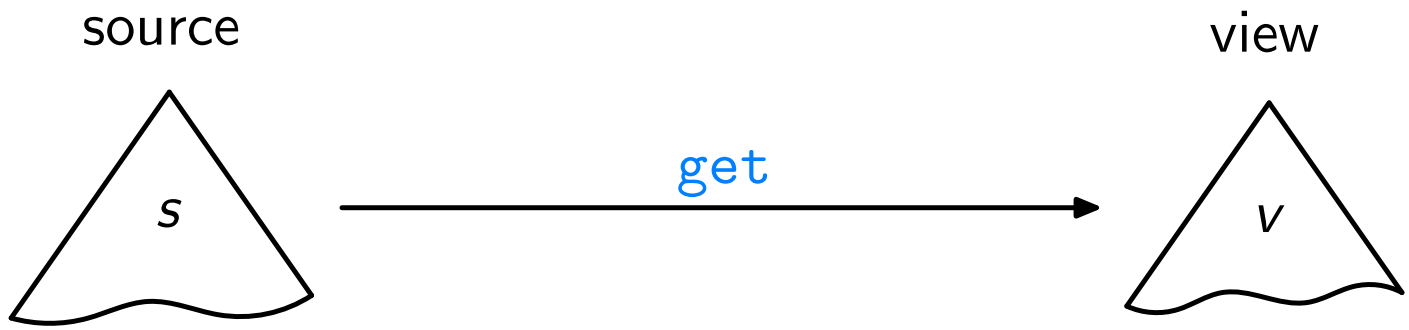
Bidirectionalization for Free!

Janis Voigtländer

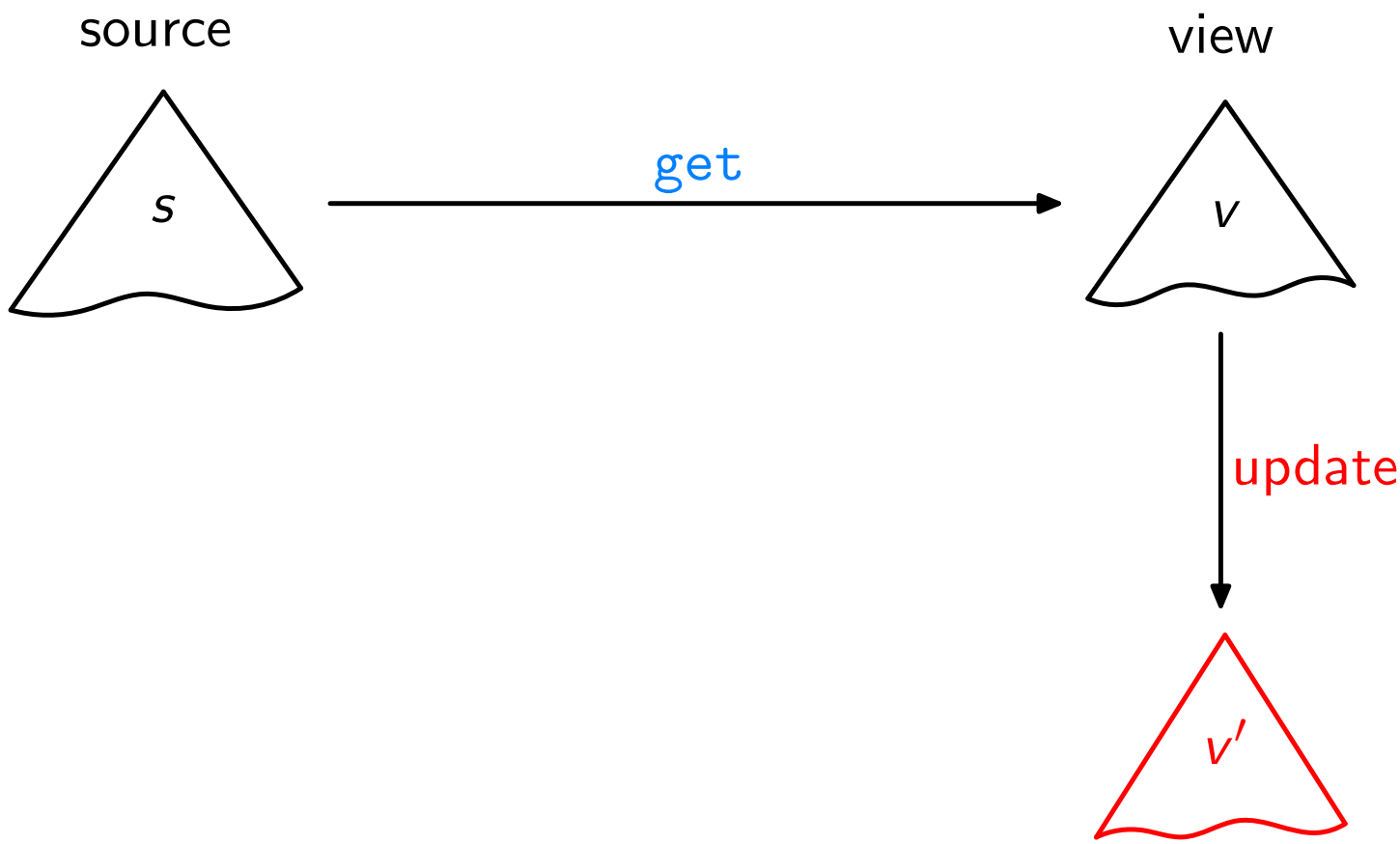
Technische Universität Dresden

POPL'09

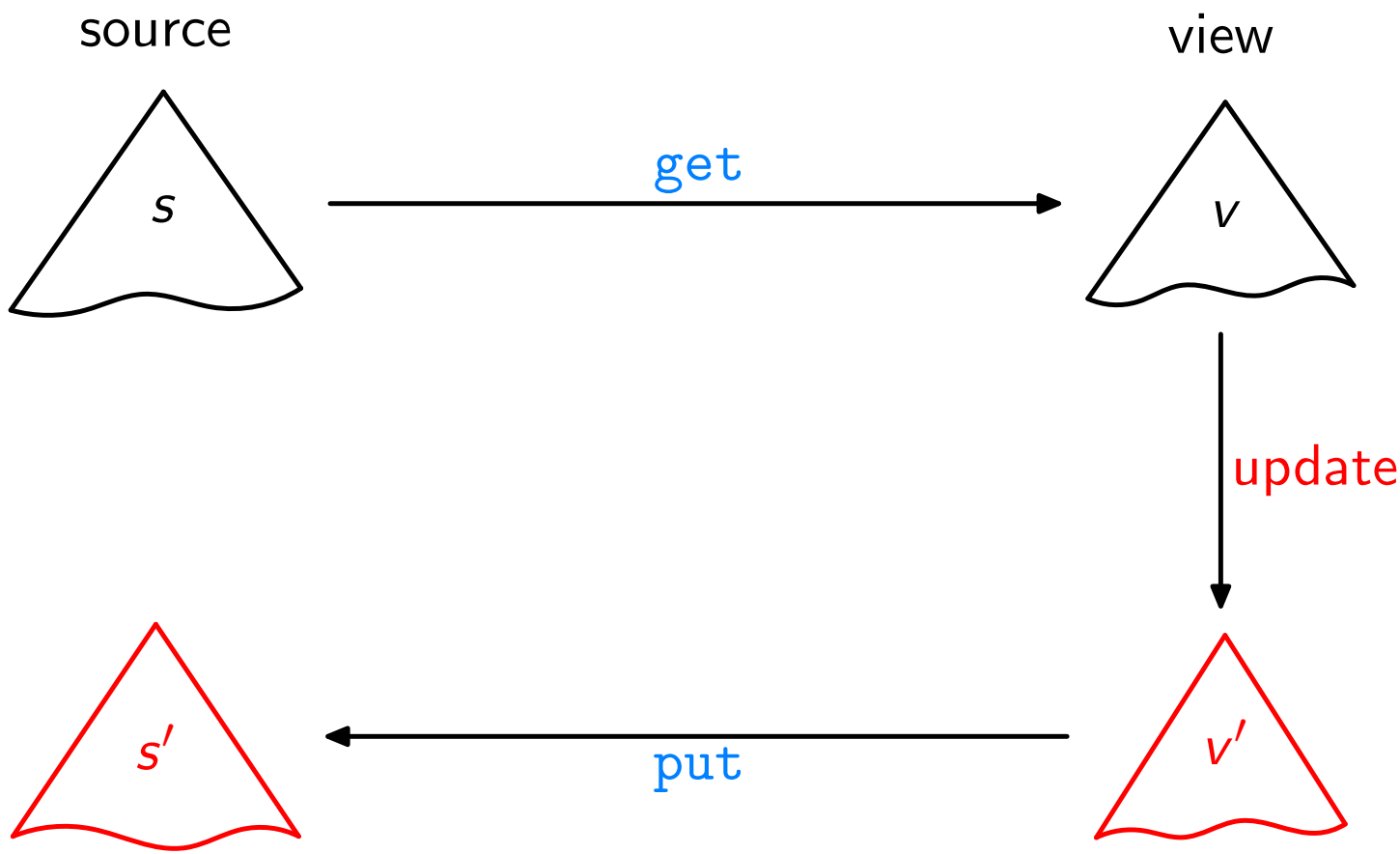
Bidirectional Transformation



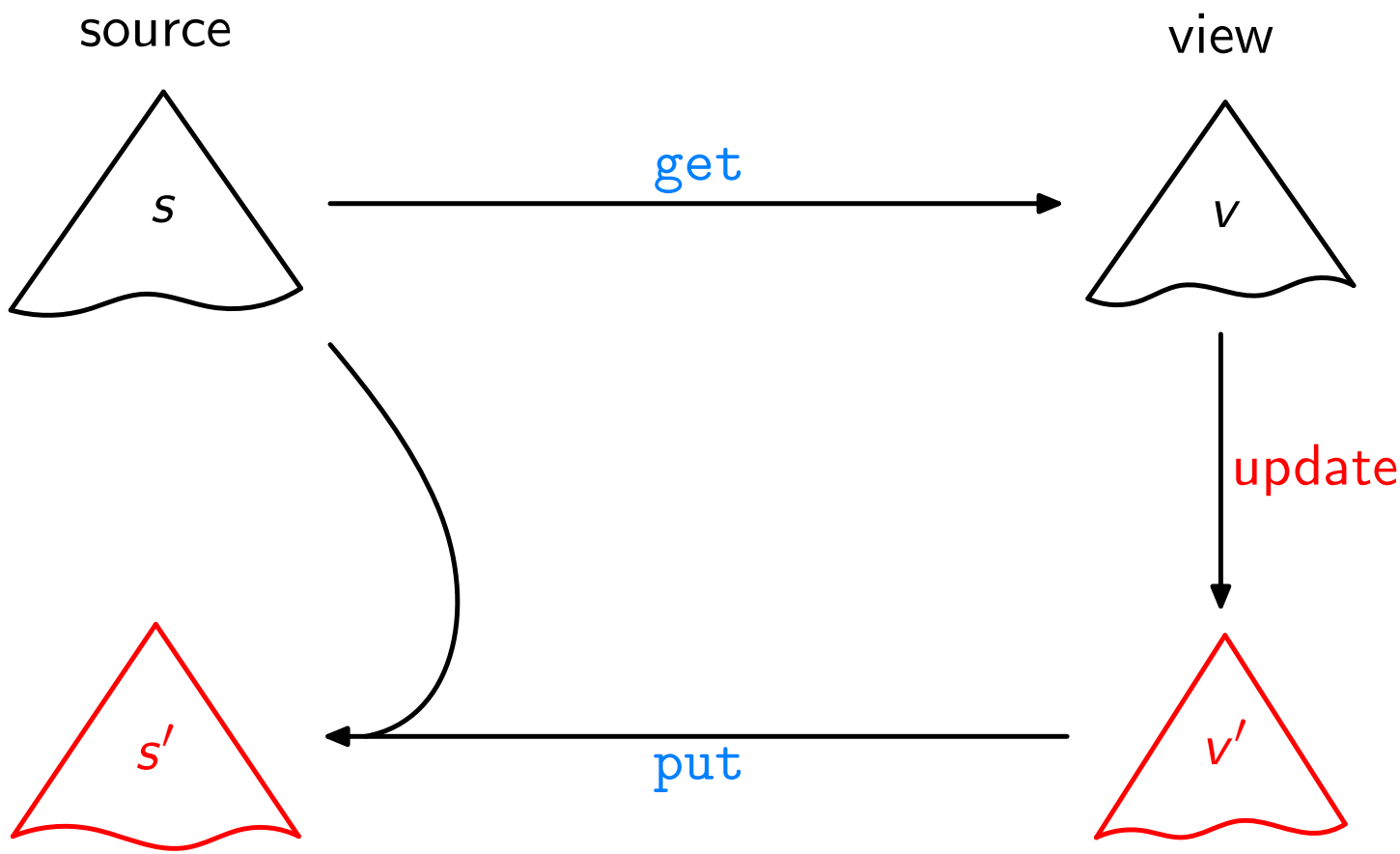
Bidirectional Transformation



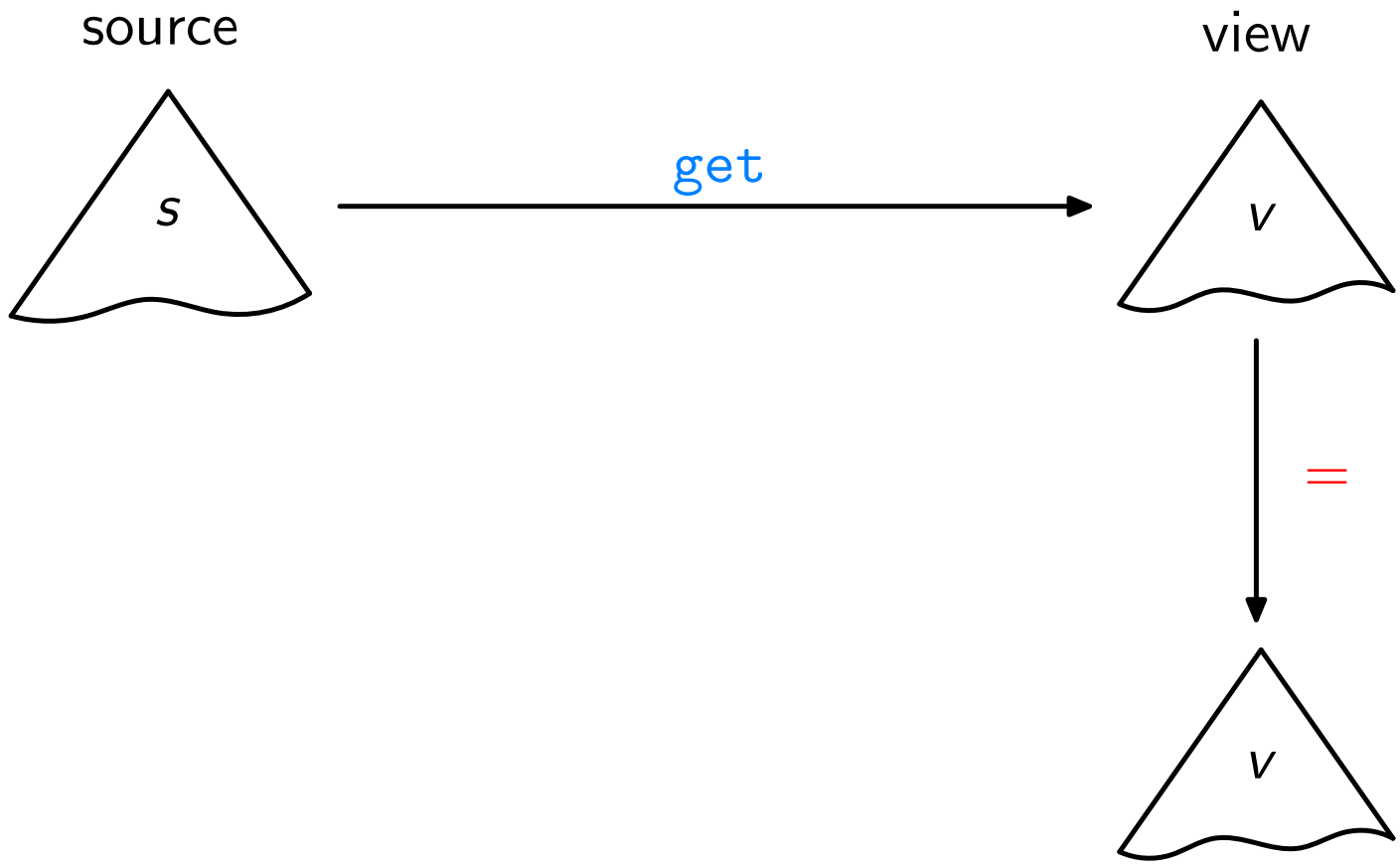
Bidirectional Transformation



Bidirectional Transformation

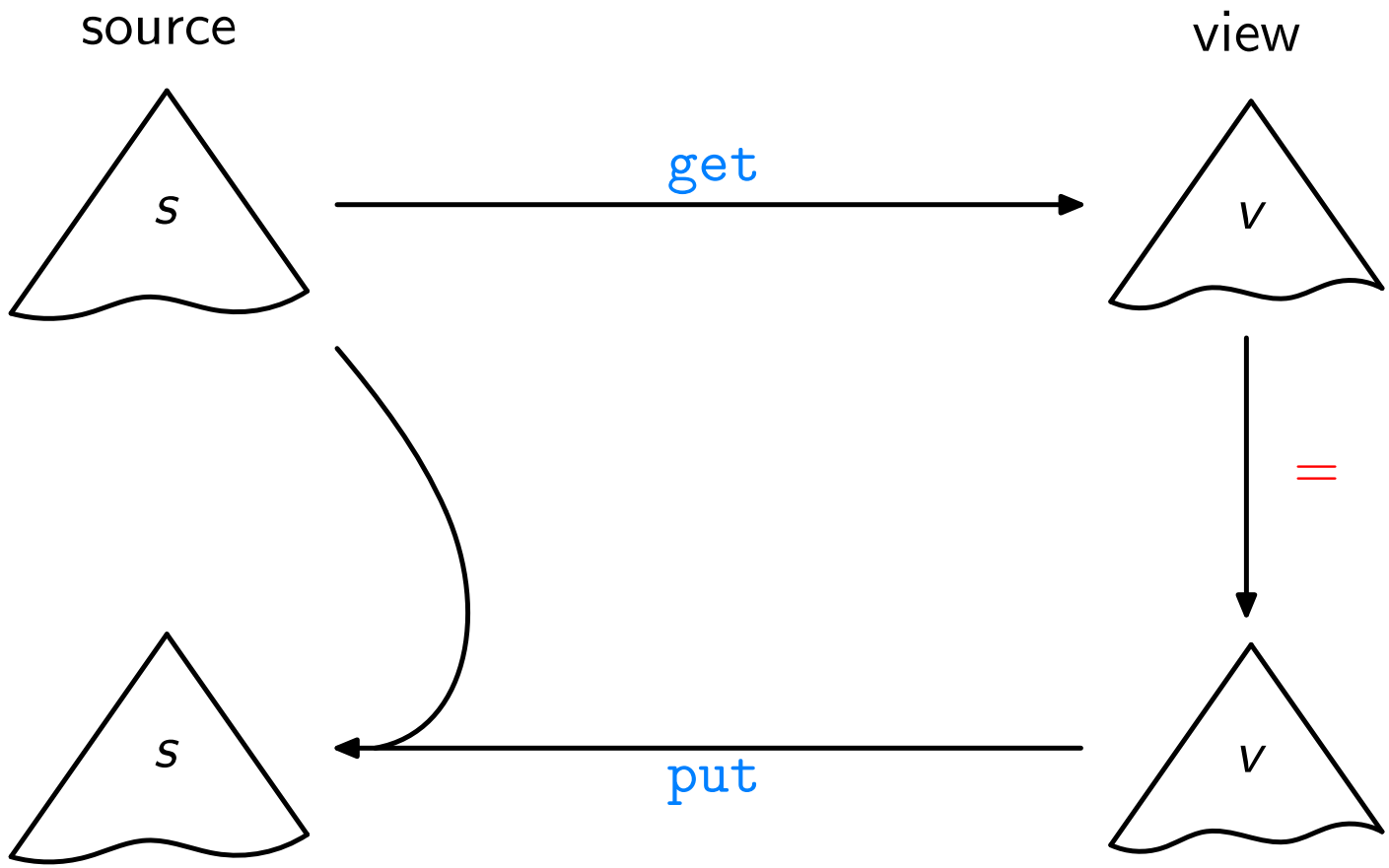


Bidirectional Transformation



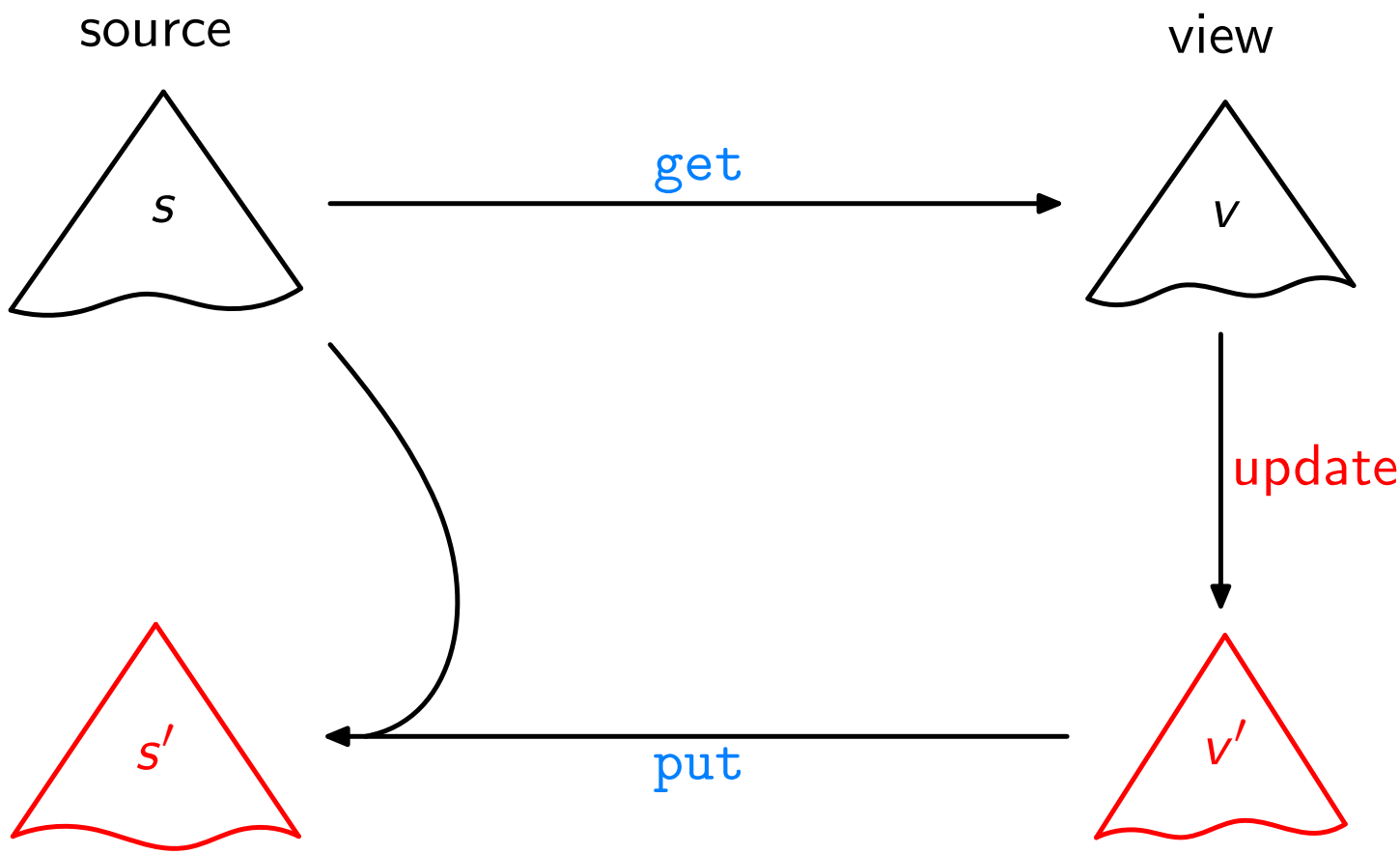
Acceptability / GetPut

Bidirectional Transformation



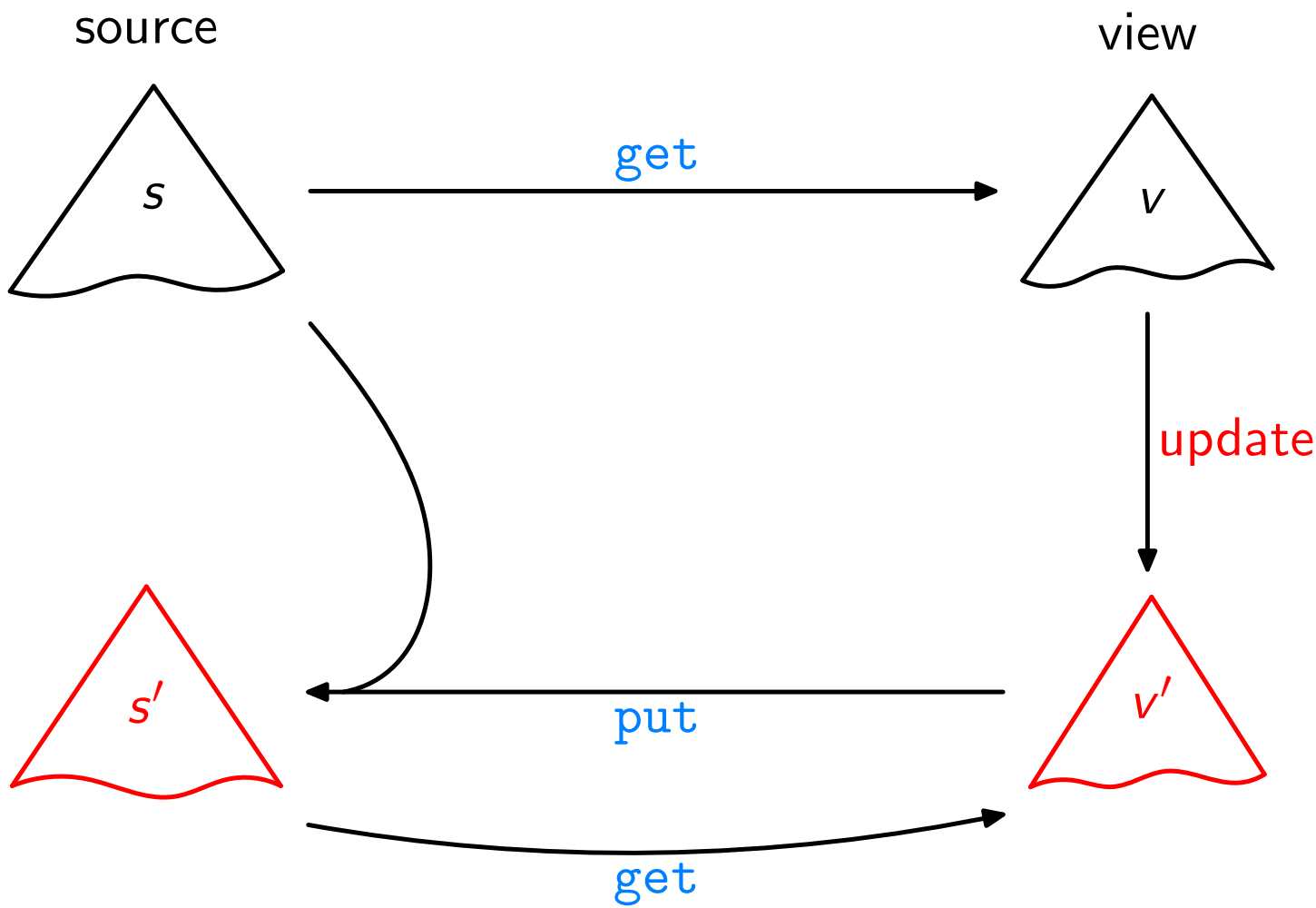
Acceptability / GetPut

Bidirectional Transformation



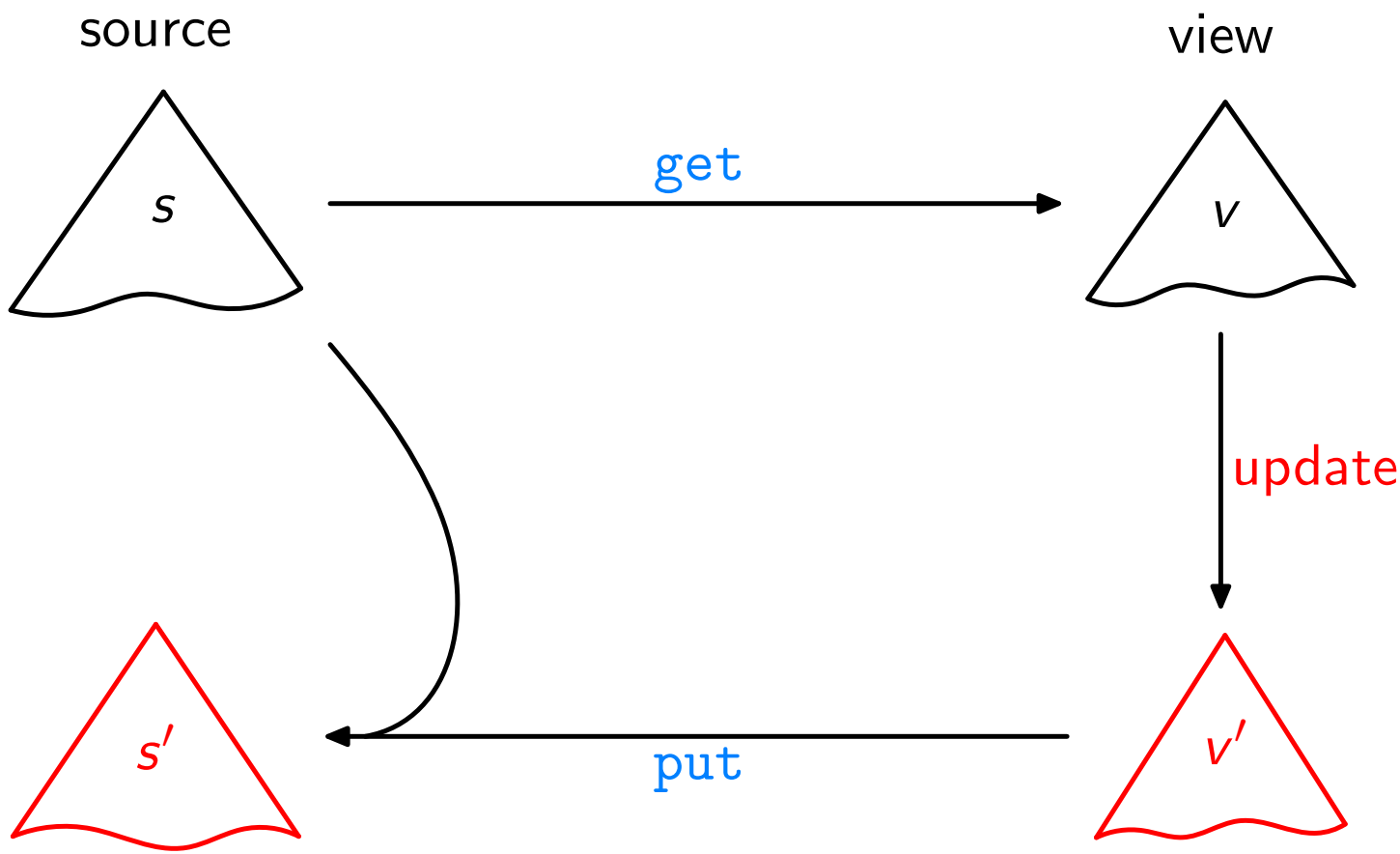
Consistency / PutGet

Bidirectional Transformation

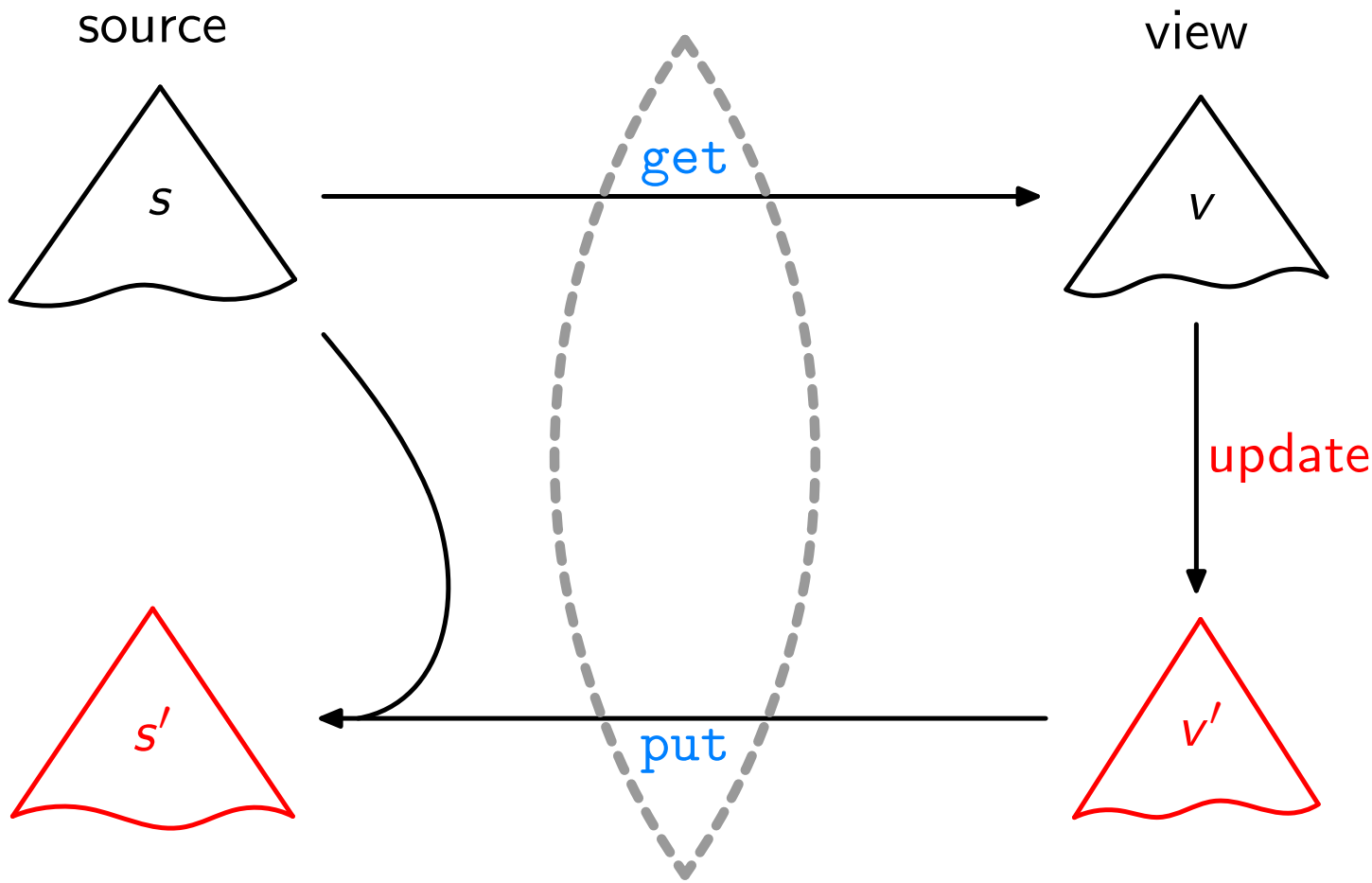


Consistency / PutGet

Bidirectional Transformation



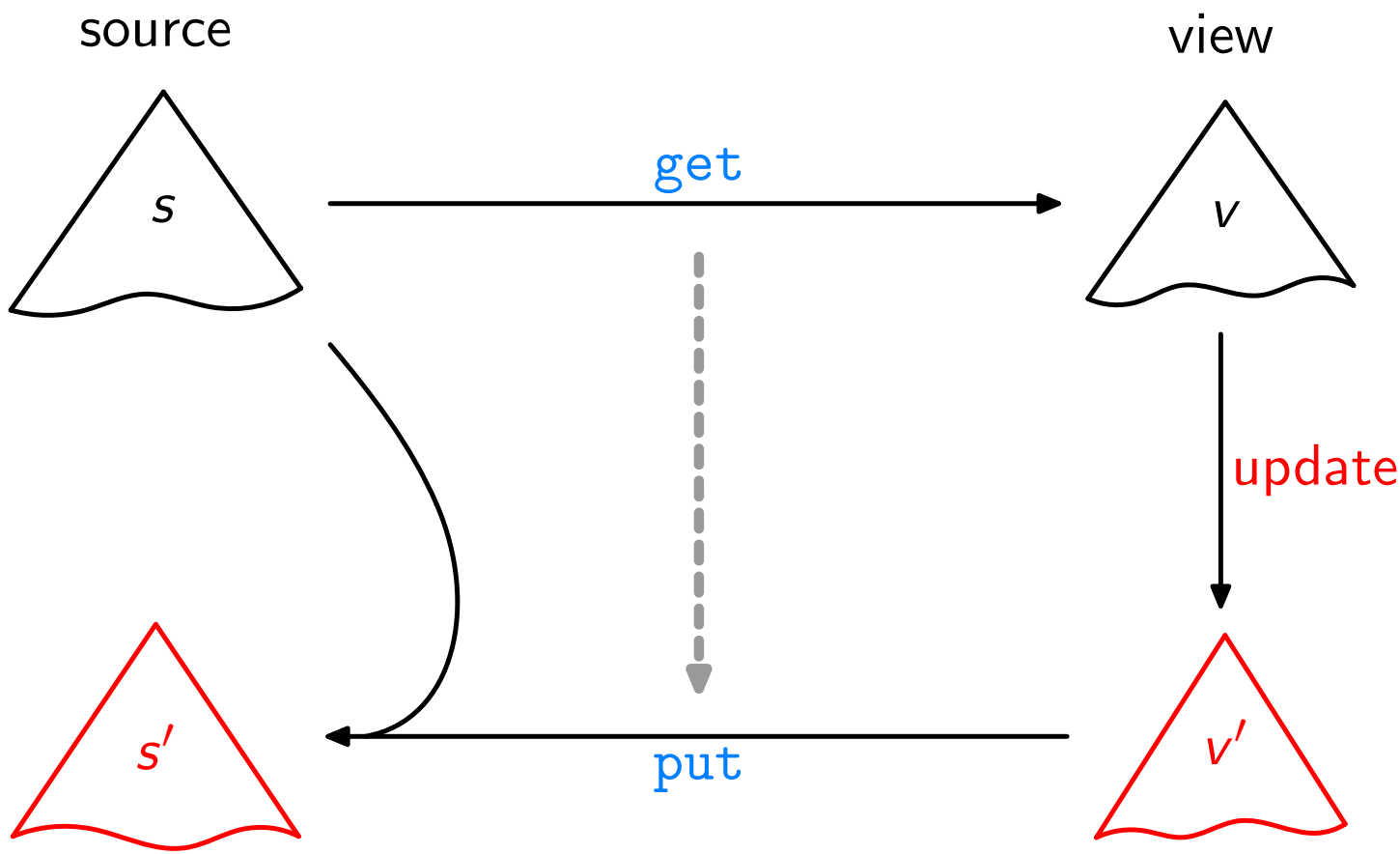
Bidirectional Transformation



Lenses, DSLs

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

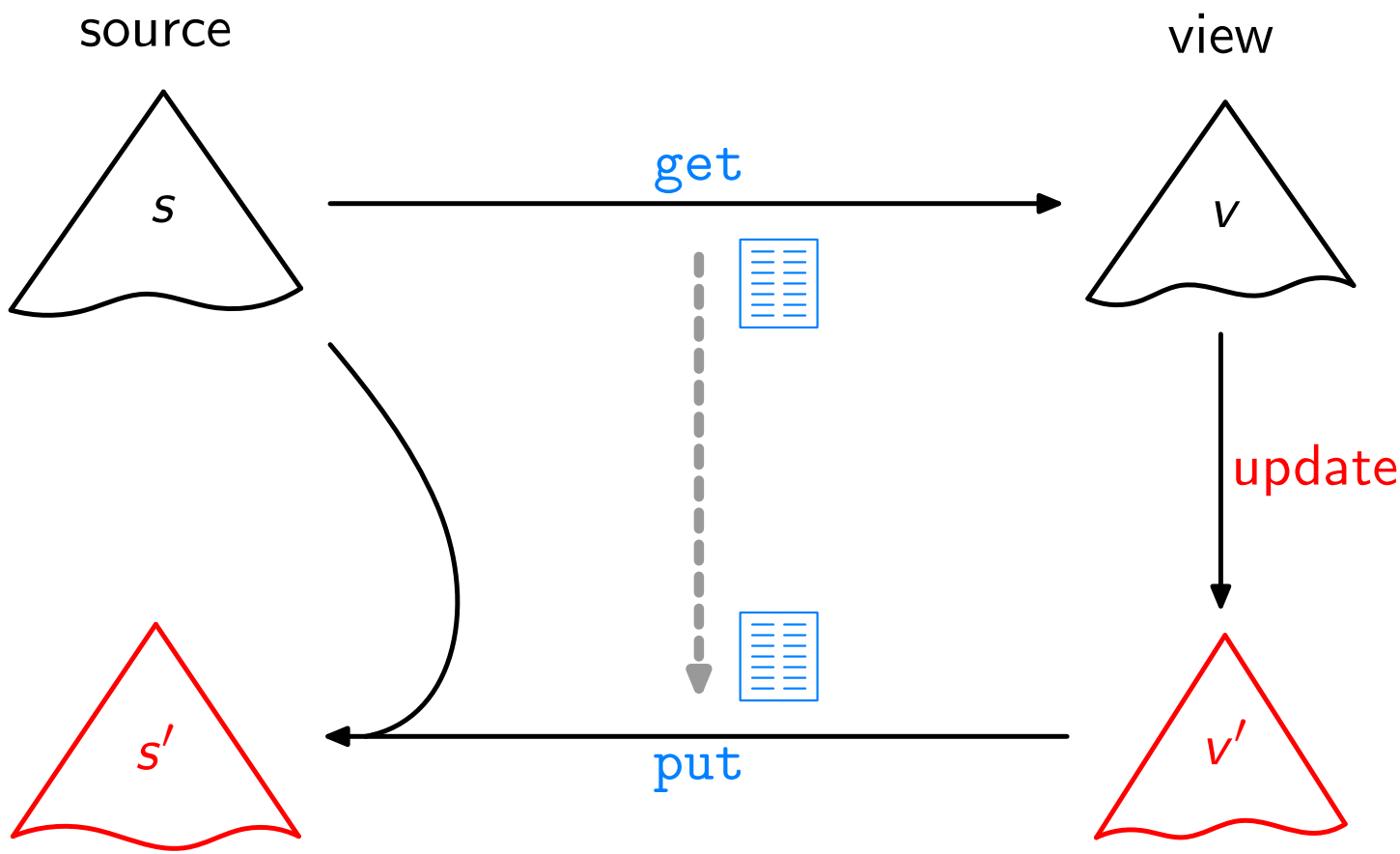
Bidirectional Transformation



Bidirectionalization

[Matsuda et al., ICFP'07]

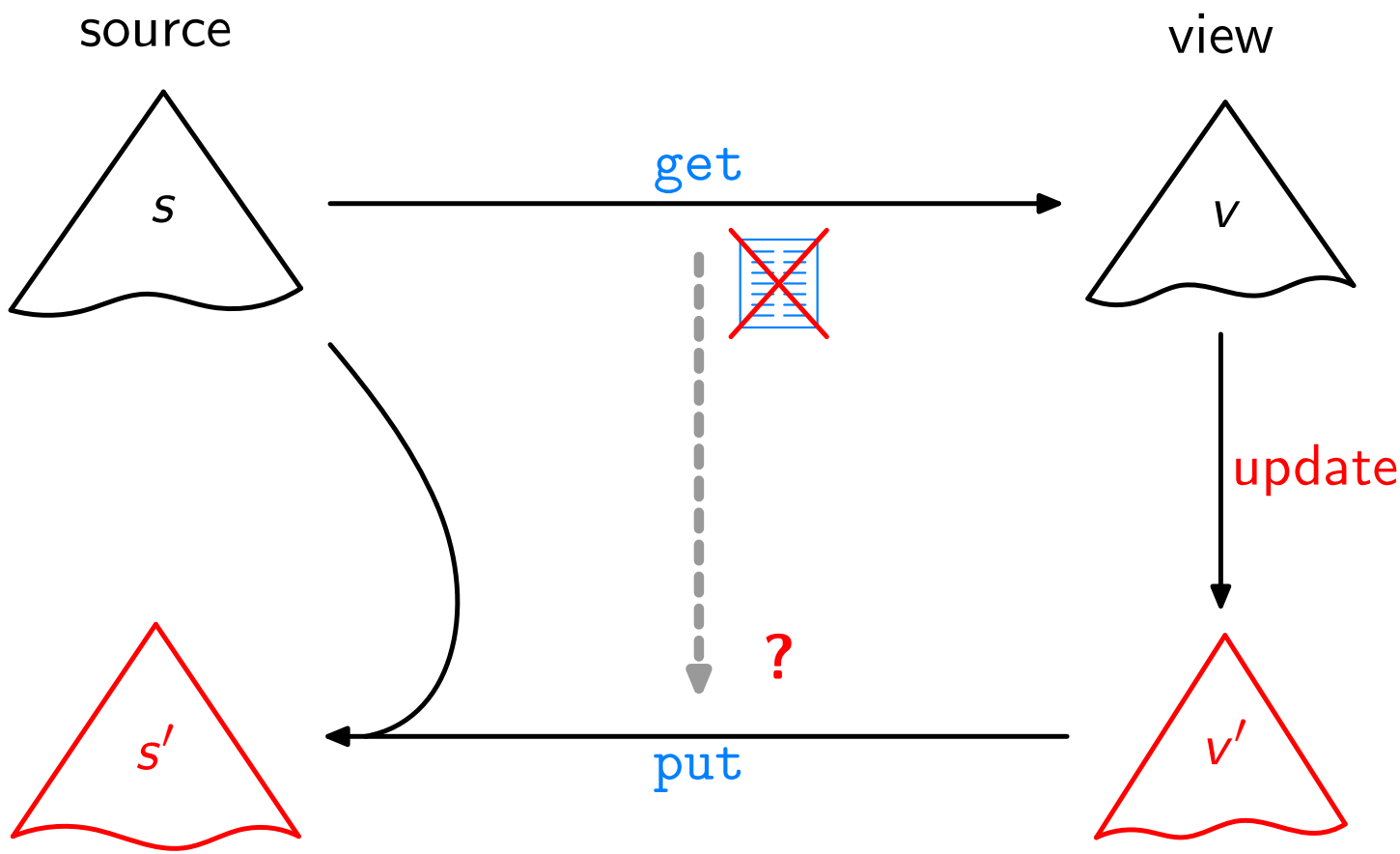
Bidirectional Transformation



(Syntactic) Bidirectionalization

[Matsuda et al., ICFP'07]

Bidirectional Transformation



This work

Semantic Bidirectionalization

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

Semantic Bidirectionalization

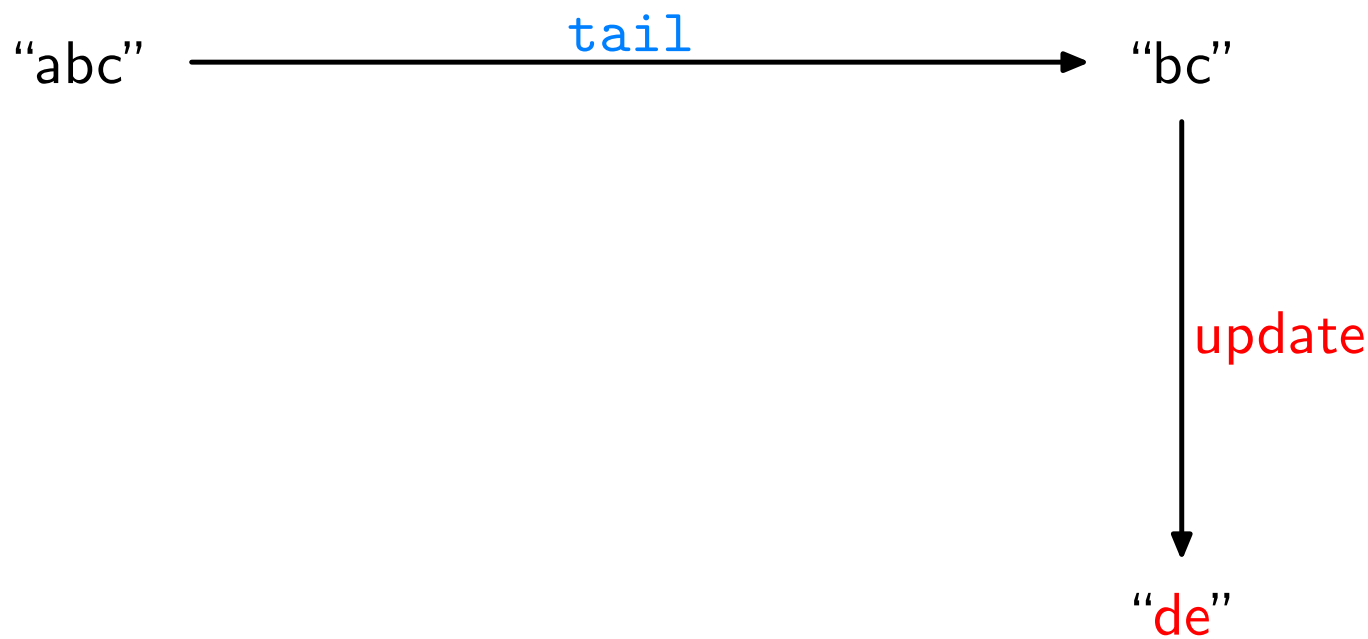
Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Example:

Semantic Bidirectionalization

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

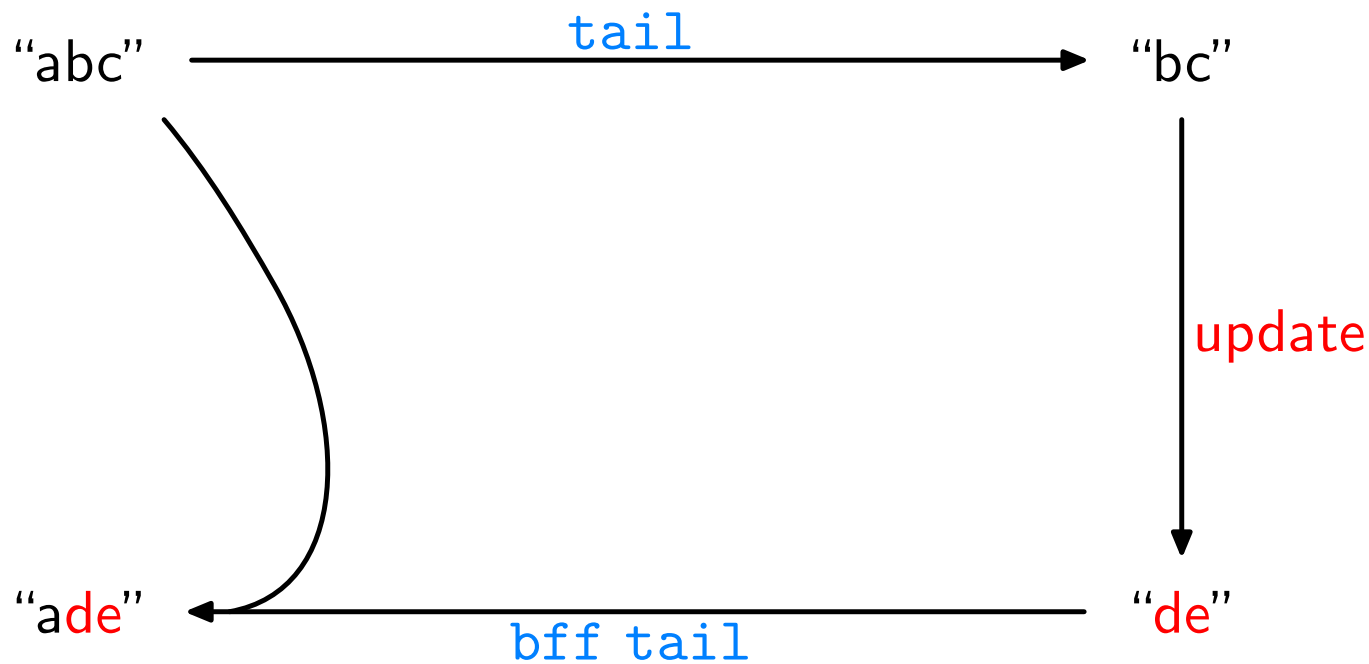
Example:



Semantic Bidirectionalization

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

Example:



Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

Like:

$$\texttt{get}\ [0..n] = \begin{cases} [1..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..0] & \text{if } \texttt{get} = \texttt{reverse} \\ [0..(\texttt{min}\ 4\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ \vdots & \end{cases}$$

Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

Like:

$$\texttt{get}\ [0..n] = \begin{cases} [1..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..0] & \text{if } \texttt{get} = \texttt{reverse} \\ [0..(\texttt{min}\ 4\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ \vdots & \end{cases}$$

Then transfer the gained insights to source lists other than $[0..n]$!

Enter Free Theorems [Wadler, FPCA'89]

For every

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

we have

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

for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Enter Free Theorems [Wadler, FPCA'89]

For every

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

we have

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

for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\ = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

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

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

Enter Free Theorems [Wadler, FPCA'89]

For every

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

we have

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

for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

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

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

Enter Free Theorems [Wadler, FPCA'89]

For every

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

we have

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

for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

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

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

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

Enter Free Theorems [Wadler, FPCA'89]

For every

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

we have

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

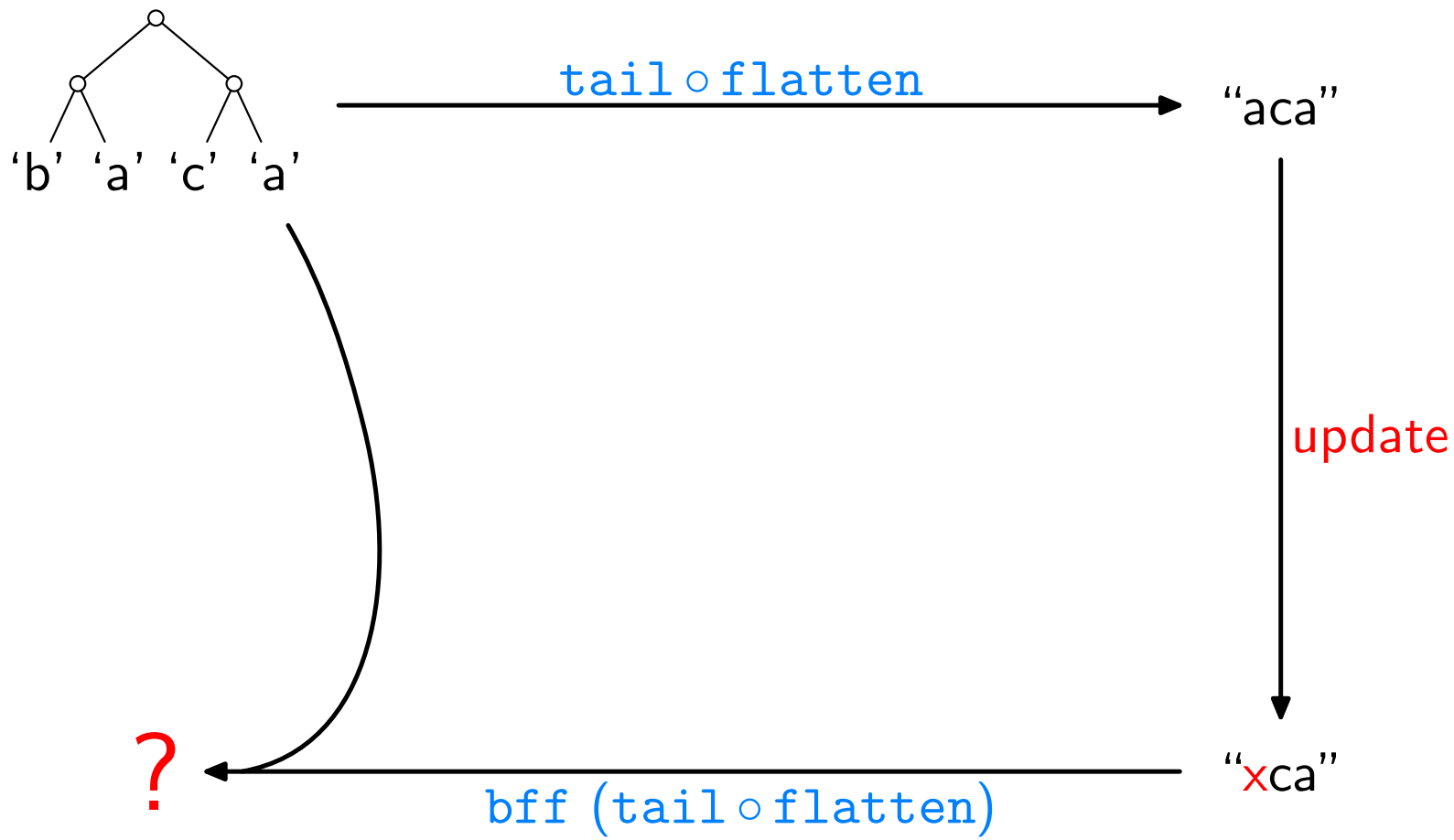
for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

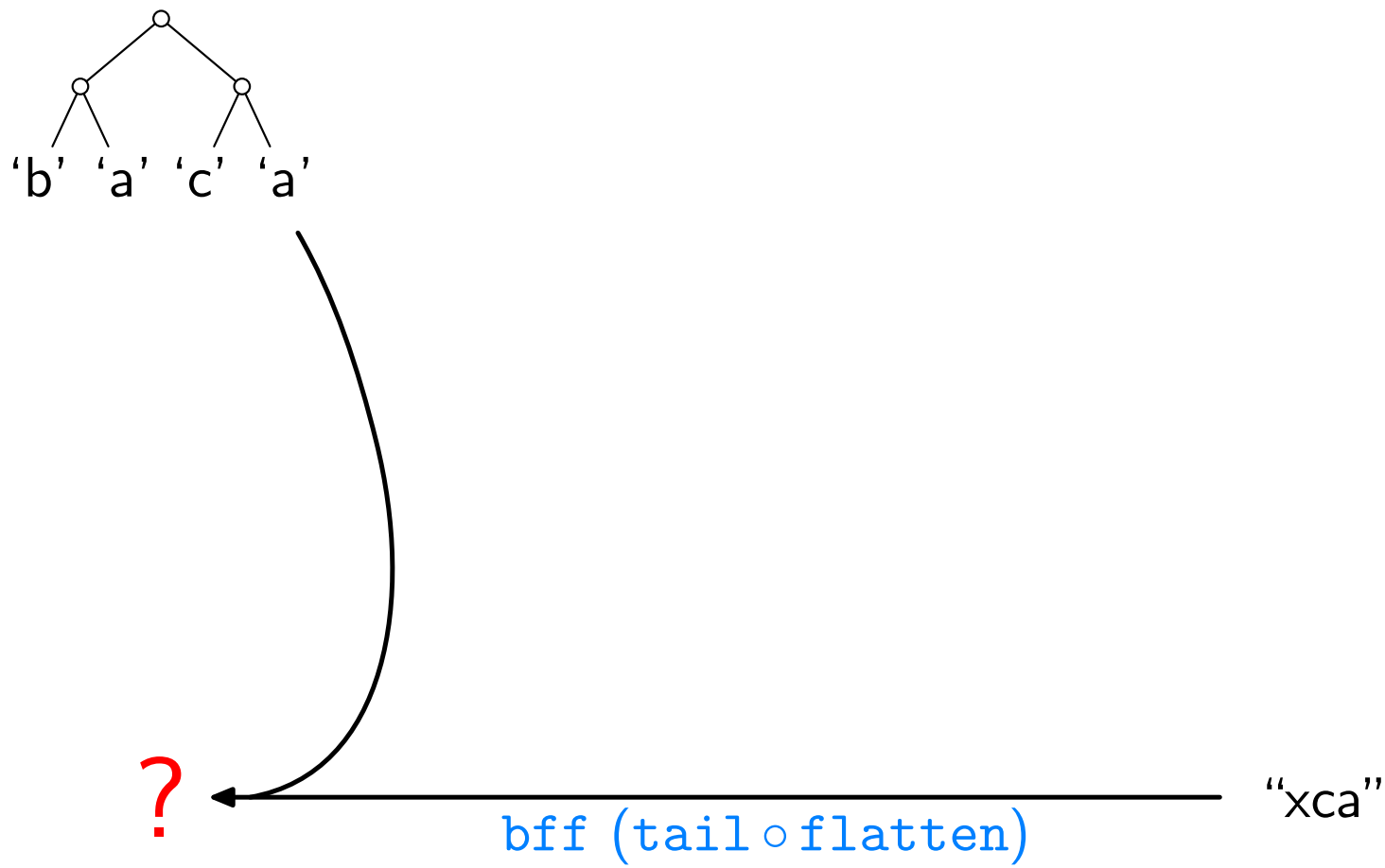
Given an arbitrary list s of length $n + 1$,

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

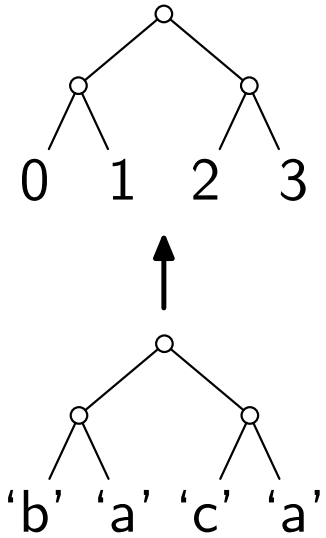
The Resulting Bidirectionalization Scheme by Example



The Resulting Bidirectionalization Scheme by Example

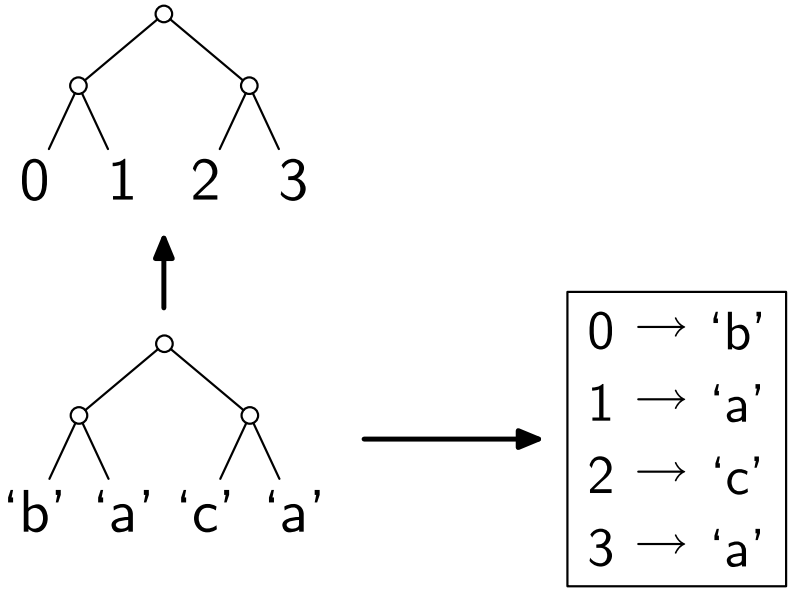


The Resulting Bidirectionalization Scheme by Example



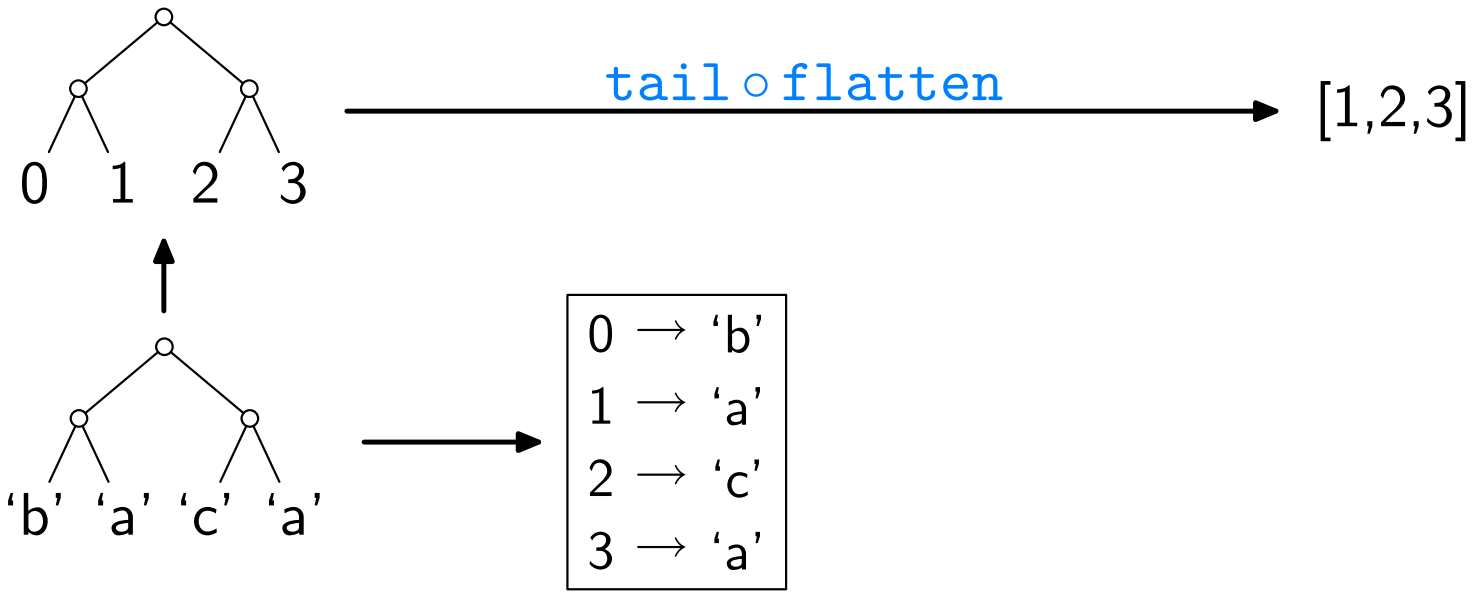
“xca”

The Resulting Bidirectionalization Scheme by Example



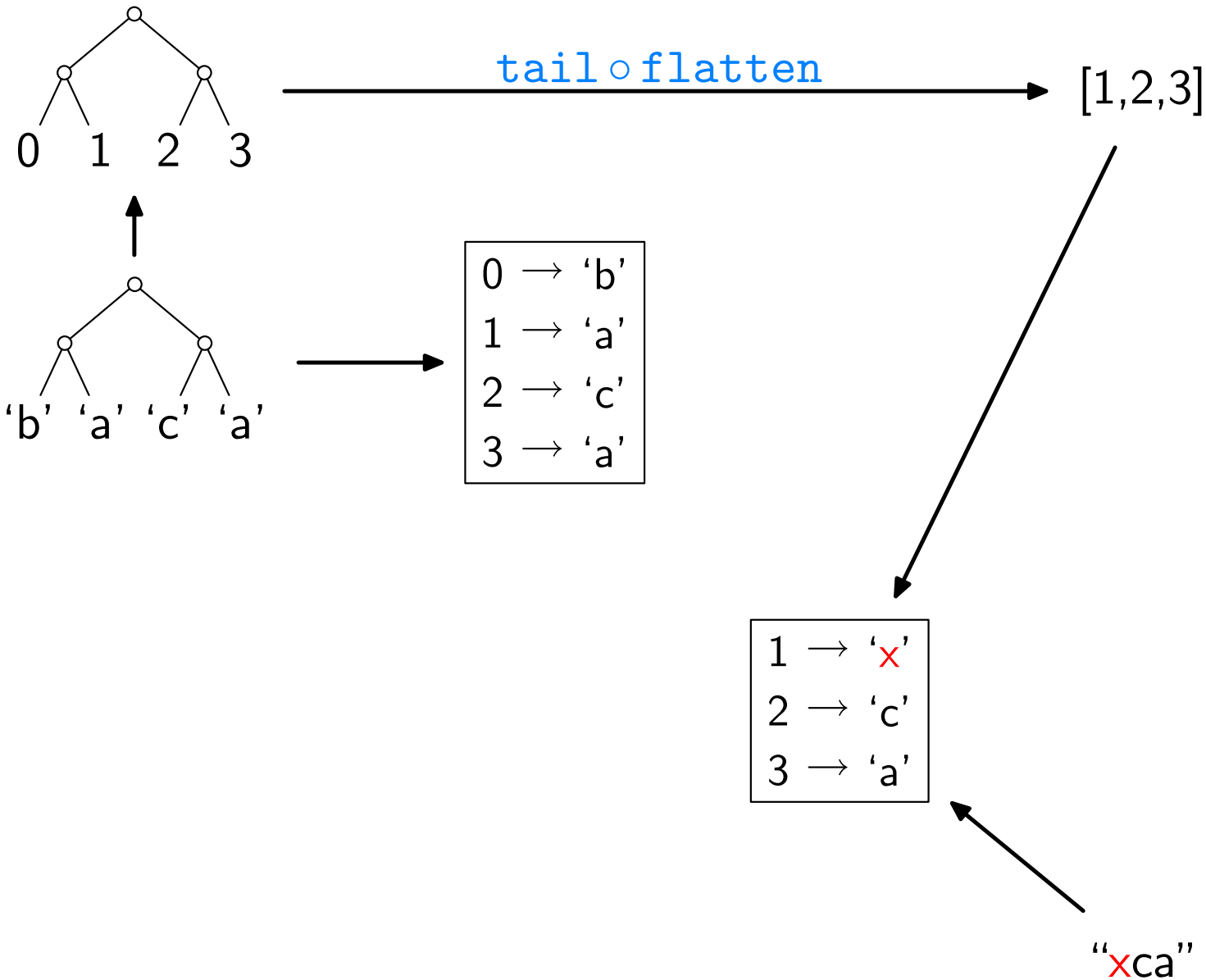
"xca"

The Resulting Bidirectionalization Scheme by Example

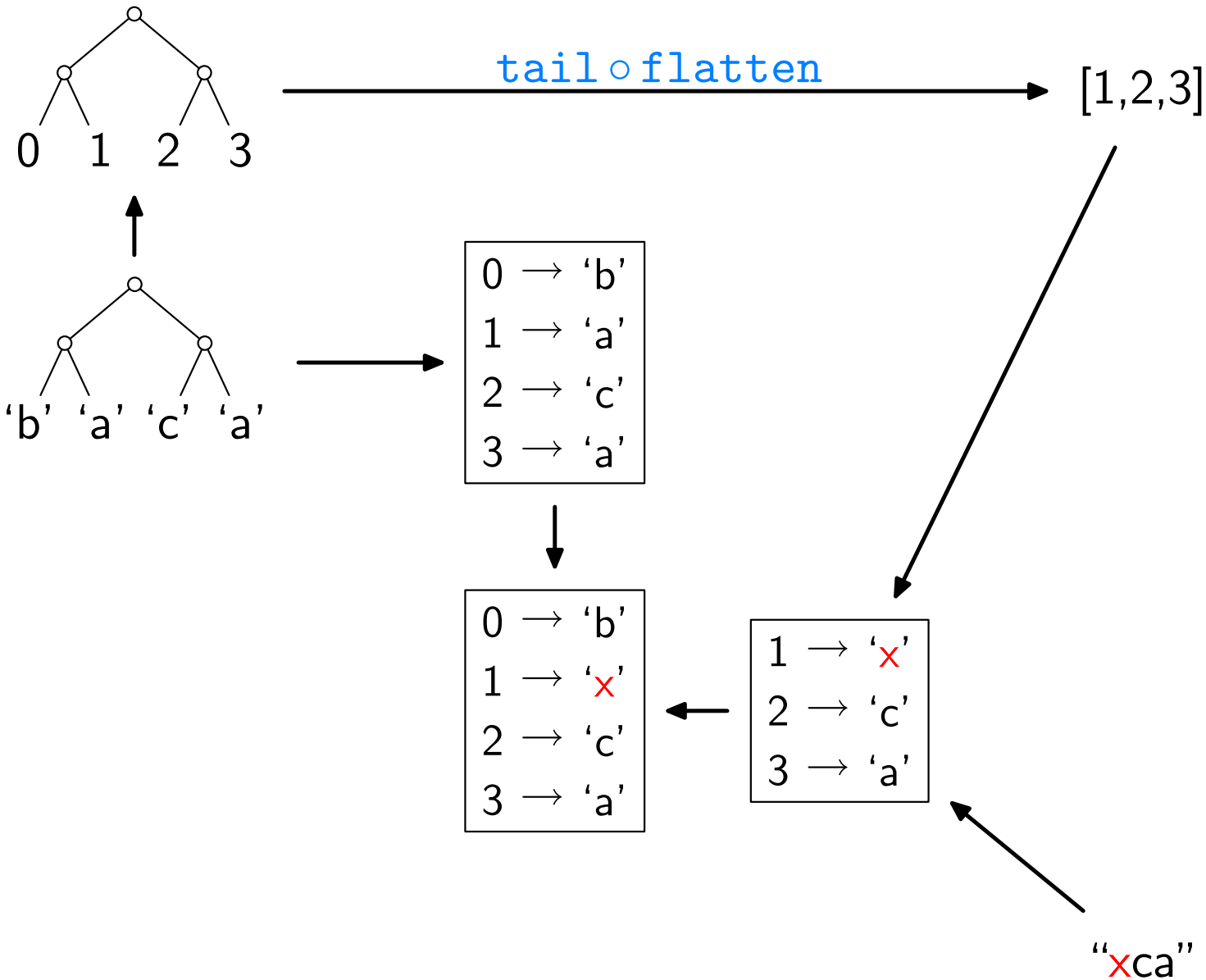


"xca"

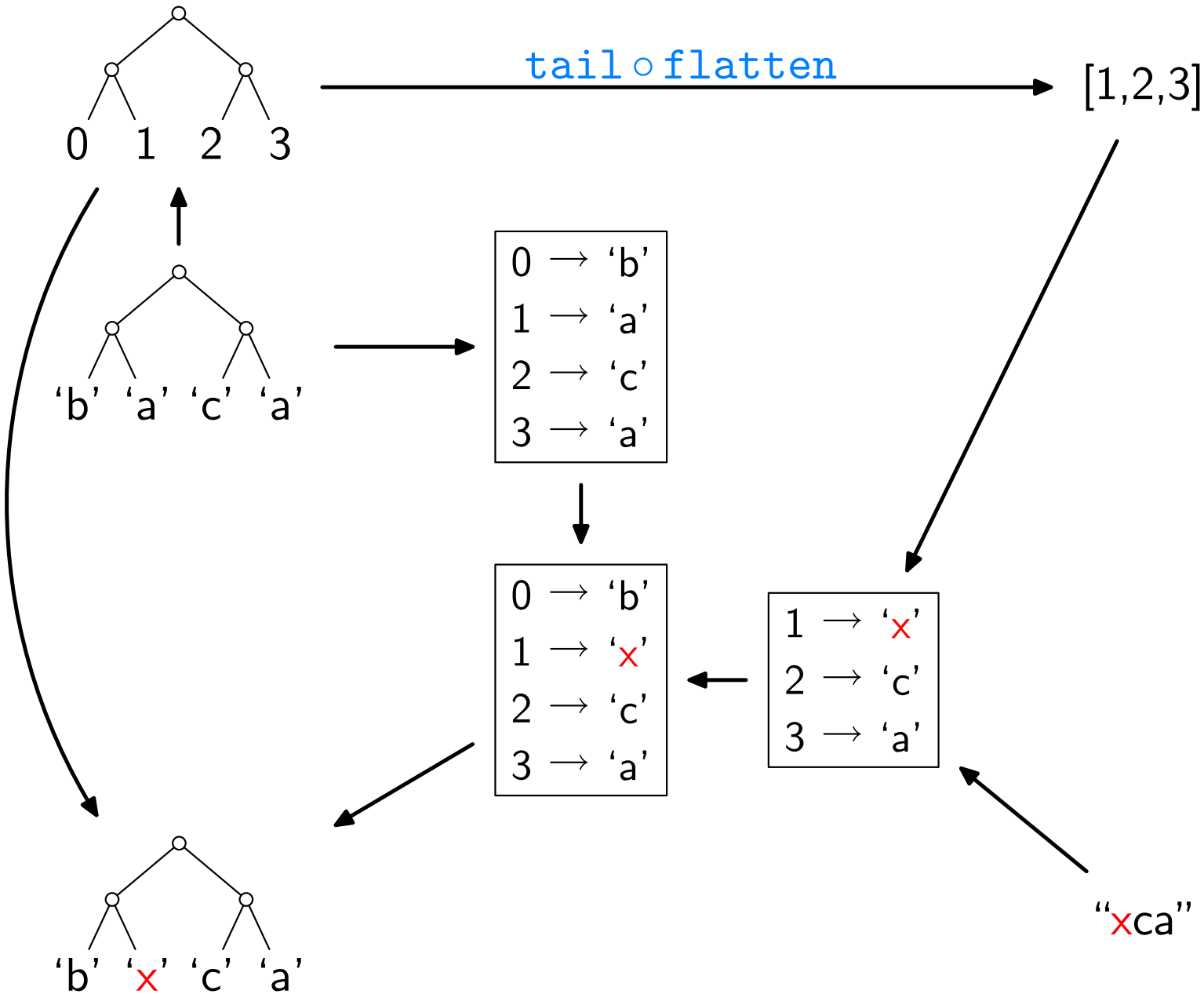
The Resulting Bidirectionalization Scheme by Example



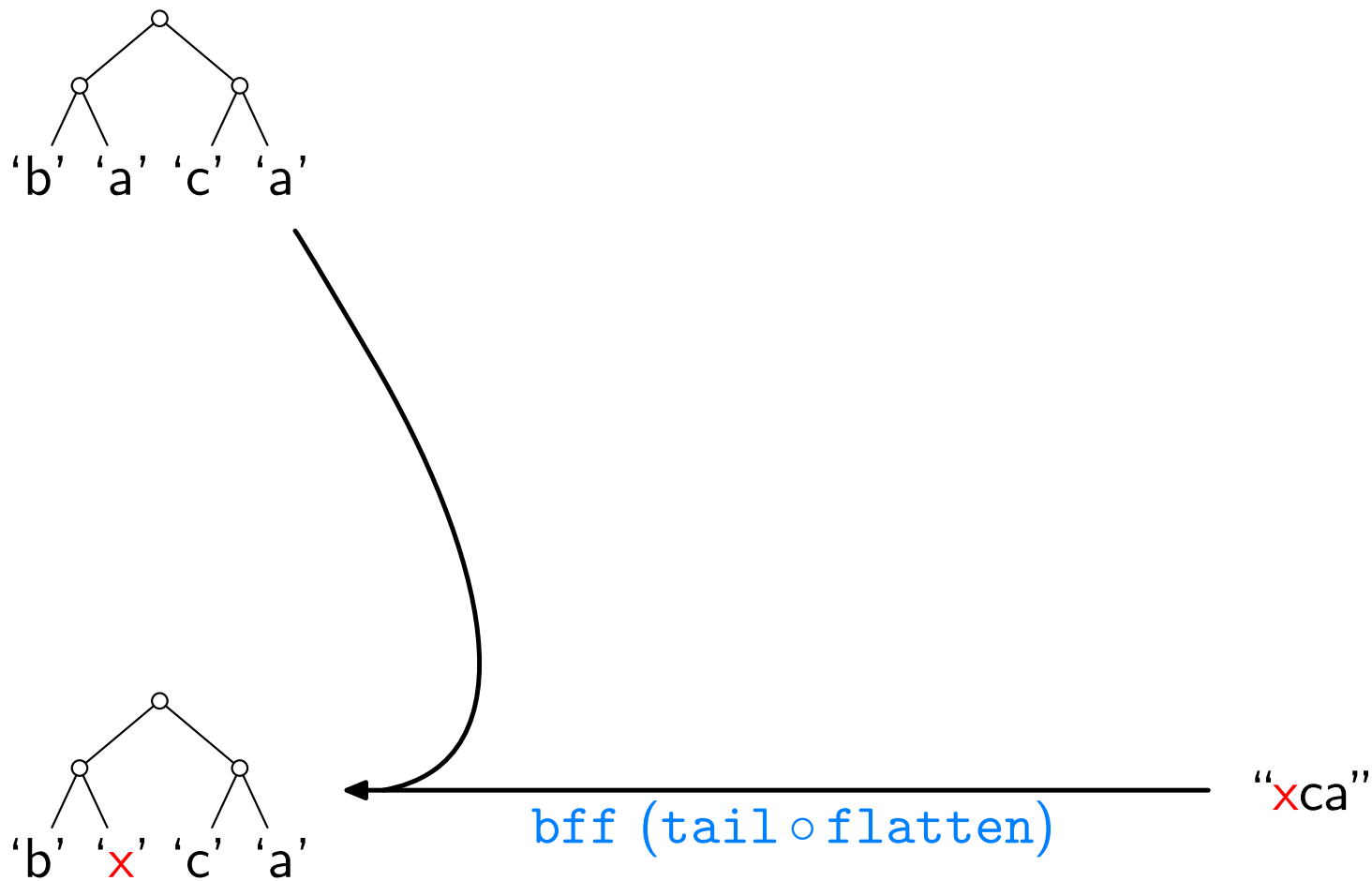
The Resulting Bidirectionalization Scheme by Example



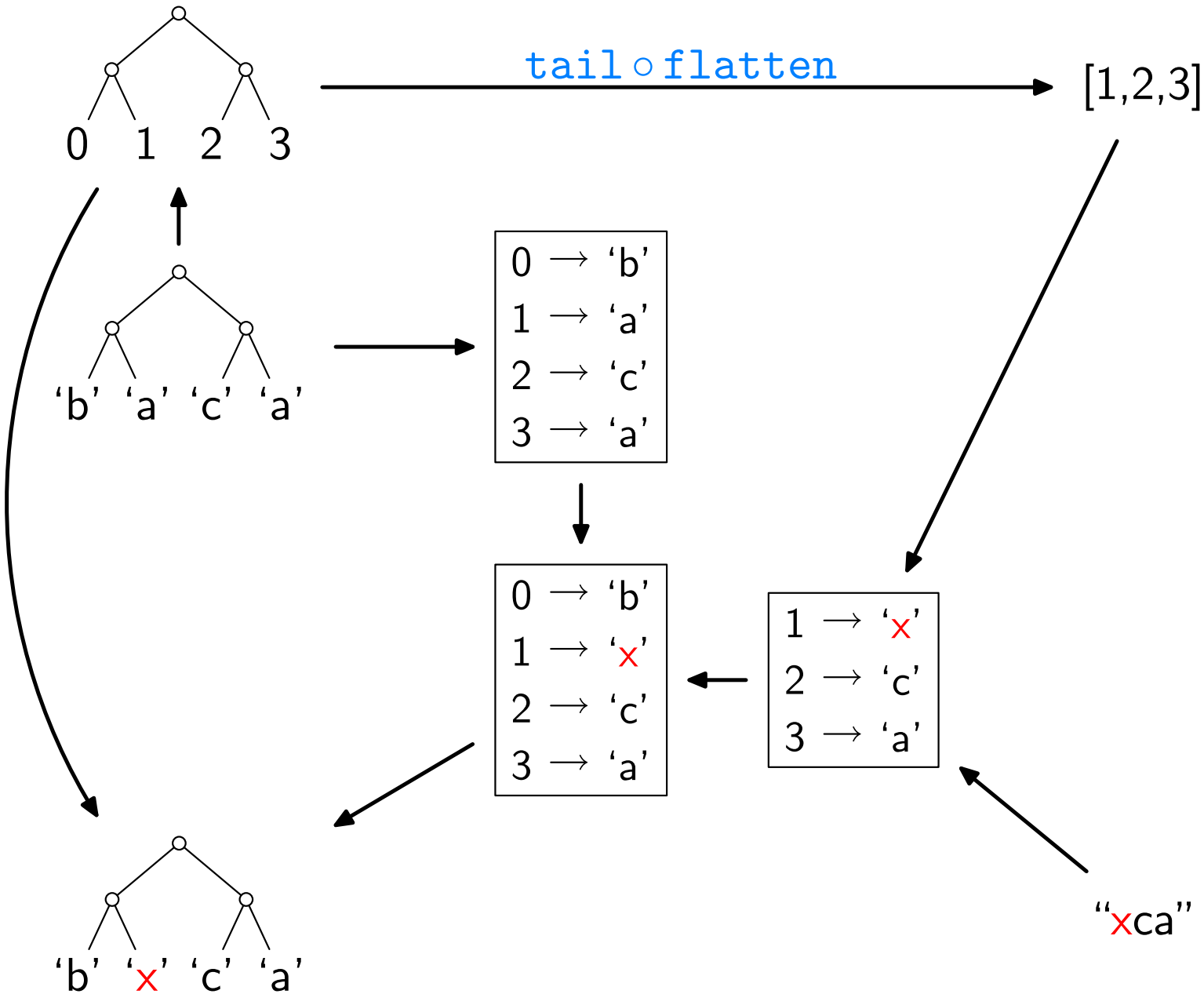
The Resulting Bidirectionalization Scheme by Example



The Resulting Bidirectionalization Scheme by Example



The Resulting Bidirectionalization Scheme by Example



The Implementation (here: lists only, inefficient version)

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

$$
\begin{aligned}
\texttt{assoc}\ [\,]\ \ \ [\,] &= [\,] \\
\texttt{assoc}\ (i : is)\ (b : bs) &= \textbf{let}\ m = \texttt{assoc}\ is\ bs \\
&\quad \textbf{in}\ \ \textbf{case}\ \texttt{lookup}\ i\ m\ \textbf{of} \\
&\qquad \text{Nothing} \to (i, b) : m \\
&\qquad \text{Just}\ c \mid b == c \to m
\end{aligned}
$$

The Implementation (here: lists only, inefficient version)

```
bff get s v' = let n = (length s) - 1
                   t = [0..n]
                   g = zip t s
                   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
```

- for the actual (slightly more elaborate) code, see the paper
- try out: `http://linux.tcs.inf.tu-dresden.de/~bff`

What Else?

In the paper:

- treatment of equality and ordering constraints
- full proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

What Else?

In the paper:

- treatment of equality and ordering constraints
- full proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Pros of the approach:

- great fun
- liberation from syntactic constraints
- very lightweight, easy access to bidirectionality

What Else?

In the paper:

- treatment of equality and ordering constraints
- full proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Pros of the approach:

- great fun
- liberation from syntactic constraints
- very lightweight, easy access to bidirectionality

Cons of the approach:

- efficiency still leaves room for improvement
- partiality, e.g., rejection of shape-affecting updates so far

References

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.

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