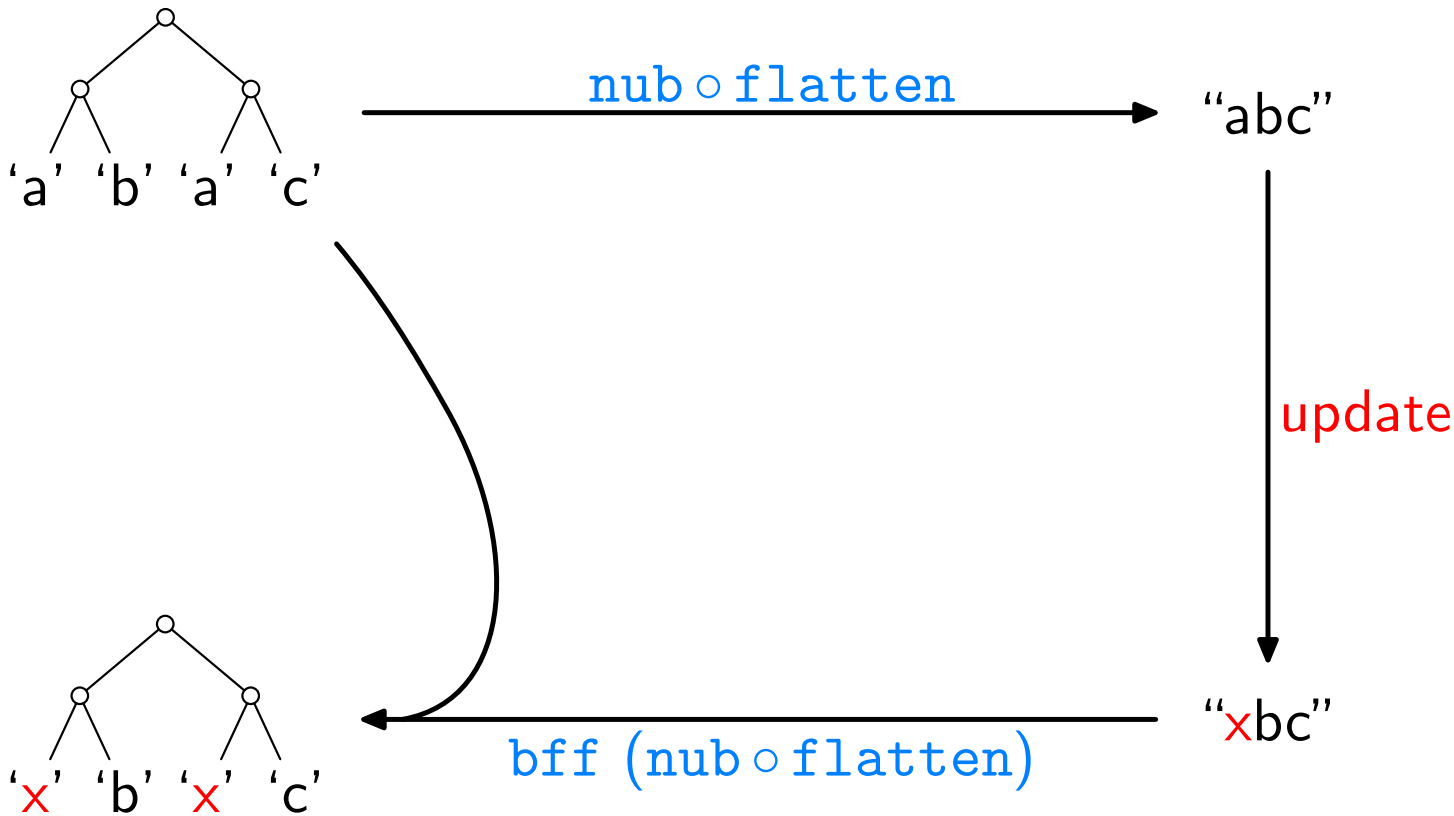
Examples:

[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

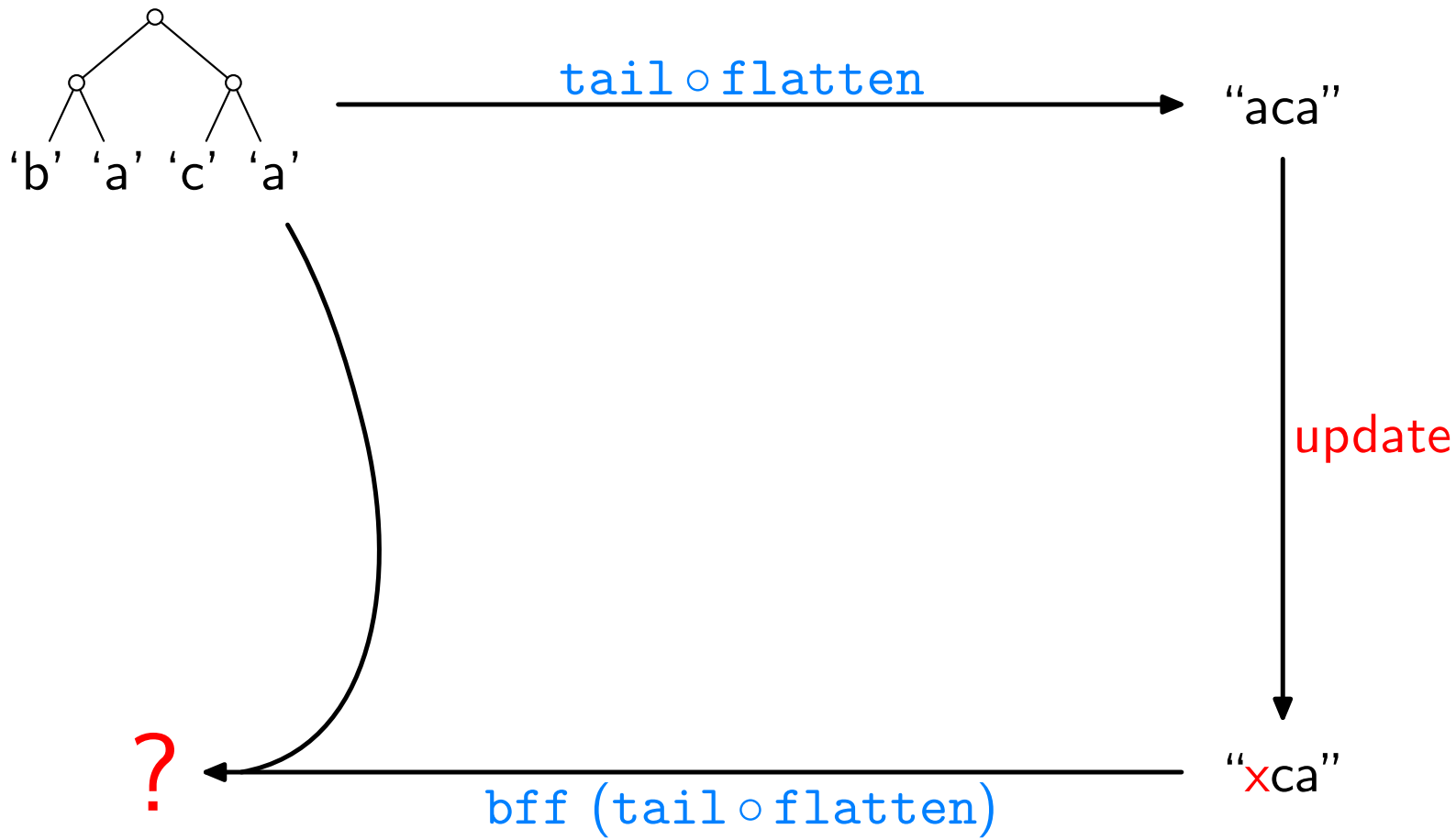
Aim: Write a higher-order function `bff`† such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

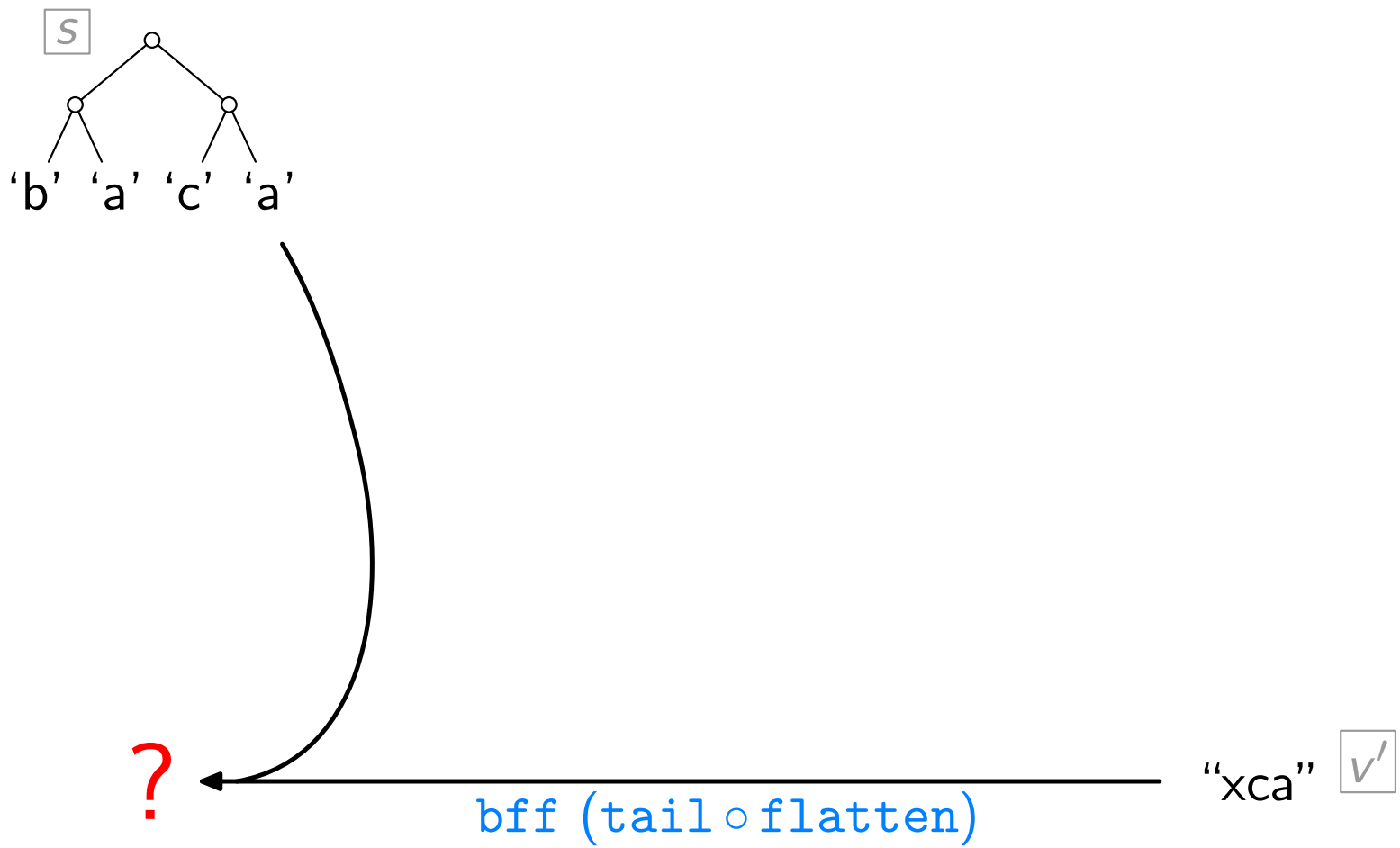


† "Bidirectionalization for free!"

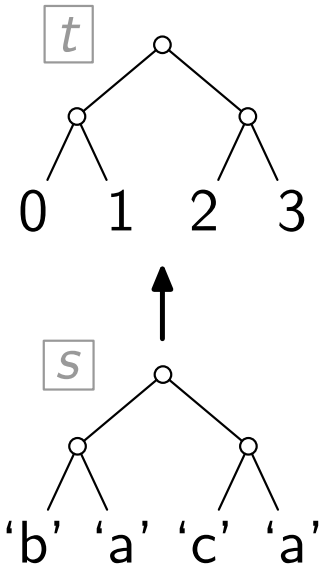
Overview of the Bidirectionalization Method



Overview of the Bidirectionalization Method

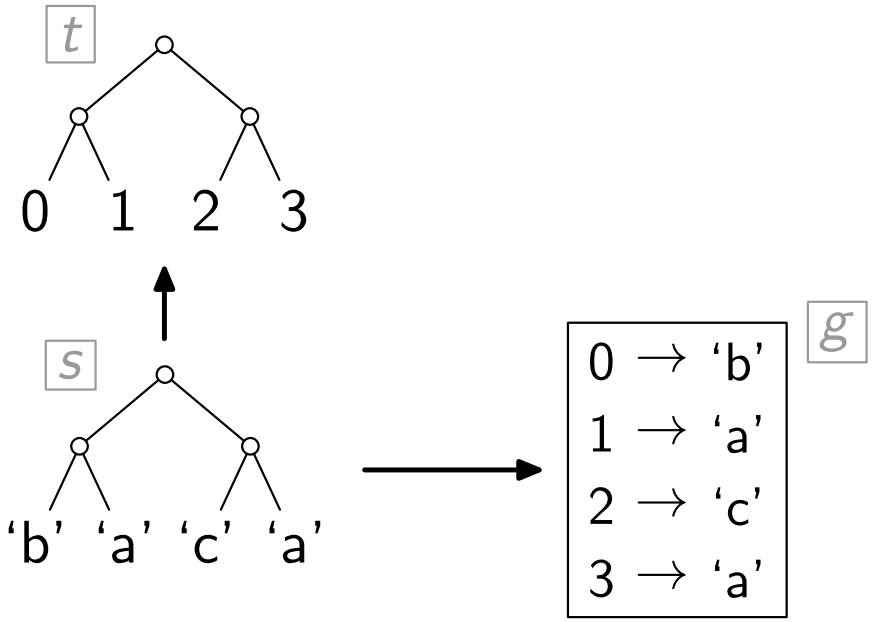


Overview of the Bidirectionalization Method



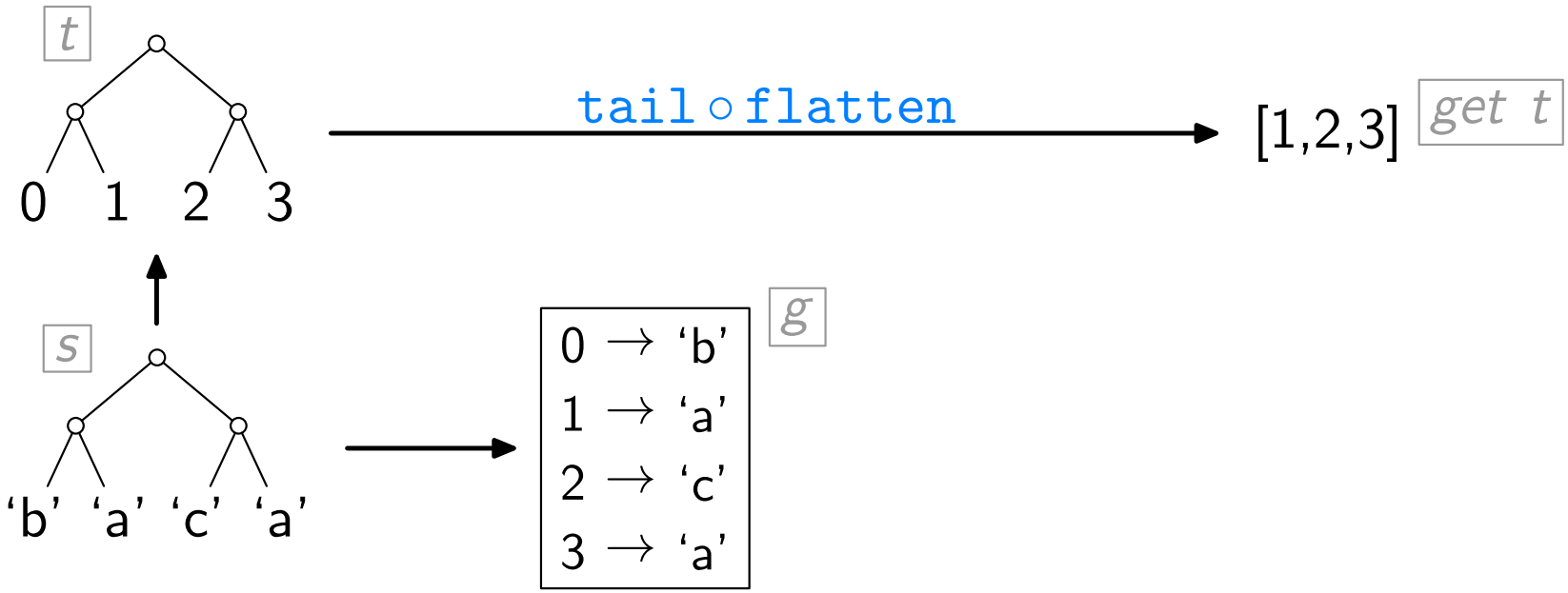
"xca" v'

Overview of the Bidirectionalization Method

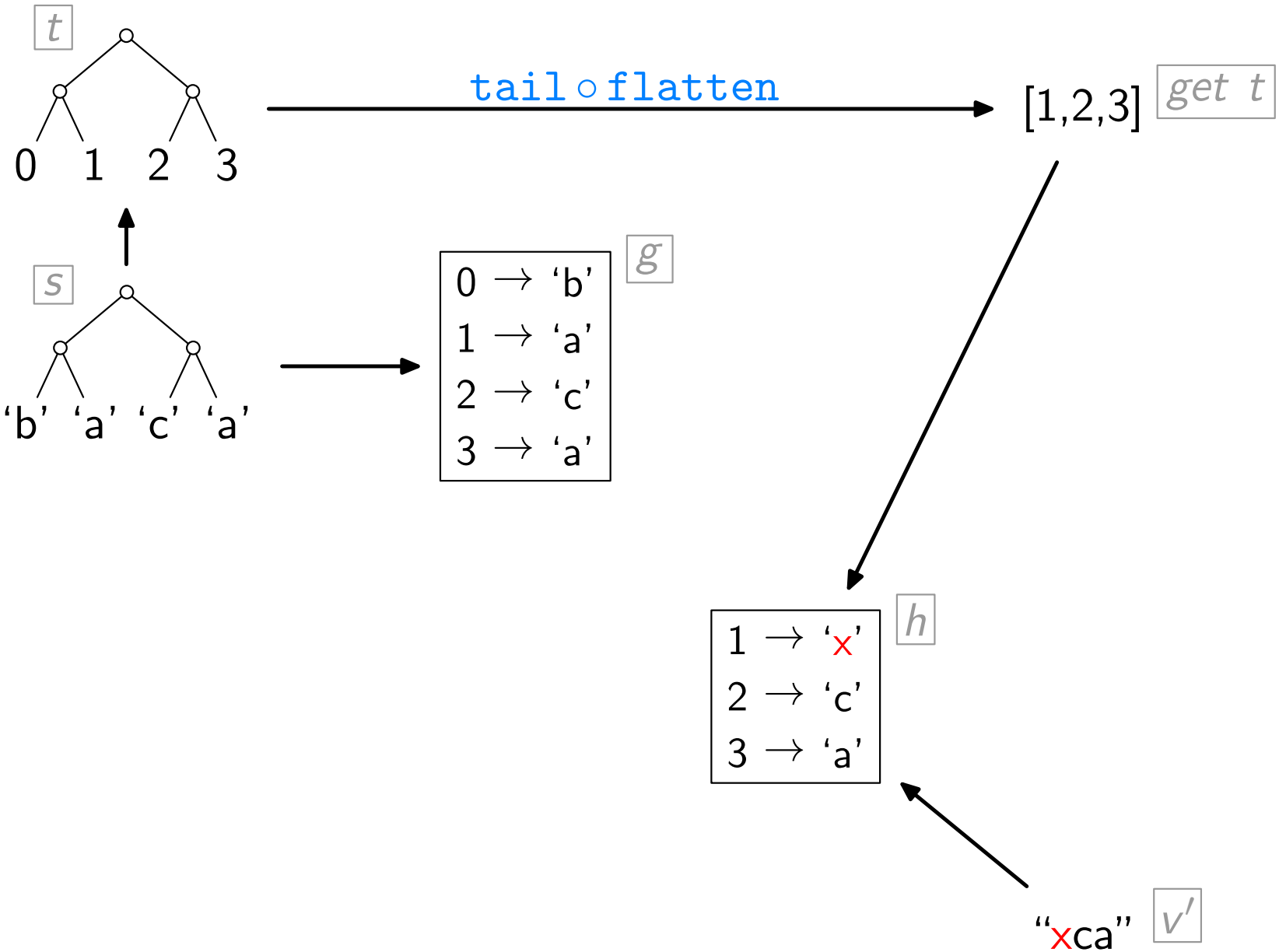


"xca" v'

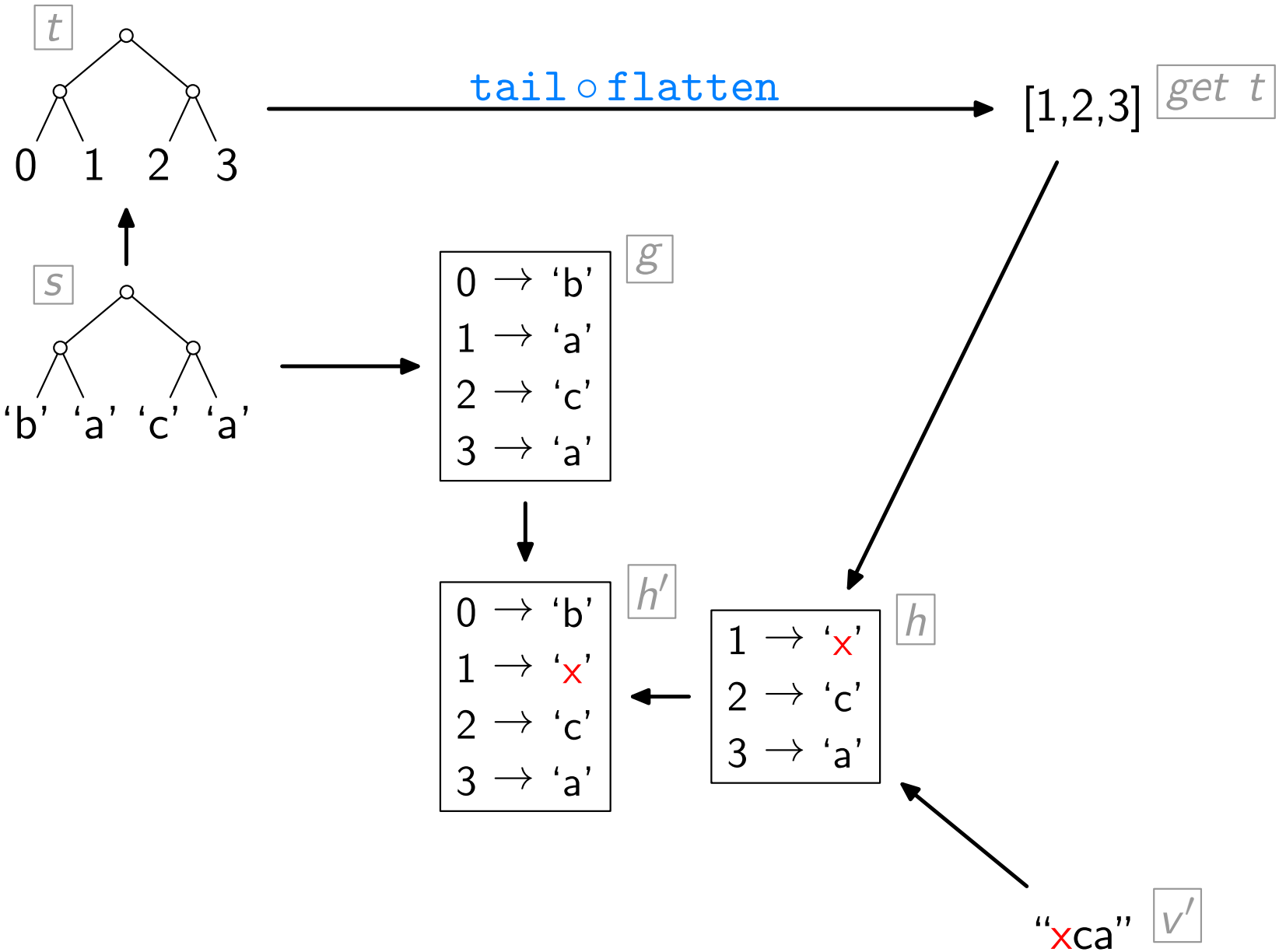
Overview of the Bidirectionalization Method



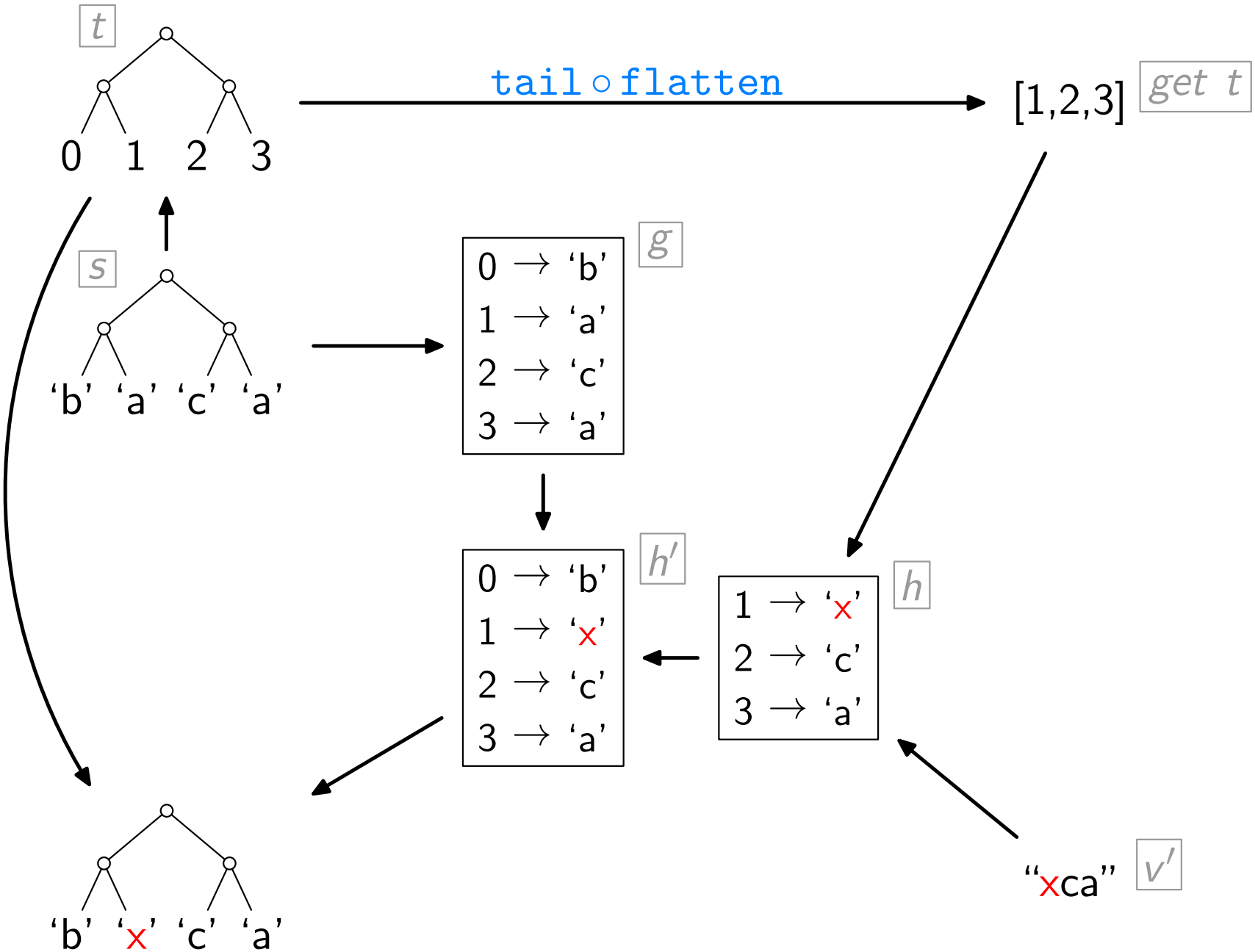
Overview of the Bidirectionalization Method



Overview of the Bidirectionalization Method



Overview of the Bidirectionalization Method



The Constant-Complement Approach [Bancilhon & Spyratos, ACM TODS'81]

In general, given

$$\texttt{get} :: S \to V$$

The Constant-Complement Approach [Bancilhon & Spyratos, ACM TODS'81]

In general, given

$$\texttt{get} :: S \to V$$

define a V^C and

$$\texttt{compl} :: S \to V^C$$

The Constant-Complement Approach [Bancilhon & Spyratos, ACM TODS'81]

In general, given

$$\texttt{get} :: S \to V$$

define a V^C and

$$\texttt{compl} :: S \to V^C$$

such that

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

is injective

The Constant-Complement Approach [Bancilhon & Spyratos, ACM TODS'81]

In general, given

$$\texttt{get} :: S \to V$$

define a V^C and

$$\texttt{compl} :: S \to V^C$$

such that

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

is injective and has an inverse

$$\texttt{inv} :: (V, V^C) \to S$$

The Constant-Complement Approach [Bancilhon & Spyratos, ACM TODS'81]

In general, given

$$\texttt{get} :: S \to V$$

define a V^C and

$$\texttt{compl} :: S \to V^C$$

such that

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

is injective and has an inverse

$$\texttt{inv} :: (V, V^C) \to S$$

Then:

$$\begin{aligned} &\texttt{put} :: S \to V \to S \\ &\texttt{put}\ s\ v' = \texttt{inv}\ (v', \texttt{compl}\ s) \end{aligned}$$

The Constant-Complement Approach [Bancilhon & Spyratos, ACM TODS'81]

In general, given

$$\texttt{get} :: S \to V$$

define a V^C and

$$\texttt{compl} :: S \to V^C$$

such that

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

is injective and has an inverse

$$\texttt{inv} :: (V, V^C) \to S$$

Then:

$$\begin{aligned} &\texttt{put} :: S \to V \to S \\ &\texttt{put}\ s\ v' = \texttt{inv}\ (v', \texttt{compl}\ s) \end{aligned}$$

Important: `compl` should "collapse" as much as possible.

The Constant-Complement Approach

For a very simple setting,

$$\texttt{get} :: [\alpha] \to [\alpha]\,,$$

what should be V^C and

$$\texttt{compl} :: [\alpha] \to V^C \quad ???$$

The Constant-Complement Approach

For a very simple setting,

$$\texttt{get} :: [\alpha] \to [\alpha]\,,$$

what should be V^C and

$$\texttt{compl} :: [\alpha] \to V^C \quad ???$$

To make

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

injective, need to record information discarded by `get`.

The Constant-Complement Approach

For a very simple setting,

$$\texttt{get} :: [\alpha] \to [\alpha]\,,$$

what should be V^C and

$$\texttt{compl} :: [\alpha] \to V^C \quad ???$$

To make

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

injective, need to record information discarded by `get`.

Candidates:

1. length of the source list

The Constant-Complement Approach

For a very simple setting,

$$\texttt{get} :: [\alpha] \to [\alpha]\,,$$

what should be V^C and

$$\texttt{compl} :: [\alpha] \to V^C \quad ???$$

To make

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

injective, need to record information discarded by `get`.

Candidates:

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

The Constant-Complement Approach

For a very simple setting,

$$\texttt{get} :: [\alpha] \to [\alpha]\,,$$

what should be V^C and

$$\texttt{compl} :: [\alpha] \to V^C \quad ???$$

To make

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

injective, need to record information discarded by `get`.

Candidates:

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

For the moment, be maximally conservative.

The Complement Function

$$
\begin{aligned}
&\texttt{compl} :: [\alpha] \to (\mathsf{Int}, [\alpha]) \\
&\texttt{compl}\ s = \textbf{let}\ n = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\quad t = [0..n] \\
&\qquad\qquad\quad g = \texttt{zip}\ t\ s \\
&\qquad\qquad\quad g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\qquad \textbf{in}\ \ (n + 1, \texttt{map}\ \texttt{snd}\ g')
\end{aligned}
$$

The Complement Function

$$
\begin{aligned}
&\texttt{compl} :: [\alpha] \to (\mathsf{Int}, [\alpha]) \\
&\texttt{compl}\ s = \textbf{let}\ n = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\quad t = [0..n] \\
&\qquad\qquad\quad g = \texttt{zip}\ t\ s \\
&\qquad\qquad\quad g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\qquad \textbf{in}\ \ (n + 1, \texttt{map}\ \texttt{snd}\ g')
\end{aligned}
$$

For example:

$$\texttt{get} = \texttt{tail} \qquad \rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [\text{'a'}])$$

The Complement Function

$$
\begin{aligned}
&\texttt{compl} :: [\alpha] \to (\mathsf{Int}, [\alpha]) \\
&\texttt{compl}\ s = \textbf{let}\ n = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\quad t = [0..n] \\
&\qquad\qquad\quad g = \texttt{zip}\ t\ s \\
&\qquad\qquad\quad g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\qquad \textbf{in}\ \ (n + 1, \texttt{map}\ \texttt{snd}\ g')
\end{aligned}
$$

For example:

$$
\begin{aligned}
\texttt{get} = \texttt{tail} \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [\text{'a'}]) \\
\texttt{get} = \texttt{take}\ 3 \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [\text{'d'}, \text{'e'}])
\end{aligned}
$$

The Complement Function

$$
\begin{aligned}
&\texttt{compl} :: [\alpha] \to (\mathsf{Int}, [\alpha]) \\
&\texttt{compl}\ s = \textbf{let}\ n = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\quad t = [0..n] \\
&\qquad\qquad\quad g = \texttt{zip}\ t\ s \\
&\qquad\qquad\quad g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\qquad \textbf{in}\ \ (n + 1, \texttt{map}\ \texttt{snd}\ g')
\end{aligned}
$$

For example:

$$
\begin{aligned}
\texttt{get} = \texttt{tail} \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [\text{'a'}]) \\
\texttt{get} = \texttt{take}\ 3 \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [\text{'d'}, \text{'e'}]) \\
\texttt{get} = \texttt{reverse} \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [\,])
\end{aligned}
$$

An Inverse of $\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], (\mathsf{Int}, [\alpha])) \to [\alpha] \\
&\texttt{inv}\ ([\,], (0, \_)) = [\,] \\
&\texttt{inv}\ (v', (n+1, as)) = \\
&\quad \textbf{let}\ t \ = [0..n] \\
&\qquad\quad h \ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
&\qquad\quad g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ t)\ as \\
&\qquad\quad h' = h +\!\!+ g' \\
&\quad \textbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t
\end{aligned}
$$

An Inverse of $\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$

```
inv :: ([α], (Int, [α])) → [α]
inv ([], (0, _)) = []
inv (v', (n + 1, as)) =
  let t  = [0..n]
      h  = assoc† (get t) v'
      g' = zip (filter (λi → notElem i (get t)) t) as
      h' = h ++ g'
  in  map (λi → fromJust (lookup i h')) t
```

† Can be thought of as `zip` for the moment.

An Inverse of $\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], (\mathsf{Int}, [\alpha])) \to [\alpha] \\
&\texttt{inv}\ ([\,], (0, \_)) = [\,] \\
&\texttt{inv}\ (v', (n+1, as)) = \\
&\quad \textbf{let}\ t \ = [0..n] \\
&\qquad\quad h \ = \texttt{assoc}^{\dagger}\ (\texttt{get}\ t)\ v' \\
&\qquad\quad g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ t)\ as \\
&\qquad\quad h' = h +\!\!+ g' \\
&\quad \textbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t
\end{aligned}
$$

For example:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{inv}\ (\text{“bcde”}, (5, [\text{'a'}])) = \text{“abcde”}$$

† Can be thought of as `zip` for the moment.

An Inverse of $\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], (\mathsf{Int}, [\alpha])) \to [\alpha] \\
&\texttt{inv}\ ([\,], (0, \_)) = [\,] \\
&\texttt{inv}\ (v', (n+1, as)) = \\
&\quad \textbf{let}\ t \ = [0..n] \\
&\qquad\quad h \ = \texttt{assoc}^{\dagger}\ (\texttt{get}\ t)\ v' \\
&\qquad\quad g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ t)\ as \\
&\qquad\quad h' = h +\!\!+ g' \\
&\quad \textbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t
\end{aligned}
$$

For example:

$$
\begin{aligned}
\texttt{get} = \texttt{tail} \quad &\rightsquigarrow \quad \texttt{inv}\ (\text{“bcde”}, (5, [\text{'a'}])) = \text{“abcde”} \\
\texttt{get} = \texttt{take}\ 3 \quad &\rightsquigarrow \quad \texttt{inv}\ (\text{“xyz”}, (5, [\text{'d'}, \text{'e'}])) = \text{“xyzde”}
\end{aligned}
$$

† Can be thought of as `zip` for the moment.

Correctness

To prove formally:

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

Correctness

To prove formally:

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

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

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have, for arbitrary f and l,

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)\,.$$

Correctness

To prove formally:

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

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

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have, for arbitrary f and l,

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)\,.$$

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

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

Correctness

To prove formally:

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

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

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have, for arbitrary f and l,

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)\,.$$

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

$$\begin{aligned} \texttt{map}\ (s\,!!)\ (\texttt{get}\ [0..n]) &= \texttt{get}\ (\underbrace{\texttt{map}\ (s\,!!)\ [0..n]}) \\ &= \texttt{get}\ \qquad\qquad s \end{aligned}$$

Correctness

To prove formally:

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

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

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have, for arbitrary f and l,

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)\,.$$

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

$$\texttt{map}\ (s\,!!)\ (\texttt{get}\ [0..n])$$

$$= \quad \texttt{get}\ s$$

Correctness

To prove formally:

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

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

$$\texttt{get} :: [\alpha] \to [\alpha]$$

we have, for arbitrary f and l,

$$\texttt{map}\ f\ (\texttt{get}\ l) \quad = \quad \texttt{get}\ (\texttt{map}\ f\ l)\,.$$

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

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

Altogether, So Far:

```
compl :: [α] → (Int, [α])
compl s = let n  = (length s) − 1
              t  = [0..n]
              g  = zip t s
              g' = filter (λ(i, _) → notElem i (get t)) g
          in  (n + 1, map snd g')

inv :: ([α], (Int, [α])) → [α]
inv ([], (0, _)) = []
inv (v', (n + 1, as)) =
  let t  = [0..n]
      h  = assoc (get t) v'
      g' = zip (filter (λi → notElem i (get t)) t) as
      h' = h ++ g'
  in  map (λi → fromJust (lookup i h')) t
```

“Fusion”

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

$$
\begin{aligned}
&\texttt{put}\ [\,]\ [\,] = [\,] \\
&\texttt{put}\ s\ v' = \textbf{let}\ n = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\quad\ t = [0..n] \\
&\qquad\qquad\quad\ g = \texttt{zip}\ t\ s \\
&\qquad\qquad\quad\ g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\qquad\quad\ h = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\quad\ h' = h +\!\!+ g' \\
&\qquad\qquad\ \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

“Fusion”

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

```
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
```

“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
```

"Fusion"

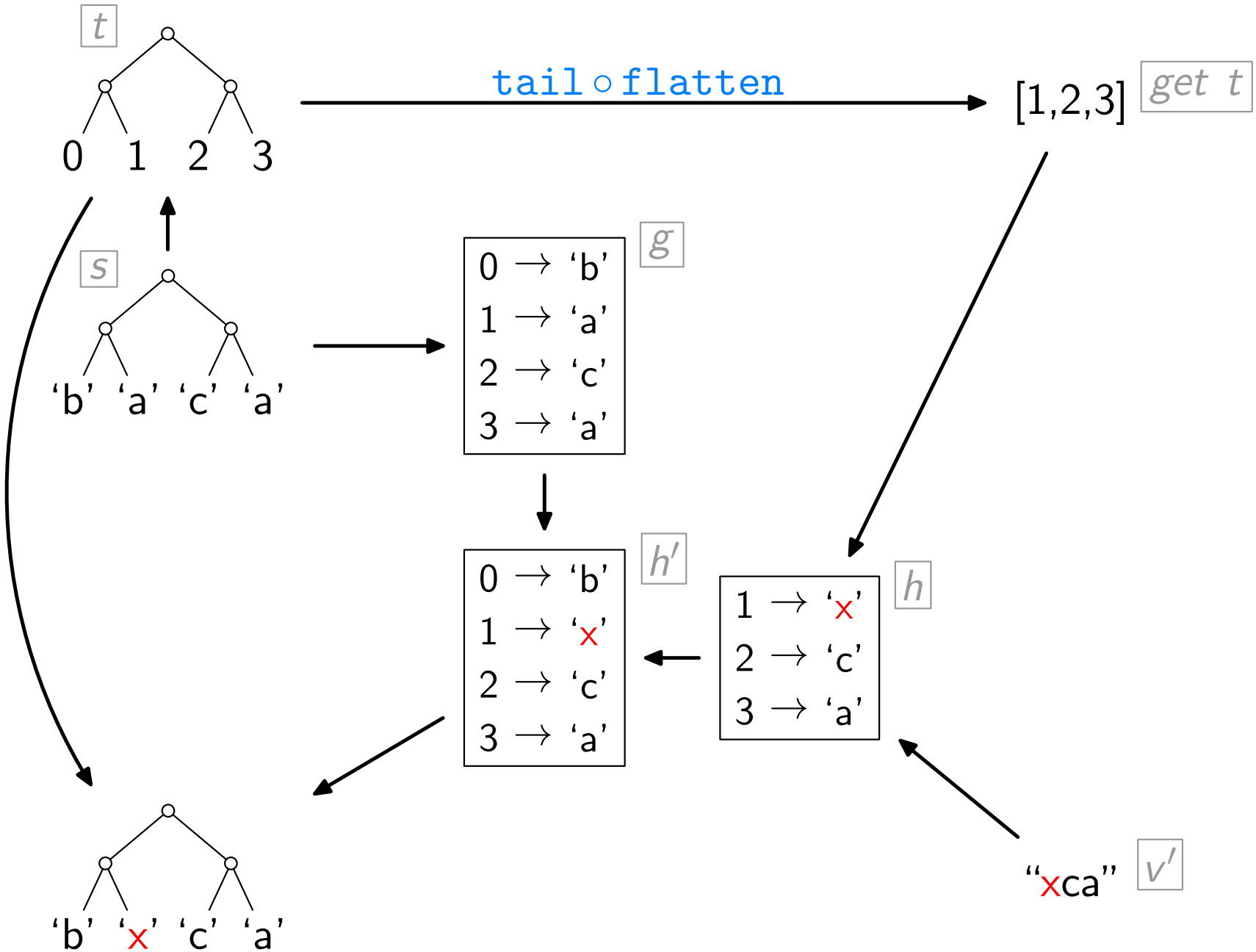
Inlining `compl` and `inv` into `put`, plus some clever rewriting:

```
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!

Overview of the Bidirectionalization Method



Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.

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, `bff tail` “abcde” “xyz” . . .

Analysis as to Why:

- Our approach to making

$$\lambda s \rightarrow (\texttt{get}\ s, \texttt{compl}\ s)$$

injective was to record, via `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, `bff tail` "abcde" "xyz" ...

Analysis as to Why:

- Our approach to making

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

injective was to record, via `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, `bff tail` “abcde” “xyz” . . .

Analysis as to Why:

- Our approach to making

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

injective was to record, via `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:

`get = tail` $\rightsquigarrow$ `put` “abcde” “xyz” fails precisely because `compl` “abcde” $= (5, [\text{'a'}])$

Assuming Shape-Injectivity

So assume there is a function

$$\texttt{shapeInv} :: \mathsf{Int} \to \mathsf{Int}$$

with, for every source list s,

$$\texttt{length}\ s = \texttt{shapeInv}\ (\texttt{length}\ (\texttt{get}\ s))$$

Assuming Shape-Injectivity

So assume there is a function

$$\texttt{shapeInv} :: \mathsf{Int} \to \mathsf{Int}$$

with, for every source list s,

$$\texttt{length}\ s = \texttt{shapeInv}\ (\texttt{length}\ (\texttt{get}\ s))$$

Then:

$$\begin{aligned}
&\texttt{compl} :: [\alpha] \to (\mathsf{Int}, [\alpha]) \\
&\texttt{compl}\ s = \textbf{let}\ n\ = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\quad\ t\ = [0..n] \\
&\qquad\qquad\quad\ g\ = \texttt{zip}\ t\ s \\
&\qquad\qquad\quad\ g' = \texttt{filter}\ (\lambda(i, _) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\qquad\ \textbf{in}\ \ (n + 1, \texttt{map}\ \texttt{snd}\ g')
\end{aligned}$$

Assuming Shape-Injectivity

So assume there is a function

$$\texttt{shapeInv} :: \mathsf{Int} \to \mathsf{Int}$$

with, for every source list s,

$$\texttt{length}\ s = \texttt{shapeInv}\ (\texttt{length}\ (\texttt{get}\ s))$$

Then:

$$\begin{array}{lll}
\texttt{compl} :: [\alpha] \to & & [\alpha] \\
\texttt{compl}\ s = \textbf{let}\ n & = & (\texttt{length}\ s) - 1 \\
\qquad\qquad\quad\ t & = & [0..n] \\
\qquad\qquad\quad\ g & = & \texttt{zip}\ t\ s \\
\qquad\qquad\quad\ g' & = & \texttt{filter}\ (\lambda(i, _) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
\qquad\quad \textbf{in} & & \qquad \texttt{map}\ \texttt{snd}\ g'
\end{array}$$

Assuming Shape-Injectivity

```
inv :: ([α], (Int, [α])) → [α]
inv ([], (0, _)) = []
inv (v', (n + 1, as)) =
  let t  = [0..n]
      h  = assoc (get t) v'
      g' = zip (filter (λi → notElem i (get t)) t) as
      h' = h ++ g'
  in  map (λi → fromJust (lookup i h')) t
```

Assuming Shape-Injectivity

```
inv :: ([α],      [α] ) → [α]
inv ([],    _ ) = []
inv (v′,        as ) =
  let n  = (shapeInv (length v′)) − 1
      t  = [0..n]
      h  = assoc (get t) v′
      g′ = zip (filter (λi → notElem i (get t)) t) as
      h′ = h ++ g′
  in  map (λi → fromJust (lookup i h′)) t
```

Assuming Shape-Injectivity

```
inv :: ([α],      [α] ) → [α]
inv ([],    _ ) = []
inv (v',        as ) =
  let n  = (shapeInv (length v')) − 1
      t  = [0..n]
      h  = assoc (get t) v'
      g' = zip (filter (λi → notElem i (get t)) t) as
      h' = h ++ g'
  in  map (λi → fromJust (lookup i h')) t
```

But how to obtain `shapeInv` ???

Assuming Shape-Injectivity

```
inv :: ([α],      [α] ) → [α]
inv ([],    _ ) = []
inv (v',        as ) =
  let n  = (shapeInv (length v')) − 1
      t  = [0..n]
      h  = assoc (get t) v'
      g' = zip (filter (λi → notElem i (get t)) t) as
      h' = h ++ g'
  in  map (λi → fromJust (lookup i h')) t
```

But how to obtain `shapeInv` ???

Just for experimentation:

```
shapeInv :: Int → Int
shapeInv lv = head [n + 1 | n ← [0..], (length (get [0..n])) == lv]
```

Some Tests

Works quite nicely in some cases:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“axyz”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'a'}]$$

Some Tests

Works quite nicely in some cases:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“axyz”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'a'}]$$

$$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“xyze”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'e'}]$$

Some Tests

Works quite nicely in some cases:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“axyz”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'a'}]$$

$$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“xyze”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'e'}]$$

But not so in others:

$$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“abc”} = \text{“abc”}$$

Some Tests

Works quite nicely in some cases:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“axyz”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'a'}]$$

$$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“xyze”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'e'}]$$

But not so in others:

$$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“abc”} = \text{“abc”}$$

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

Some Tests

Works quite nicely in some cases:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“axyz”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'a'}]$$

$$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“xyze”}\text{, using } \texttt{compl}\ \text{“abcde”} = [\text{'e'}]$$

But not so in others:

$$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“abc”} = \text{“abc”}$$

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

Better:

$$\texttt{shapeInv} :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$$
$$\texttt{shapeInv}\ l_s\ l_v = \texttt{head}\ [n+1 \mid n \leftarrow (l_s - 1) : [0..],\ (\texttt{length}\ (\texttt{get}\ [0..n])) == l_v]$$

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

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

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. Spyratos.
Update semantics of relational views.
ACM Transactions on Database Systems, 6(3):557–575, 1981.


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.
ACM Transactions on Programming Languages and Systems, 29(3):17, 2007.


K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi.
Bidirectionalization transformation based on automatic derivation of view complement functions.
In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.

References II

J. Voigtländer.
Bidirectionalization for free!
In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

P. Wadler.
Theorems for free!
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.