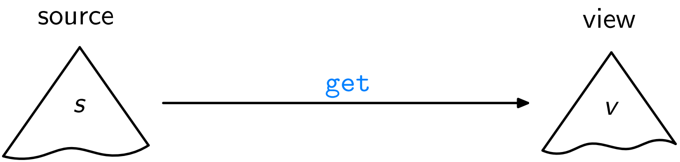
Semantic Bidirectionalisation

Janis Voigtländer

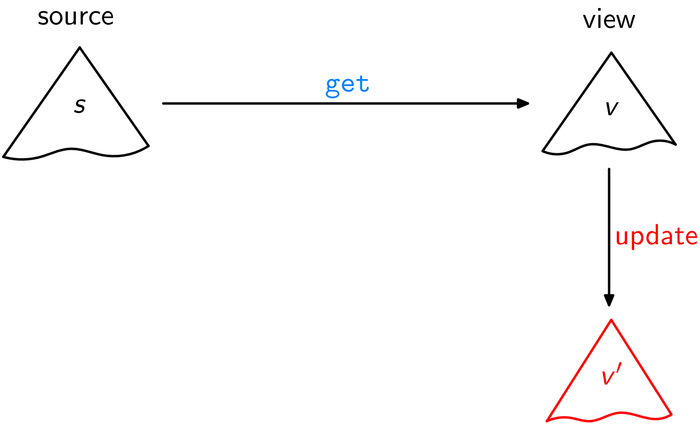
Technische Universität Dresden

April 21st, 2009

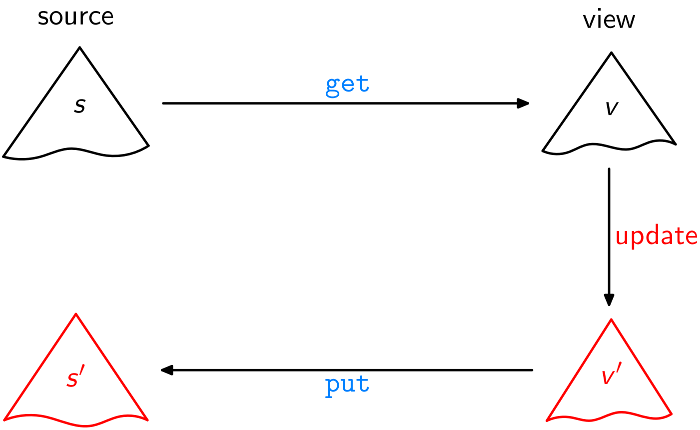
Bidirectional Transformation



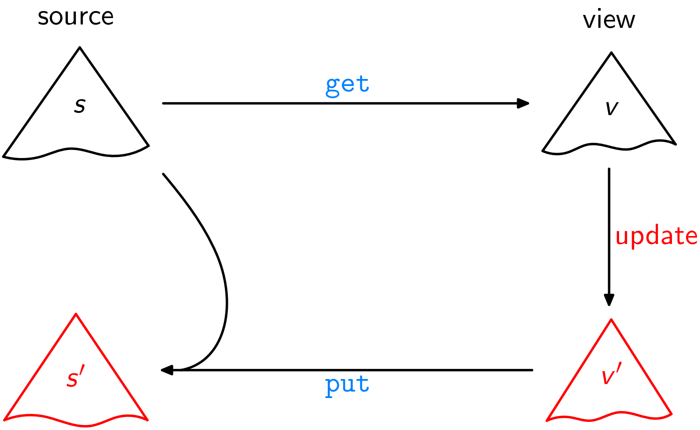
Bidirectional Transformation



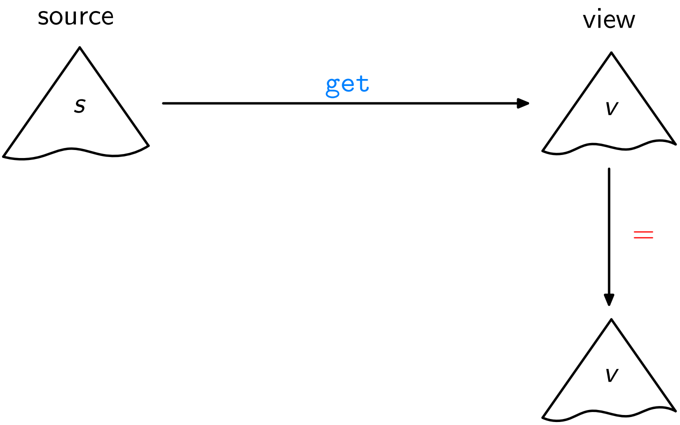
Bidirectional Transformation



Bidirectional Transformation

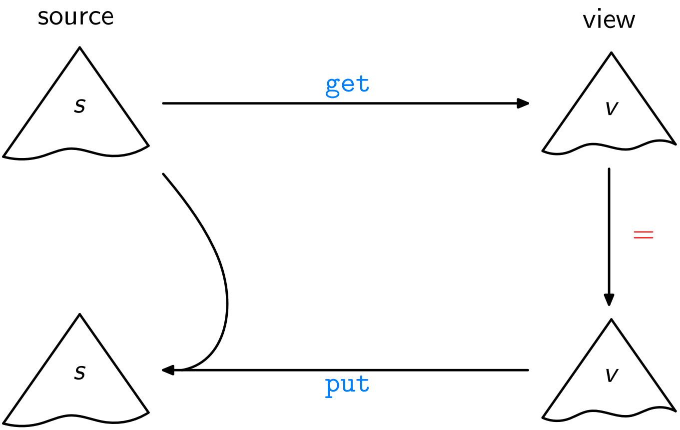


Bidirectional Transformation



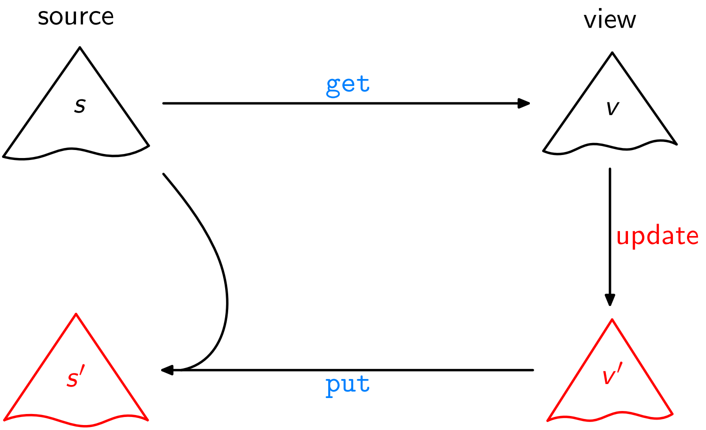
Acceptability / GetPut

Bidirectional Transformation



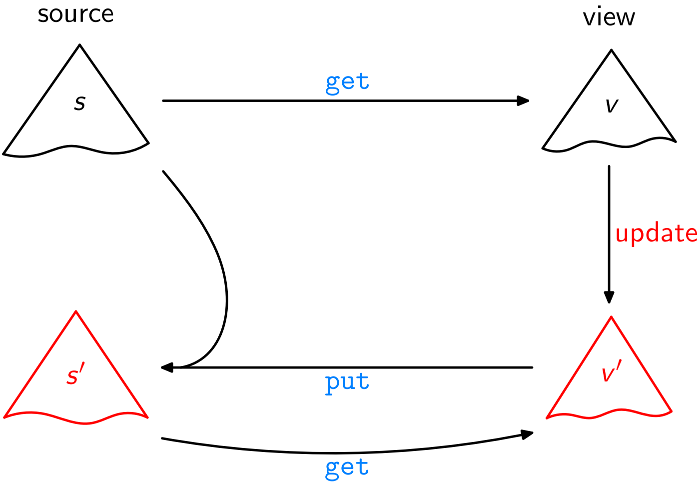
Acceptability / GetPut

Bidirectional Transformation



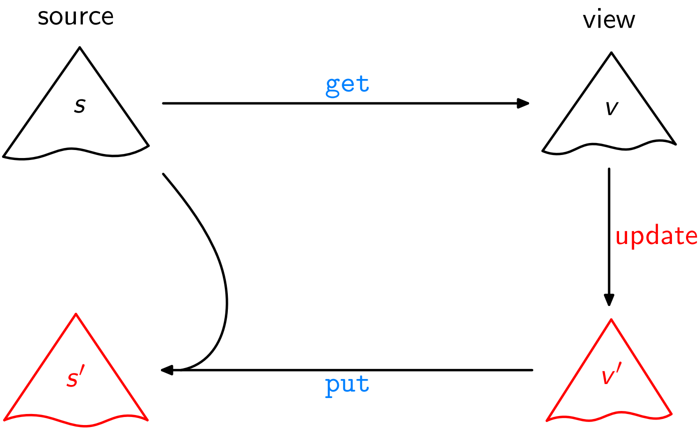
Consistency / PutGet

Bidirectional Transformation

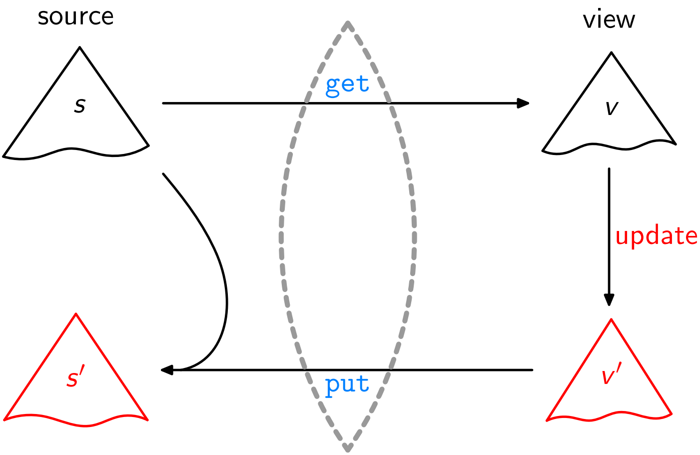


Consistency / PutGet

Bidirectional Transformation



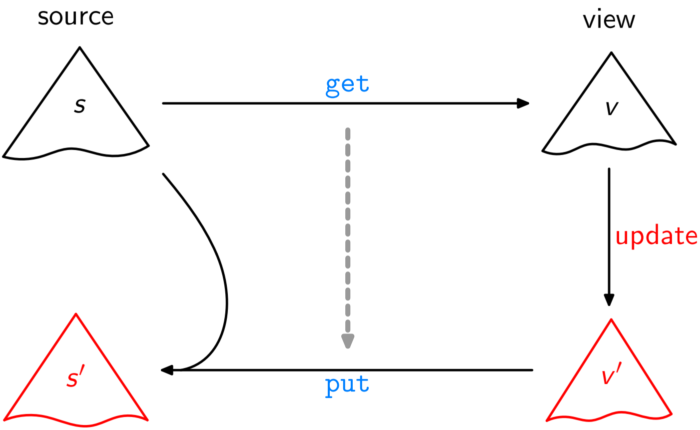
Bidirectional Transformation



Lenses, DSLs

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

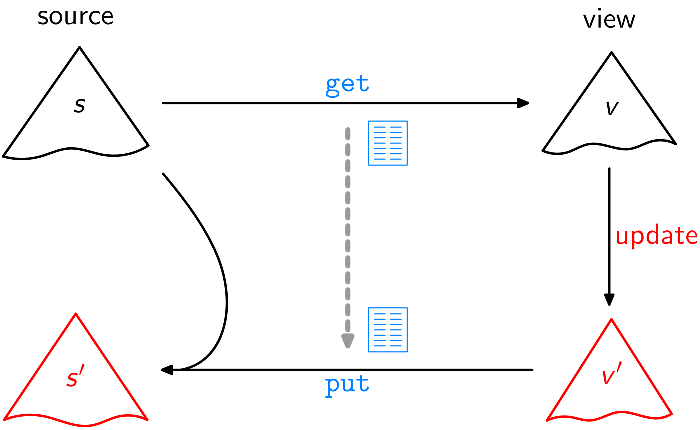
Bidirectional Transformation



Bidirectionalisation

[Matsuda et al., ICFP'07]

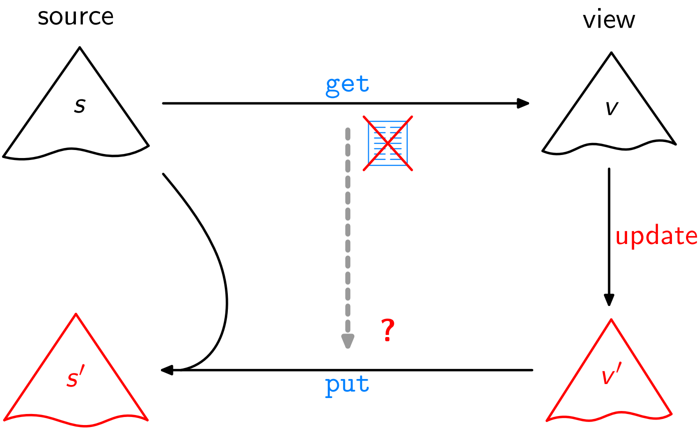
Bidirectional Transformation



Syntactic Bidirectionalisation

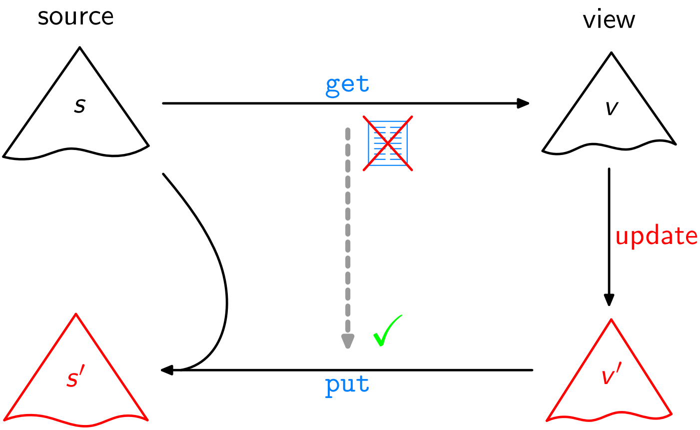
[Matsuda et al., ICFP'07]

Bidirectional Transformation



Semantic Bidirectionalisation

Bidirectional Transformation



Semantic Bidirectionalisation

[V., POPL'09]

Semantic Bidirectionalisation

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

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

[1] "Bidirectionalisation for free!"

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

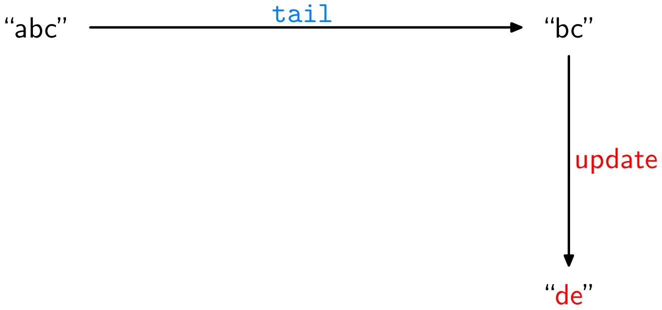
Examples:

[1] “Bidirectionalisation for free!”

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

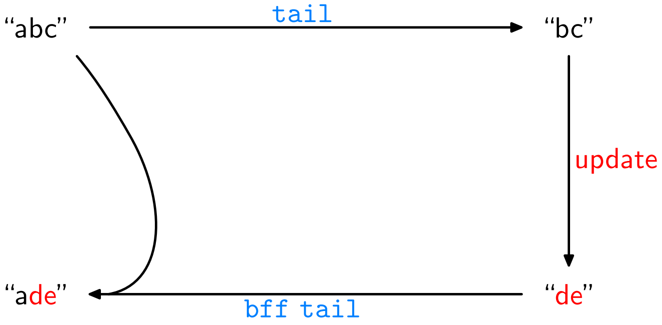


[1] "Bidirectionalisation for free!"

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

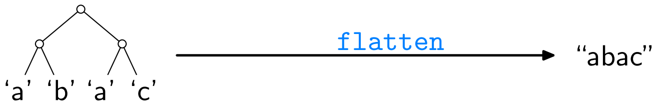


[1] "Bidirectionalisation for free!"

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:



[1] "Bidirectionalisation for free!"

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

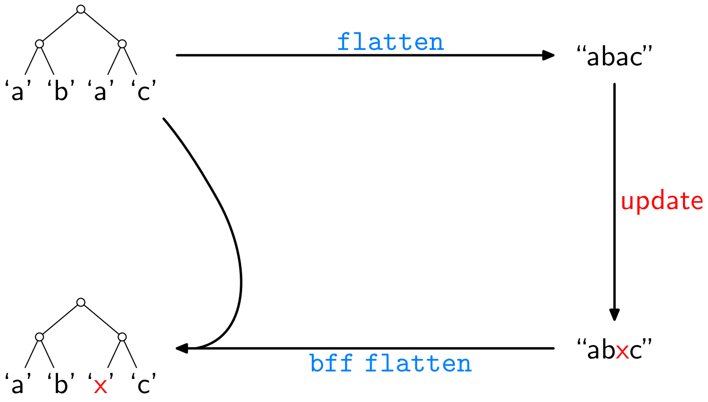
Examples:

[1] "Bidirectionalisation for free!"

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

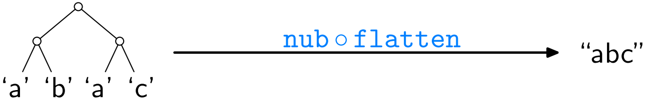


[1] "Bidirectionalisation for free!"

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:



[1] “Bidirectionalisation for free!”

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

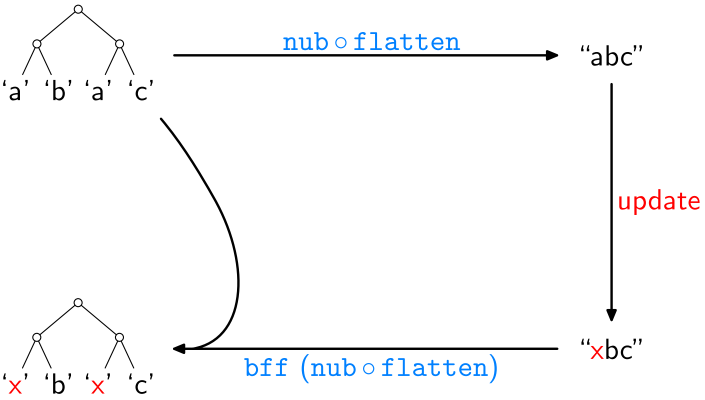
Examples:

[1] “Bidirectionalisation for free!”

Semantic Bidirectionalisation

Aim: Write a higher-order function `bff`[1] such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:



[1] "Bidirectionalisation for free!"

Analysing Specific Instances

Assume we are given some

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

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

Analysing Specific Instances

Assume we are given some

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

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

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

Analysing Specific Instances

Assume we are given some

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

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

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

Like:

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

Analysing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyse 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 $[0..n]$!

Using a Free Theorem [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

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

Using a Free Theorem [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 $l = [0..n]$, $f = (s\,!!)$, leading to:

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

Using a Free Theorem [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 $l = [0..n]$, $f = (s\,!!)$, leading to:

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

Using a Free Theorem [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 $n + 1$,

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

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

Using a Free Theorem [Wadler, FPCA'89]

For every

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

we have

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

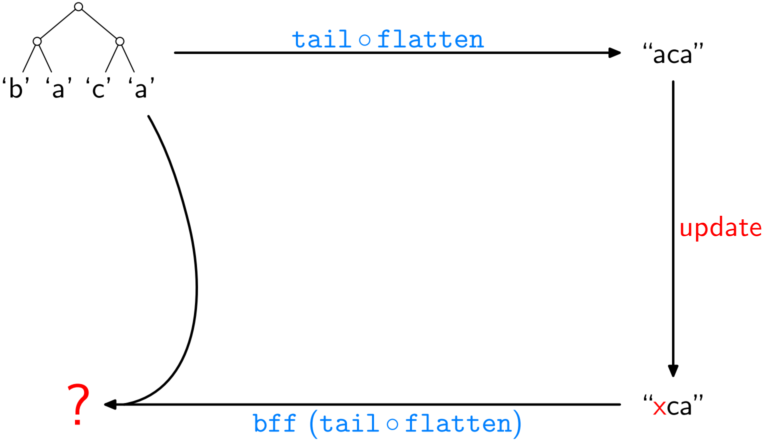
for arbitrary f and l, where

$$\begin{array}{l} \texttt{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ \texttt{map}\ f\ [\,] \qquad\quad = [\,] \\ \texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as) \end{array}$$

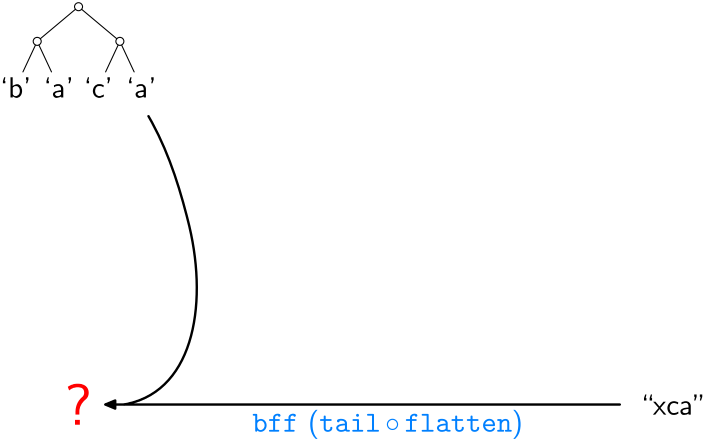
Given an arbitrary list s of length $n + 1$,

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

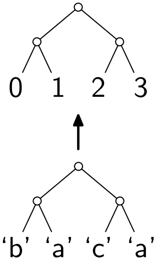
The Resulting Bidirectionalisation Scheme by Example



The Resulting Bidirectionalisation Scheme by Example

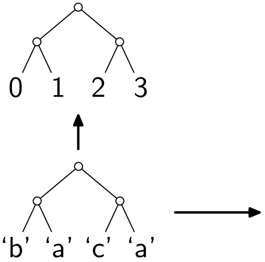


The Resulting Bidirectionalisation Scheme by Example



"xca"

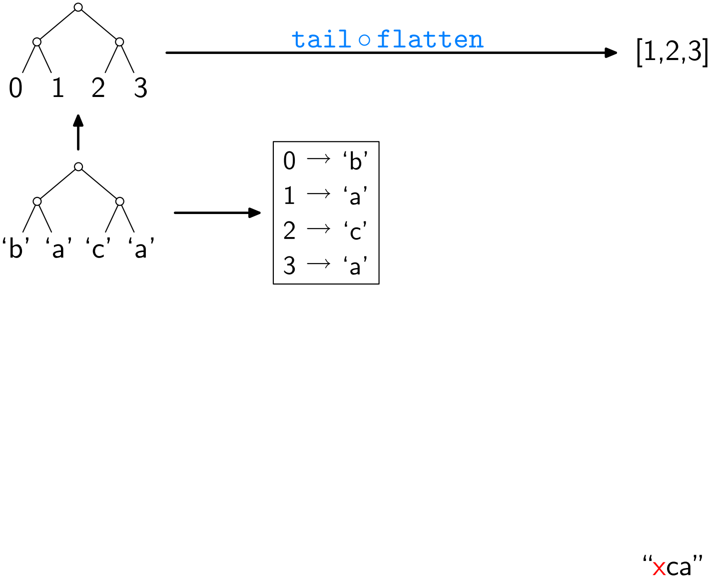
The Resulting Bidirectionalisation Scheme by Example



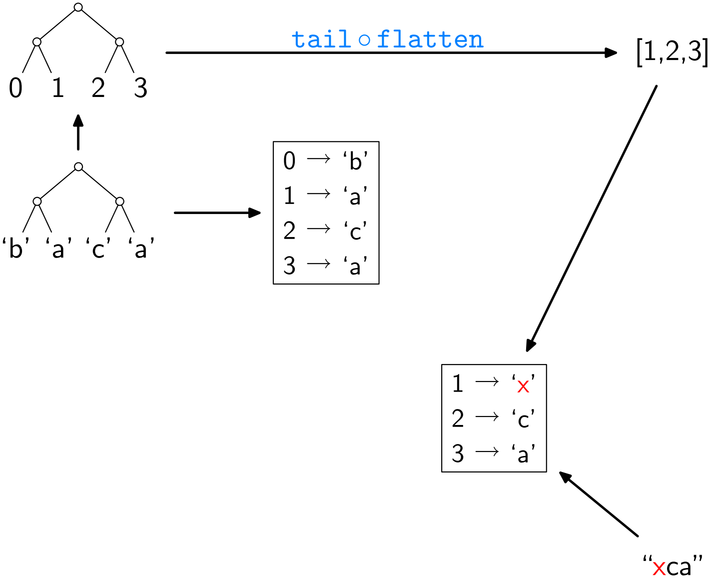
0	$\to$	'b'
1	$\to$	'a'
2	$\to$	'c'
3	$\to$	'a'

"xca"

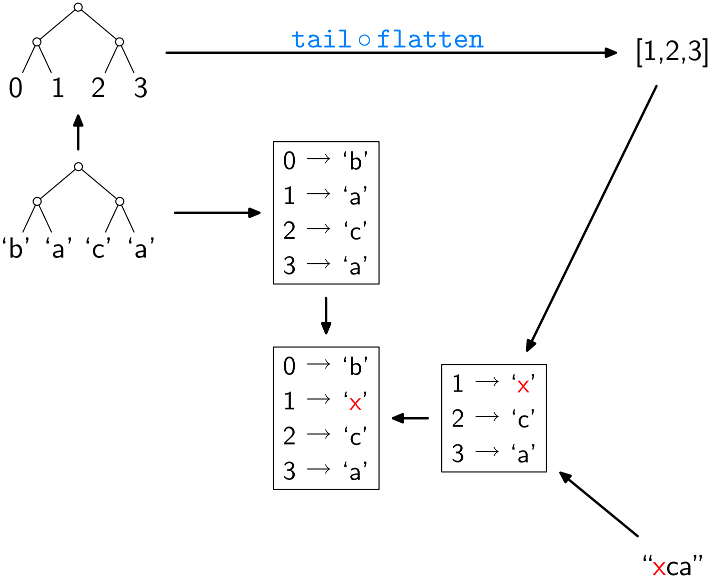
The Resulting Bidirectionalisation Scheme by Example



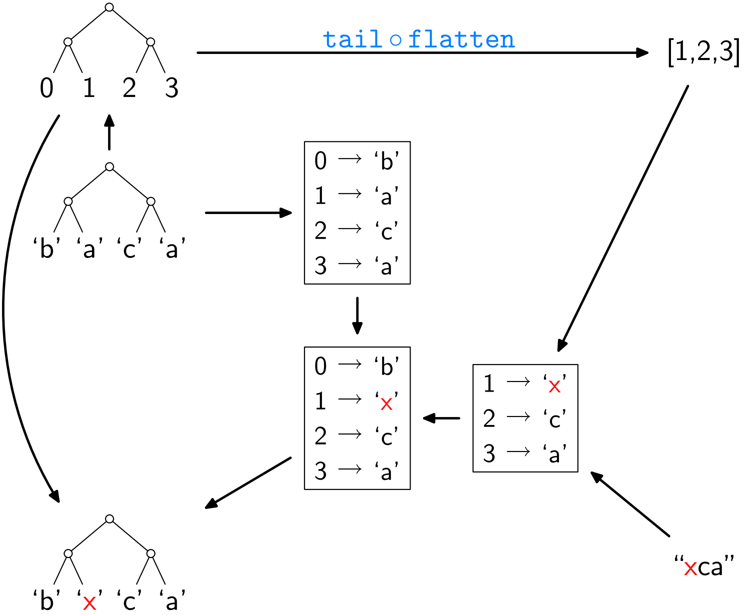
The Resulting Bidirectionalisation Scheme by Example



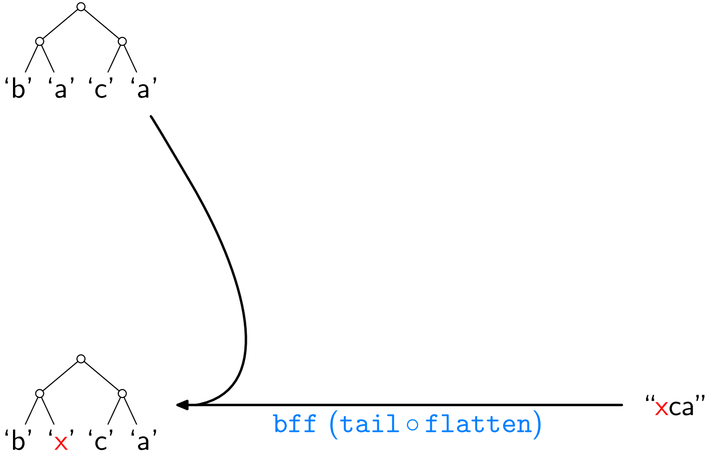
The Resulting Bidirectionalisation Scheme by Example



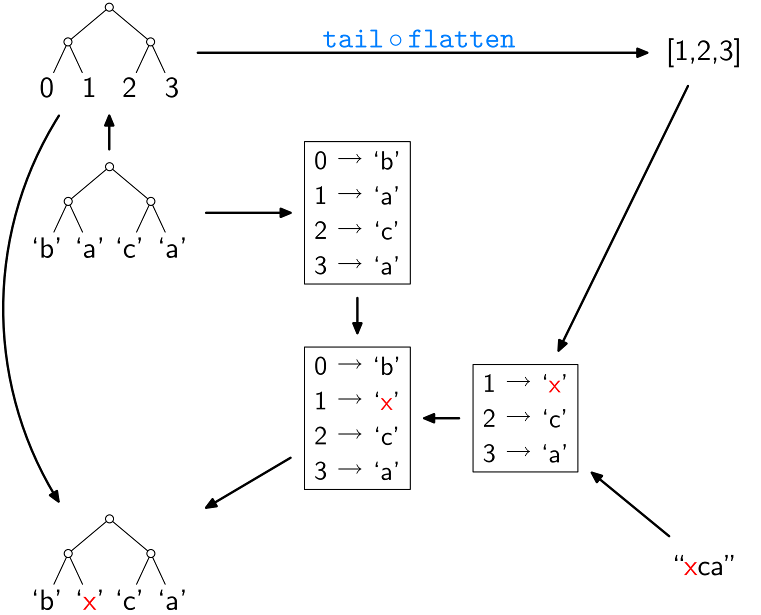
The Resulting Bidirectionalisation Scheme by Example



The Resulting Bidirectionalisation Scheme by Example



The Resulting Bidirectionalisation Scheme by Example



The Implementation (here: lists only, inefficient version)

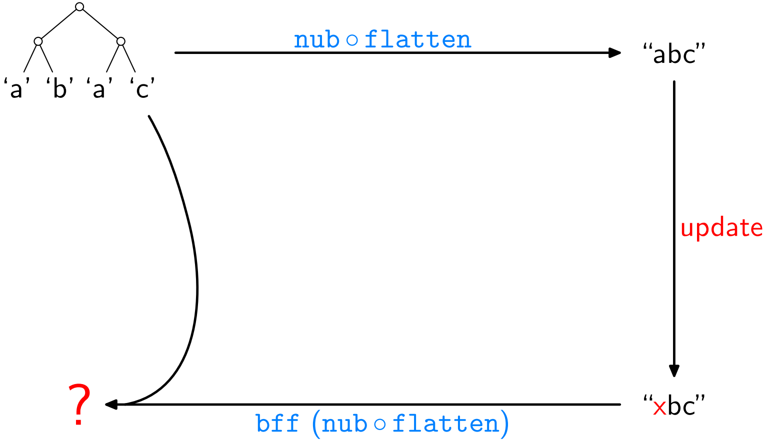
```
bff get s v' = let n  = (length s) − 1
                   t  = [0..n]
                   g  = zip t s
                   h  = assoc (get t) v'
                   h' = h ++ g
               in  seq h (map (λi → fromJust (lookup i h')) t)

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
```

The Implementation (here: lists only, inefficient version)

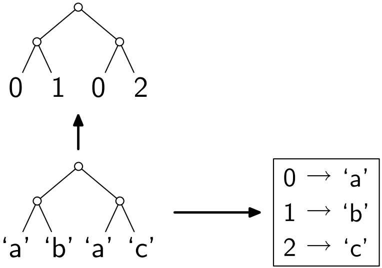
```
bff get s v' = let n  = (length s) − 1
                   t  = [0..n]
                   g  = zip t s
                   h  = assoc (get t) v'
                   h' = h ++ g
               in  seq h (map (λi → fromJust (lookup i h')) t)

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
```

- actual code only slightly more elaborate
- online: `http://linux.tcs.inf.tu-dresden.de/~bff`

Another Interesting Example

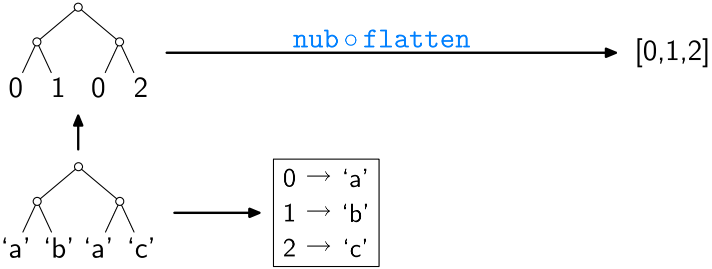


Another Interesting Example



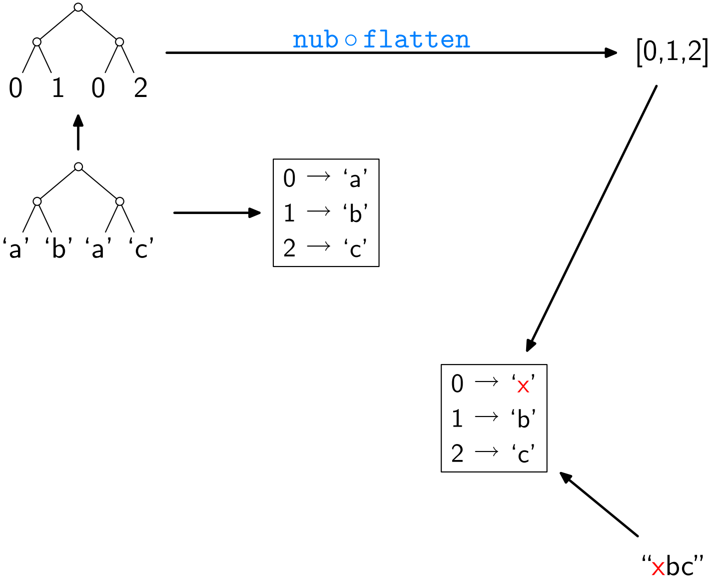
"xbc"

Another Interesting Example

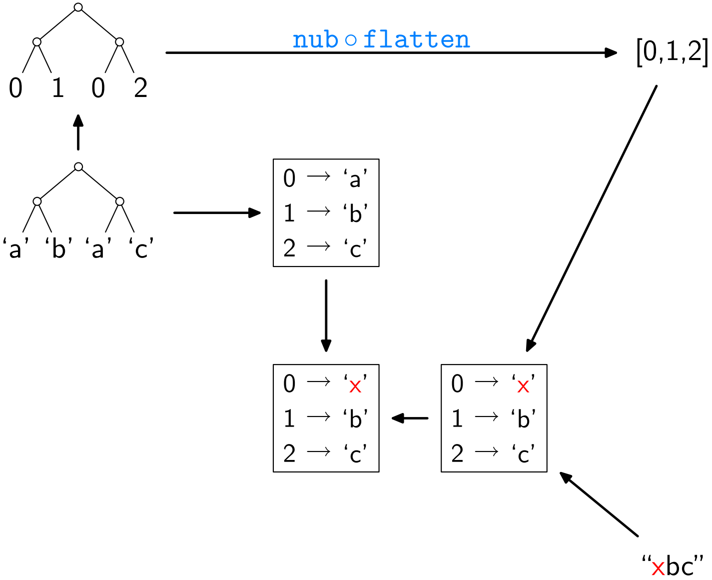


“xbc”

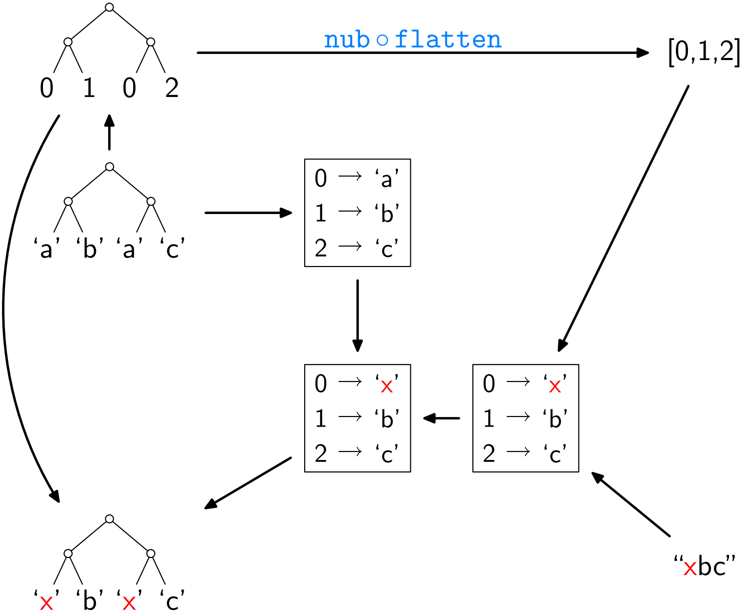
Another Interesting Example



Another Interesting Example



Another Interesting Example



What Else?

[V., POPL'09]:

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

What Else?

[V., POPL'09]:

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

Pros of the approach:

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

What Else?

[V., POPL'09]:

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

Pros of the approach:

- 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

What Else?

[V., POPL'09]:

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

Pros of the approach:

- 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

Outlook:

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

Short Course “Free Theorems and Applications”

Three lectures, 22nd–24th April, 16.00–17.00, room IF 3.02

1. Free Theorems — Foundations
   - from intuition to a formal account
   - actually deriving free theorems
2. Knuth’s 0-1-Principle and Beyond
   - reducing algorithm correctness from infinite to finite cases
   - comparison-swap sorting and parallel prefix computation
3. Free Theorems and “Real” Languages
   - free theorems and type classes
   - free theorems and general recursion
   - automatic generation of counterexamples

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.