News About
A Recent Application of Parametricity

Janis Voigtländer

Technische Universität Dresden

ISS-AiPL'09

Functional Programming in Haskell

A standard function:

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

Functional Programming in Haskell

A standard function:

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

Some invocations:

$\texttt{map}\ \texttt{succ}\ [1, 2, 3] \qquad = [2, 3, 4]$

Functional Programming in Haskell

A standard function:

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

Some invocations:

$$\begin{aligned}&\texttt{map}\ \texttt{succ}\ [1, 2, 3] &&= [2, 3, 4] \\ &\texttt{map}\ \texttt{not}\ \ [\text{True}, \text{False}] &&= [\text{False}, \text{True}]\end{aligned}$$

Functional Programming in Haskell

A standard function:

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

Some invocations:

$$\begin{aligned} \texttt{map succ}\ [1,2,3] &= [2,3,4] \\ \texttt{map not}\ \ [\mathsf{True}, \mathsf{False}] &= [\mathsf{False}, \mathsf{True}] \\ \texttt{map even}\ [1,2,3] &= [\mathsf{False}, \mathsf{True}, \mathsf{False}] \end{aligned}$$

Functional Programming in Haskell

A standard function:

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

Some invocations:

$$\begin{aligned} &\texttt{map succ}\ [1, 2, 3] &&= [2, 3, 4] \\ &\texttt{map not}\ \ [\mathsf{True}, \mathsf{False}] &&= [\mathsf{False}, \mathsf{True}] \\ &\texttt{map even}\ [1, 2, 3] &&= [\mathsf{False}, \mathsf{True}, \mathsf{False}] \\ &\texttt{map not}\ \ [1, 2, 3] && \end{aligned}$$

Functional Programming in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$

$\texttt{map not}\ [\text{True}, \text{False}] = [\text{False}, \text{True}]$

$\texttt{map even}\ [1, 2, 3] = [\text{False}, \text{True}, \text{False}]$

$\texttt{map not}\ [1, 2, 3]$

Functional Programming in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$\texttt{map succ}$	$[1, 2, 3]$	$= [2, 3, 4]$	— $\alpha, \beta \mapsto \text{Int}, \text{Int}$
$\texttt{map not}$	$[\text{True}, \text{False}]$	$= [\text{False}, \text{True}]$	— $\alpha, \beta \mapsto \text{Bool}, \text{Bool}$
$\texttt{map even}$	$[1, 2, 3]$	$= [\text{False}, \text{True}, \text{False}]$	— $\alpha, \beta \mapsto \text{Int}, \text{Bool}$
$\texttt{map not}$	$[1, 2, 3]$		

Functional Programming in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [\,] = [\,]$$
$$\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)$$

Some invocations:

$\texttt{map succ}\ [1,2,3] = [2,3,4]$		— $\alpha, \beta \mapsto$ Int, Int
$\texttt{map not}\ [\text{True}, \text{False}] = [\text{False}, \text{True}]$		— $\alpha, \beta \mapsto$ Bool, Bool
$\texttt{map even}\ [1,2,3] = [\text{False}, \text{True}, \text{False}]$		— $\alpha, \beta \mapsto$ Int, Bool
$\texttt{map not}\ [1,2,3]$	↯ rejected at compile-time	

Functional Programming in Haskell

A standard function:

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

Some invocations:

`map succ` $[1, 2, 3]$	$= [2, 3, 4]$	— $\alpha, \beta \mapsto$ Int, Int
`map not` [True, False]	= [False, True]	— $\alpha, \beta \mapsto$ Bool, Bool
`map even` $[1, 2, 3]$	= [False, True, False]	— $\alpha, \beta \mapsto$ Int, Bool
`map not` $[1, 2, 3]$	$\lightning$ rejected at compile-time	

Another Example

$$\texttt{reverse} :: [\alpha] \rightarrow [\alpha]$$
$$\texttt{reverse}\ [\,] \quad = [\,]$$
$$\texttt{reverse}\ (a : as) = (\texttt{reverse}\ as) \mathbin{+\!\!+} [a]$$

Another Example

$$\begin{aligned}
&\texttt{reverse} :: [\alpha] \to [\alpha] \\
&\texttt{reverse}\ [\,] \qquad\quad = [\,] \\
&\texttt{reverse}\ (a : as) = (\texttt{reverse}\ as) \mathbin{+\!\!+} [a]
\end{aligned}$$

For every choice of f and l:

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

Provable by induction.

Another Example

$$\texttt{reverse} :: [\alpha] \to [\alpha]$$
$$\texttt{reverse}\ [\,] = [\,]$$
$$\texttt{reverse}\ (a : as) = (\texttt{reverse}\ as) \mathbin{+\!\!+} [a]$$

For every choice of f and l:

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

Provable by induction.

Or as a "free theorem" [Wadler, FPCA'89].

Another Example

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

For every choice of f and l:

$$\texttt{reverse}\,(\texttt{map}\ f\ \ l) \ \ = \ \ \texttt{map}\ f\ (\texttt{reverse}\ l)$$

Provable by induction.

Or as a "free theorem" [Wadler, FPCA'89].

Another Example

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

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

For every choice of f and l:

$$\texttt{reverse}\,(\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{reverse}\ l)$$

$$\texttt{tail}\,(\texttt{map}\ f\ l) \;=\; \texttt{map}\ f\ (\texttt{tail}\ l)$$

Another Example

$$\texttt{reverse} :: [\alpha] \to [\alpha]$$
$$\texttt{tail} :: [\alpha] \to [\alpha]$$
$$\texttt{g} :: [\alpha] \to [\alpha]$$

For every choice of f and l:

$$\texttt{reverse}\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{reverse}\ l)$$
$$\texttt{tail}\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{tail}\ l)$$
$$\texttt{g}\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{g}\ l)$$

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- . . .
- Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]

Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]
- Bidirectionalisation [V., POPL'09]

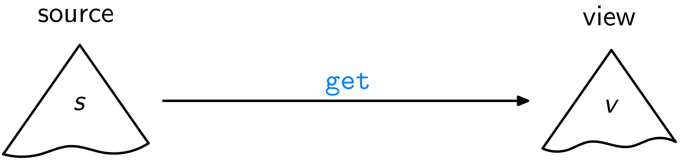
Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]
- Bidirectionalisation [V., POPL'09]
- Reasoning about invariants for monadic programs [V., ICFP'09]

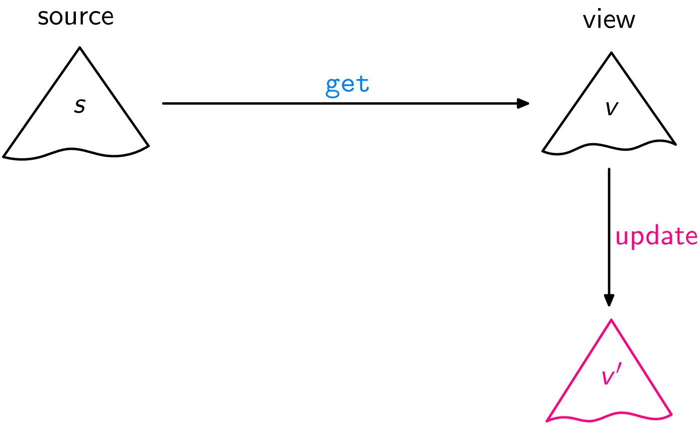
Some Applications

- Short Cut Fusion [Gill et al., FPCA'93]
- The Dual of Short Cut Fusion [Svenningsson, ICFP'02]
- Circular Short Cut Fusion [Fernandes et al., Haskell'07]
- ...
- Knuth's 0-1-principle and the like [Day et al., Haskell'99], [V., POPL'08]
- Bidirectionalisation [V., POPL'09]
- Reasoning about invariants for monadic programs [V., ICFP'09]

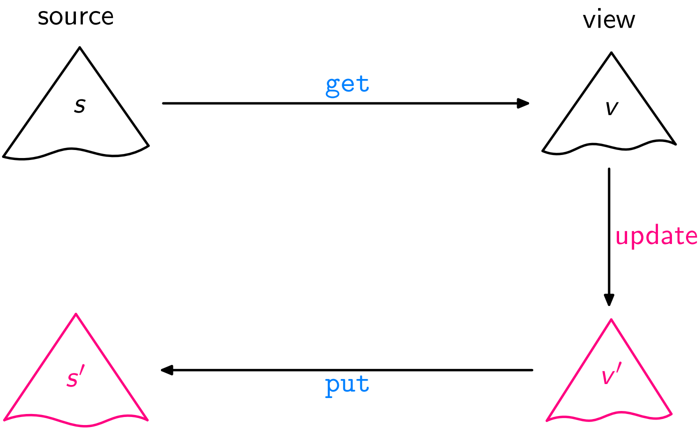
Bidirectional Transformation



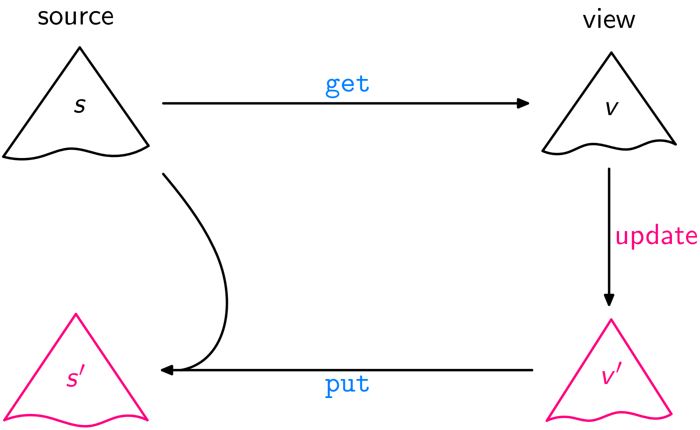
Bidirectional Transformation



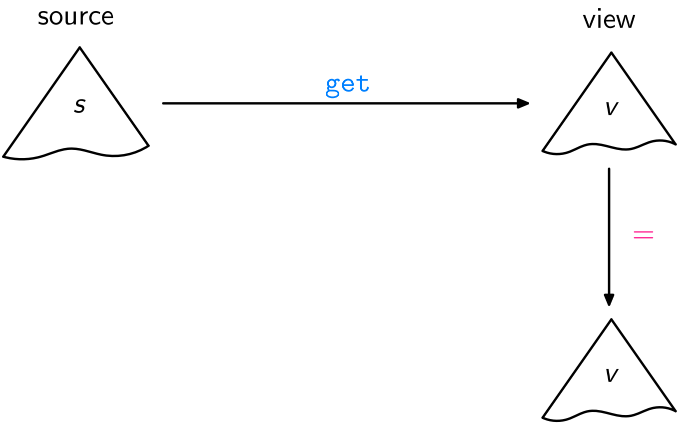
Bidirectional Transformation



Bidirectional Transformation

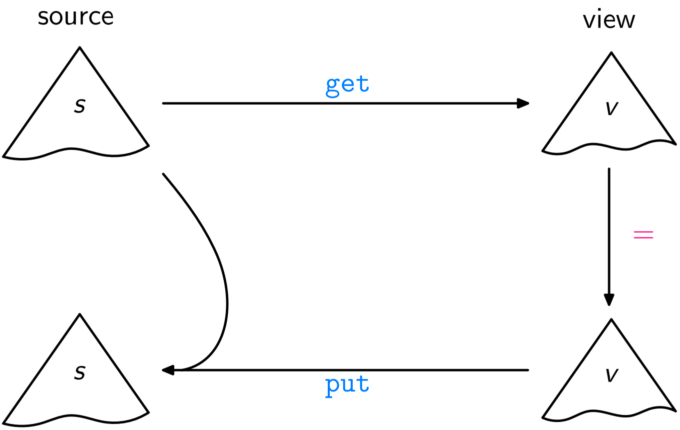


Bidirectional Transformation



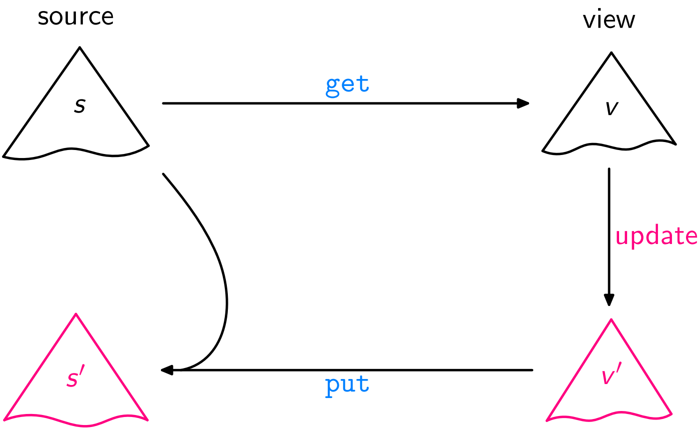
Acceptability / GetPut

Bidirectional Transformation



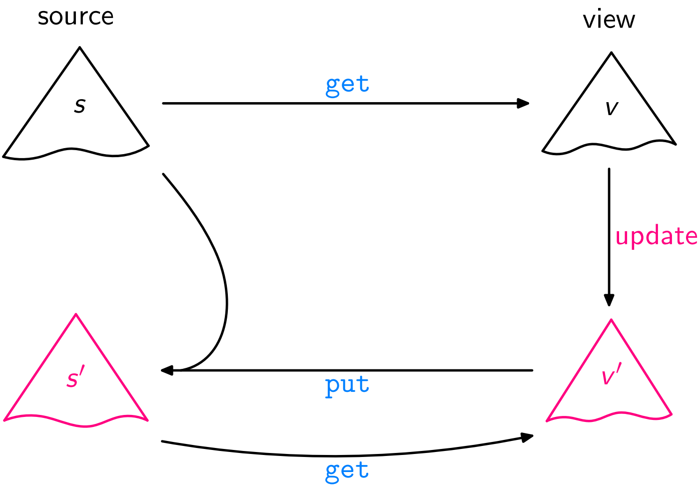
Acceptability / GetPut

Bidirectional Transformation



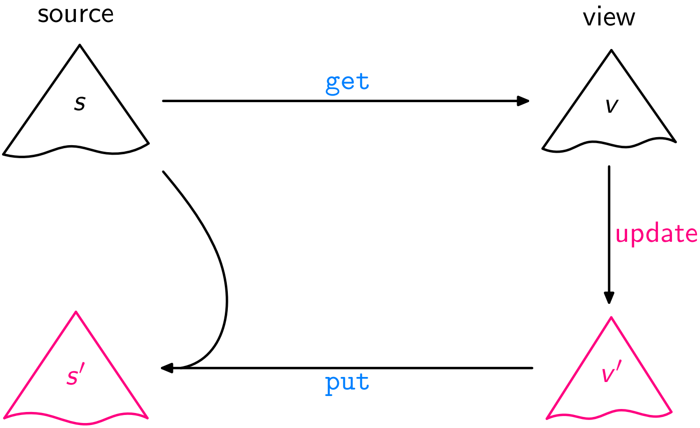
Consistency / PutGet

Bidirectional Transformation

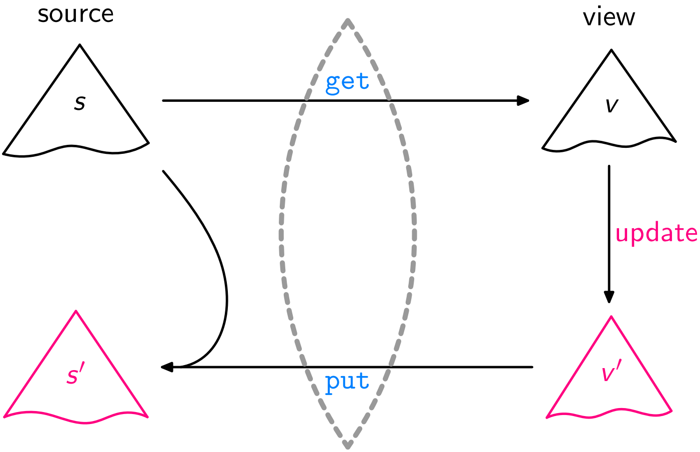


Consistency / PutGet

Bidirectional Transformation



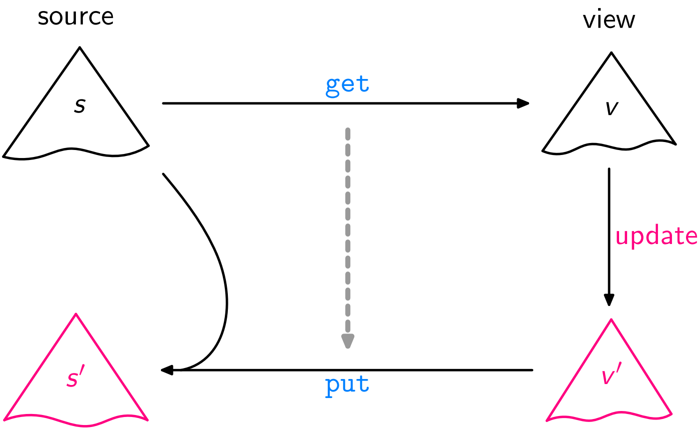
Bidirectional Transformation



Lenses, DSLs

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

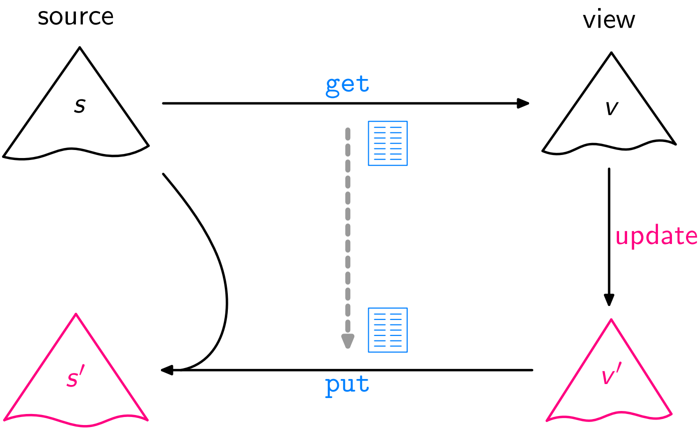
Bidirectional Transformation



Bidirectionalisation

[Matsuda et al., ICFP'07]

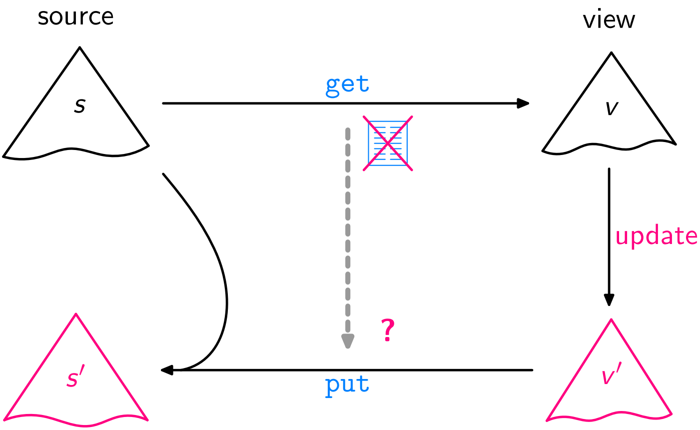
Bidirectional Transformation



Syntactic Bidirectionalisation

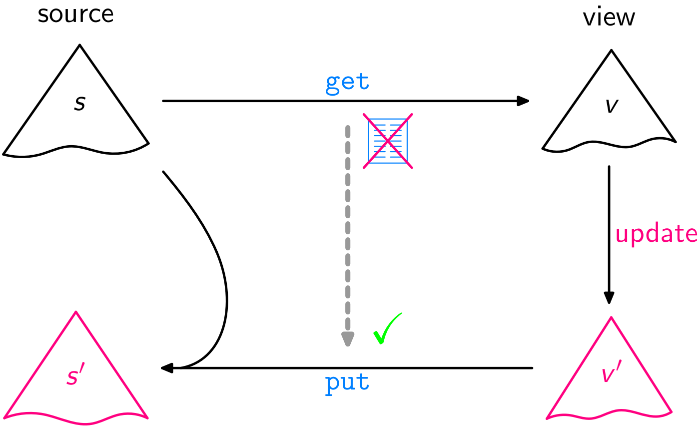
[Matsuda et al., ICFP'07]

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`† such that any `get` and `bff get` satisfy GetPut, PutGet, ...

† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

Examples:

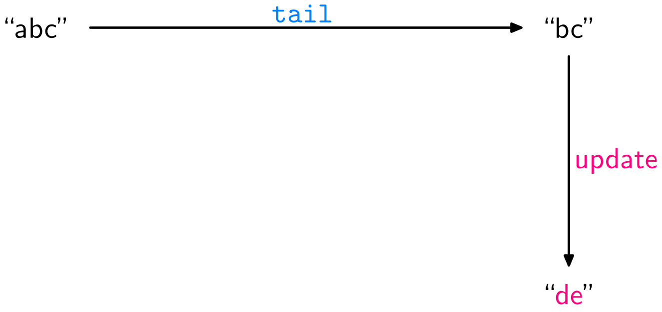
"abc" —`tail`→ "bc"

† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

Examples:

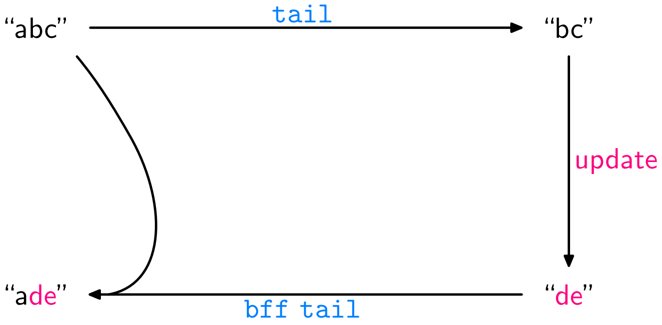


† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

Examples:

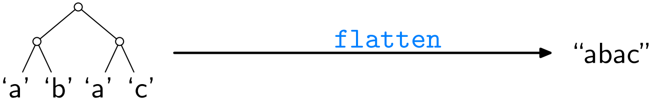


† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

Examples:



† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

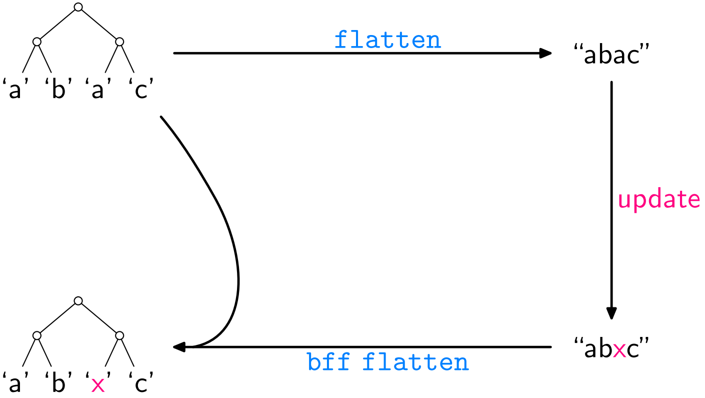
Examples:

† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

Examples:

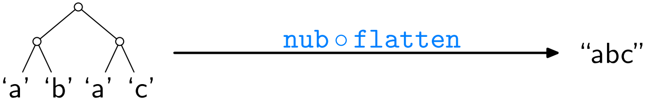


† “Bidirectionalization for free!”

Semantic Bidirectionalisation

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

Examples:



† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

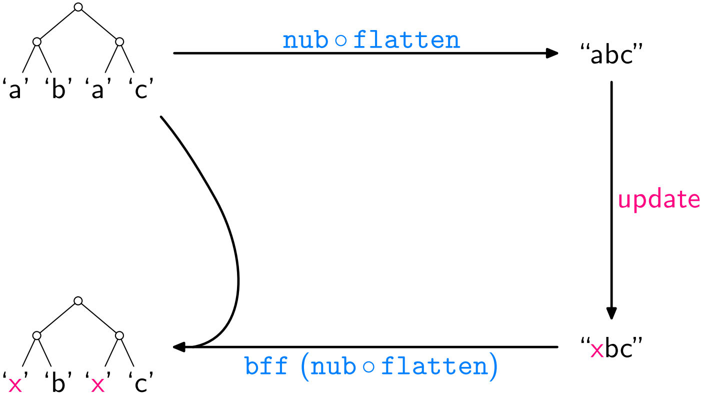
Examples:

† “Bidirectionalization for free!”

Semantic Bidirectionalisation

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

Examples:



† “Bidirectionalization for free!”

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?

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?

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

For every

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

we have

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

for arbitrary f and l.

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l.

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

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

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l.

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

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

Using a Free Theorem

For every

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

we have

$$\texttt{map}\ f\ (\texttt{g}\ l) \;=\; \texttt{g}\ (\texttt{map}\ f\ l)$$

for arbitrary f and l.

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

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

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

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l.

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

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

for every $\texttt{get} :: [\alpha] \rightarrow [\alpha]$.

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

In general, given

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

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

In general, given

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

define a V^C and

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

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

In general, given

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

define a V^C and

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

such that

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

is injective

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

In general, given

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

define a V^C and

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

such that

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

is injective and has an inverse

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

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

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

define a V^C and

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

such that

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

is injective and has an inverse

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

Then:

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

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

The Constant-Complement Approach

For our setting,

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

what should be V^C and

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

The Constant-Complement Approach

For our 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 our setting,

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

what should be V^C and

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

To make

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

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

Candidates:

1. length of the source list

The Constant-Complement Approach

For our setting,

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

what should be V^C and

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

To make

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

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

Candidates:

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

The Constant-Complement Approach

For our 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

$$\textbf{type}\ \mathsf{IntMap}\ \alpha = [(\mathsf{Int}, \alpha)]$$

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

The Complement Function

$$\textbf{type}\ \mathsf{IntMap}\ \alpha = [(\mathsf{Int}, \alpha)]$$

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

For example:

$$\texttt{get} = \texttt{tail} \qquad \rightsquigarrow \qquad \texttt{compl}\ \text{“abcde”} = (5, [(0, \text{'a'})])$$

The Complement Function

$$\textbf{type}\ \mathsf{IntMap}\ \alpha = [(\mathsf{Int}, \alpha)]$$

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

For example:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [(0, \text{'a'})])$$

$$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [(3, \text{'d'}), (4, \text{'e'})])$$

The Complement Function

$$\textbf{type}\ \mathsf{IntMap}\ \alpha = [(\mathsf{Int}, \alpha)]$$

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

For example:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [(0, \text{'a'})])$$

$$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [(3, \text{'d'}), (4, \text{'e'})])$$

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

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

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

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

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

† Can be thought of as zip for the moment.

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

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], (\text{Int}, \text{IntMap}\ \alpha)) \rightarrow [\alpha] \\
&\texttt{inv}\ (v', (n+1, g')) = \textbf{let}\ t\ \ = [0..n] \\
&\qquad\qquad\qquad\qquad\quad\ \ h\ \ = \texttt{assoc}^{\dagger}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\qquad\quad\ \ h' = h + g' \\
&\qquad\qquad\qquad\qquad\ \ \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \rightarrow \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

For example:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{inv}\ (\text{“bcde”}, (5, [(0, \text{'a'})])) = \text{“abcde”}$$

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

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

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

For example:

$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{inv}\ (\text{“bcde”}, (5, [(0, \text{'a'})])) = \text{“abcde”}$

$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{inv}\ (\text{“xyz”}, (5, [(3, \text{'d'}), (4, \text{'e'})])) = \text{“xyzde”}$

† Can be thought of as zip for the moment.

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

$$\begin{aligned}
&\texttt{inv} :: ([\alpha], (\mathsf{Int}, \mathsf{IntMap}\ \alpha)) \rightarrow [\alpha] \\
&\texttt{inv}\ (v', (n+1, g')) = \mathbf{let}\ t\ \ = [0..n] \\
&\qquad\qquad\qquad\qquad\quad\ \ h\ \ = \texttt{assoc}^{\dagger}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\qquad\quad\ \ h' = h \mathbin{+\!\!+} g' \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \rightarrow \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}$$

For example:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{inv}\ (\text{“bcde”}, (5, [(0, \text{'a'})])) = \text{“abcde”}$$

$$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{inv}\ (\text{“xyz”}, (5, [(3, \text{'d'}), (4, \text{'e'})])) = \text{“xyzde”}$$

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 $\texttt{compl}\ (\texttt{inv}\ (v, c)) = c$

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

Altogether:

```
type IntMap α = [(Int, α)]

compl :: [α] → (Int, IntMap 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, g′)

inv :: ([α], (Int, IntMap α)) → [α]
inv (v′, (n + 1, g′)) = let t  = [0..n]
                            h  = assoc (get t) v′
                            h′ = h ++ g′
                        in  seq h (map (λi → fromJust (lookup i h′)) t)

put :: [α] → [α] → [α]
put s v′ = inv (v′, compl s)
```

"Fusion"

Inlining compl and inv into put:

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

"Fusion"

Inlining `compl` and `inv` into `put`:

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

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

"Fusion"

Inlining `compl` and `inv` into `put`:

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

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

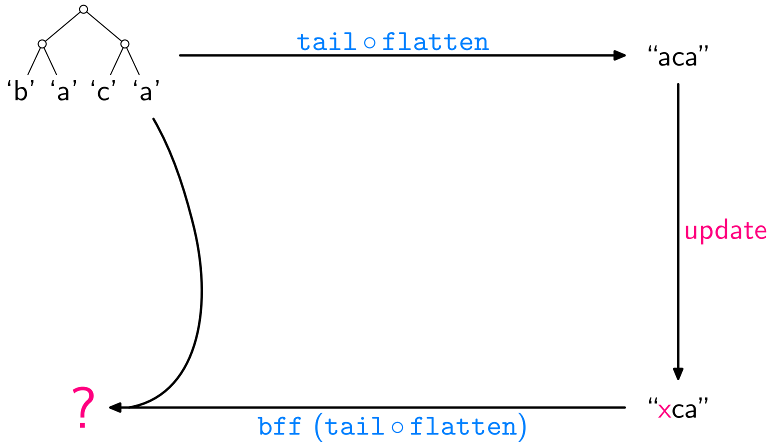
"Fusion"

Inlining `compl` and `inv` into `put`:

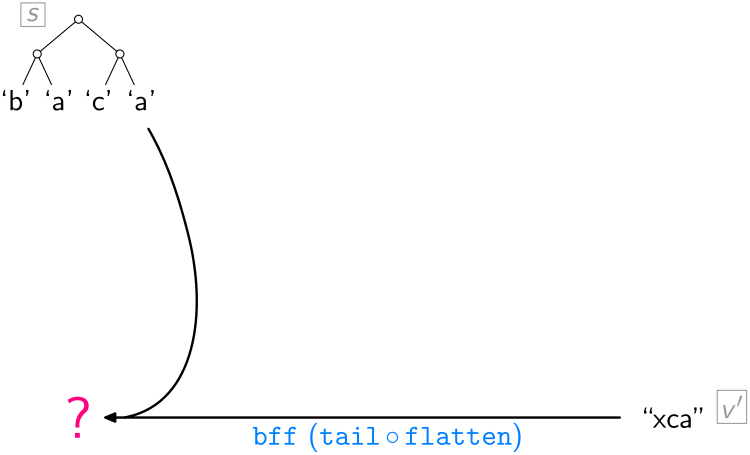
```
bff get s v' = let n  = (length s) − 1
                   t  = [0..n]
                   g  = zip t s
                   g' = filter (λ(i, _) → notElem i (get t)) g
                   h  = assoc (get t) v'
                   h' = h ++ g'
               in  seq h (map (λi → fromJust (lookup i h'))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

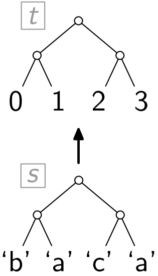
The Resulting Bidirectionalisation Method in Action



The Resulting Bidirectionalisation Method in Action

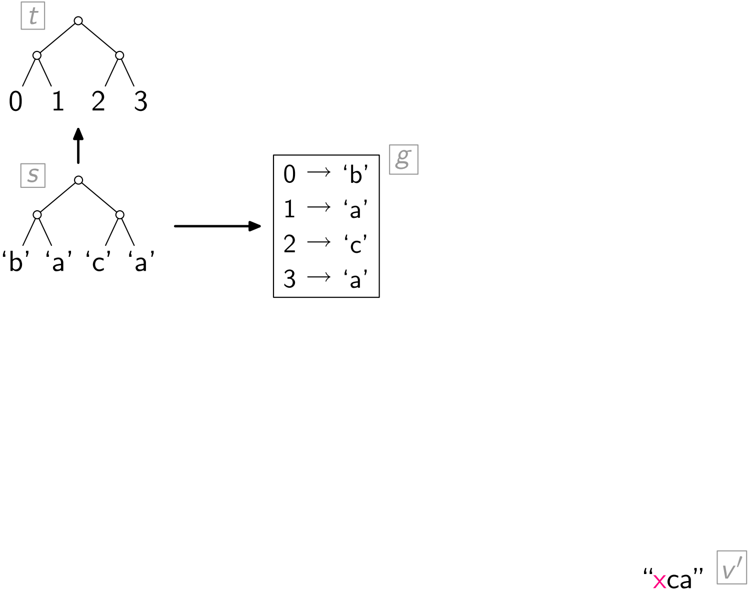


The Resulting Bidirectionalisation Method in Action

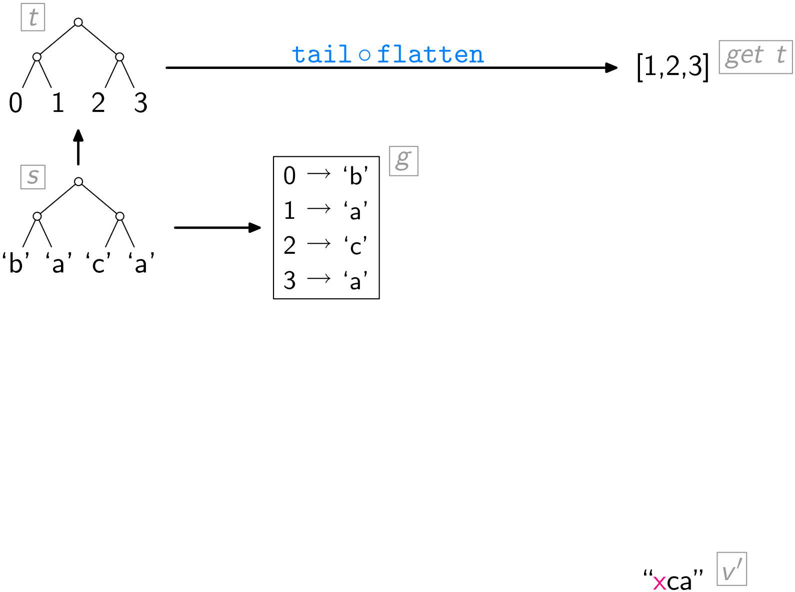


"xca" v'

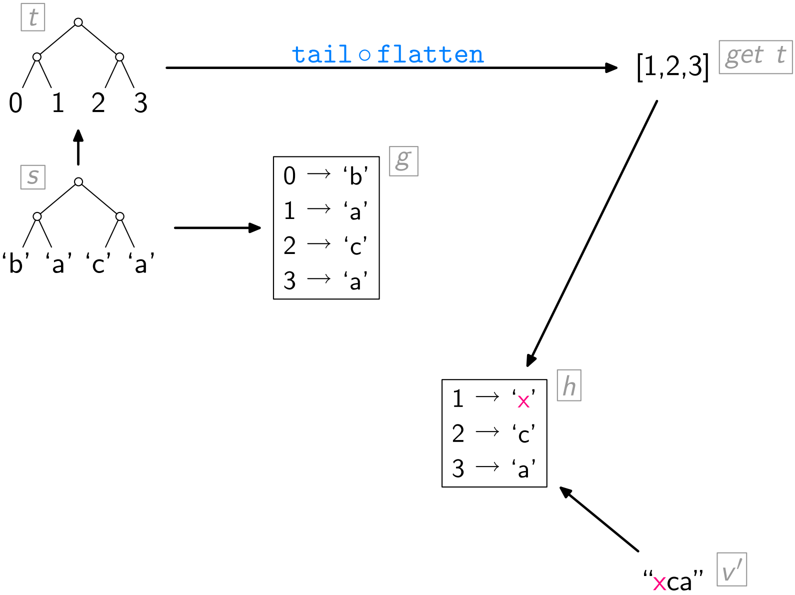
The Resulting Bidirectionalisation Method in Action



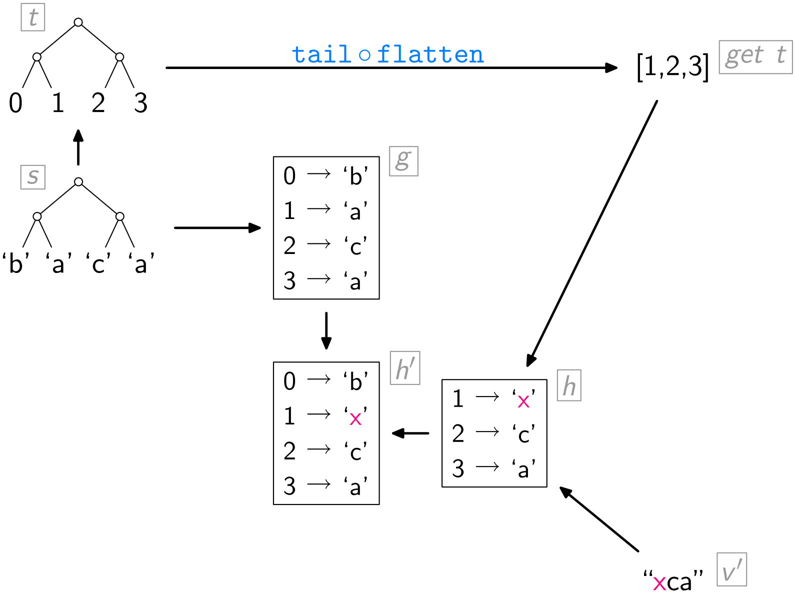
The Resulting Bidirectionalisation Method in Action



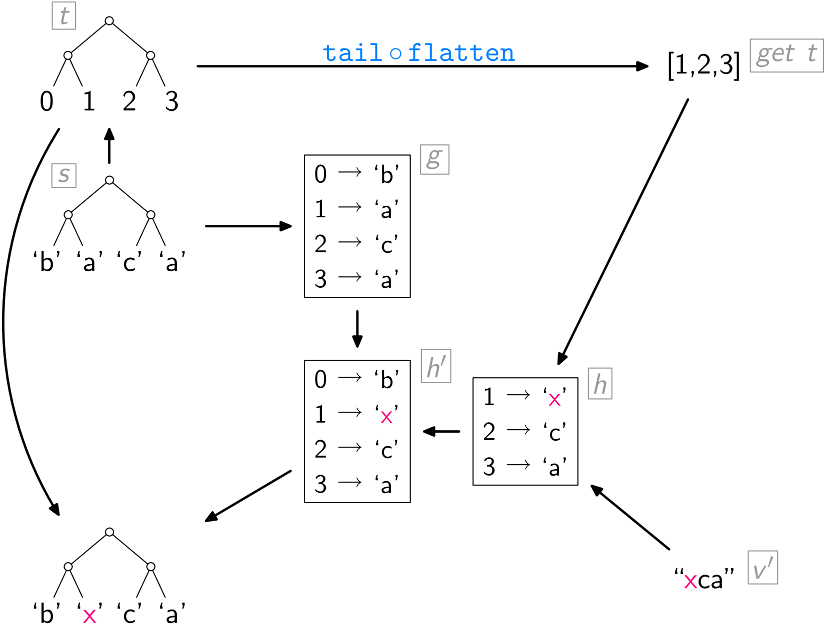
The Resulting Bidirectionalisation Method in Action



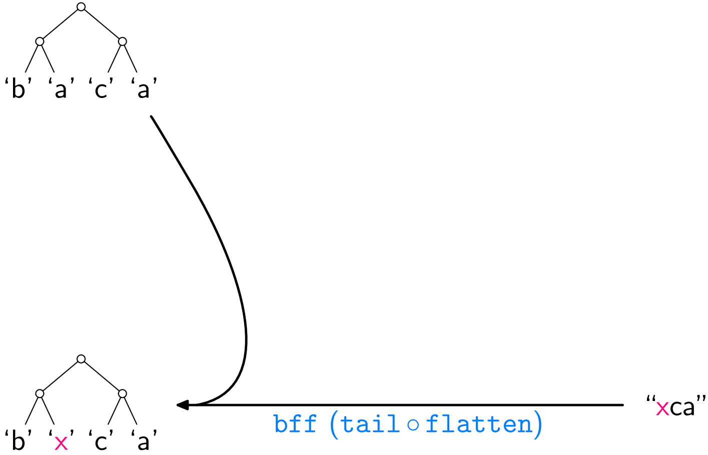
The Resulting Bidirectionalisation Method in Action



The Resulting Bidirectionalisation Method in Action



The Resulting Bidirectionalisation Method in Action



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

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

  injective was to record, via `compl`, the following information:

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

- 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

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

  injective was to record, via `compl`, the following information:

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

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

- For example:

  $\texttt{get} = \texttt{tail}$ $\rightsquigarrow$ `put` “abcde” “xyz” fails precisely because `compl` “abcde” $= (5, [(0, \text{'a'})])$

Assuming Shape-Injectivity

So assume there is a function

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

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

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

with, for every source list s,

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

Then:

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

Assuming Shape-Injectivity

So assume there is a function

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

with, for every source list s,

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

Then:

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

Assuming Shape-Injectivity

$$
\begin{array}{l}
\texttt{inv} :: ([\alpha], (\mathsf{Int}, \mathsf{IntMap}\ \alpha)) \to [\alpha] \\
\texttt{inv}\ (v', (n+1, g')) = \mathbf{let}\ \begin{array}[t]{l} t\ = [0..n] \\ h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\ h' = h +\!\!+\ g' \end{array} \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{array}
$$

Assuming Shape-Injectivity

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], \quad \mathsf{IntMap}\ \alpha\ ) \to [\alpha] \\
&\texttt{inv}\ (v', \quad g'\ ) = \mathbf{let}\ n\ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
&\qquad\qquad\qquad\qquad\quad t\ = [0..n] \\
&\qquad\qquad\qquad\qquad\quad h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\qquad\quad h' = h +\!\!+\ g' \\
&\qquad\qquad\qquad\quad \mathbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

Assuming Shape-Injectivity

$$
\begin{array}{l}
\texttt{inv} :: ([\alpha], \quad \mathsf{IntMap}\ \alpha\ ) \to [\alpha] \\
\texttt{inv}\ (v', \quad g'\ ) = \textbf{let}\ n\ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
\qquad\qquad t\ = [0..n] \\
\qquad\qquad h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
\qquad\qquad h' = h \mathbin{+\!\!+} g' \\
\qquad \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{array}
$$

But how to obtain `shapeInv` ???

Assuming Shape-Injectivity

$$
\begin{array}{ll}
\texttt{inv} :: ([\alpha], \quad \mathrm{IntMap}\ \alpha\ ) \to [\alpha] \\
\texttt{inv}\ (v', \quad g'\ ) = \textbf{let} & n\ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
& t\ = [0..n] \\
& h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
& h' = h +\!\!+\ g' \\
\qquad\qquad \textbf{in} & \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{array}
$$

But how to obtain `shapeInv` ???

One possibility: provided by user.

Assuming Shape-Injectivity

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], \quad \mathsf{IntMap}\ \alpha\ ) \to [\alpha] \\
&\texttt{inv}\ (v', \quad g'\ ) = \textbf{let}\ n\ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
&\qquad\qquad\qquad\qquad\quad t\ = [0..n] \\
&\qquad\qquad\qquad\qquad\quad h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\qquad\quad h' = h +\!\!+\ g' \\
&\qquad\qquad\qquad\quad \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

But how to obtain `shapeInv` ???

One possibility: provided by user.

Another possibility: determined statically (dependent types?).

Assuming Shape-Injectivity

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], \quad \text{IntMap } \alpha\ ) \to [\alpha] \\
&\texttt{inv}\ (v', \quad g'\ ) = \textbf{let}\ n\ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
&\qquad\qquad\qquad\qquad\quad t\ = [0..n] \\
&\qquad\qquad\qquad\qquad\quad h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\qquad\quad h' = h +\!\!+\ g' \\
&\qquad\qquad\qquad\qquad \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

But how to obtain `shapeInv` ???

One possibility: provided by user.

Another possibility: determined statically (dependent types?).

Just for experimentation:

$$
\begin{aligned}
&\texttt{shapeInv} :: \text{Int} \to \text{Int} \\
&\texttt{shapeInv}\ l = \texttt{head}\ [n+1 \mid n \leftarrow [0..],\ (\texttt{length}\ (\texttt{get}\ [0..n])) == l]
\end{aligned}
$$

Not Quite There, Yet

Works quite nicely in some cases:

$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}$ “abcde” “xyz” = “axyz”

Not Quite There, Yet

Works quite nicely in some cases:

$$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“axyz”, using} \\ \texttt{compl}\ \text{“abcde”} = [(0, \text{'a'})] \end{array}$$

Not Quite There, Yet

Works quite nicely in some cases:

$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}$ "abcde" "xyz" = "axyz", using
$\texttt{compl}$ "abcde" = [(0, 'a')]

But not so in others:

$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}$ "abcde" "xyz" fails

Not Quite There, Yet

Works quite nicely in some cases:

$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad \texttt{put}$ "abcde" "xyz" = "axyz", using
$\texttt{compl}$ "abcde" = [(0, 'a')]

But not so in others:

$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}$ "abcde" "xyz" fails, because
$\texttt{compl}$ "abcde" = [(4, 'e')]

Not Quite There, Yet

Works quite nicely in some cases:

$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad$ $\texttt{put}$ "abcde" "xyz" = "axyz", using
$\texttt{compl}$ "abcde" = [(0, 'a')]

But not so in others:

$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad$ $\texttt{put}$ "abcde" "xyz" fails, because
$\texttt{compl}$ "abcde" = [(4, 'e')]

The problem: by keeping indices around, compl still does not "collapse enough".

Not Quite There, Yet

Works quite nicely in some cases:

$\texttt{get} = \texttt{tail} \quad \rightsquigarrow \quad$ $\texttt{put}$ "abcde" "xyz" = "axyz", using
$\texttt{compl}$ "abcde" = [(0, 'a')]

But not so in others:

$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad$ $\texttt{put}$ "abcde" "xyz" fails, because
$\texttt{compl}$ "abcde" = [(4, 'e')]

The problem: by keeping indices around, `compl` still does not "collapse enough".

Note: even without these indices, $\lambda s \rightarrow (\texttt{get}\ s, \texttt{compl}\ s)$ would be injective.

Eliminating Indices

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

Eliminating Indices

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

Eliminating Indices

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

$$
\begin{array}{l}
\texttt{inv} :: ([\alpha], [(\mathsf{Int}, \alpha)]) \to [\alpha] \\
\texttt{inv}\ (v', g') = \textbf{let}\ n \ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
\qquad\qquad\qquad\ t \ = [0..n] \\
\qquad\qquad\qquad\ h \ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
\qquad\qquad\qquad\ h' = h \mathbin{+\!\!+} g' \\
\qquad\qquad\quad\ \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{array}
$$

Eliminating Indices

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

$$
\begin{array}{l}
\texttt{inv} :: ([\alpha], [\quad \alpha \ ]) \rightarrow [\alpha] \\
\texttt{inv}\ (v', c\ ) = \textbf{let}\ n \ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
\qquad\qquad\qquad\quad t \ = [0..n] \\
\qquad\qquad\qquad\quad h \ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
\qquad\qquad\qquad\quad g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \rightarrow \texttt{notElem}\ i\ (\texttt{get}\ t))\ t)\ c \\
\qquad\qquad\qquad\quad h' = h + + g' \\
\qquad\qquad\quad \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \rightarrow \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{array}
$$

Eliminating Indices

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

$$
\begin{array}{l}
\texttt{inv} :: ([\alpha], [\quad \alpha\ ]) \rightarrow [\alpha] \\
\texttt{inv}\ (v', c\ ) = \textbf{let}\ n \ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
\qquad\qquad\qquad t \ = [0..n] \\
\qquad\qquad\qquad h \ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
\qquad\qquad\qquad g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \rightarrow \texttt{notElem}\ i\ (\texttt{get}\ t))\ t)\ c \\
\qquad\qquad\qquad h' = h \mathbin{+\!\!+} g' \\
\qquad\qquad \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \rightarrow \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{array}
$$

Now:

$$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \ \text{“xyz”} = \text{“xyze”}$$

More Examples

Let $\texttt{get} = \texttt{sieve}$ with:

$$\begin{aligned}
&\texttt{sieve} :: [\alpha] \rightarrow [\alpha] \\
&\texttt{sieve}\ (a : b : cs) = b : (\texttt{sieve}\ cs) \\
&\texttt{sieve}\ _ \qquad\qquad\quad = []
\end{aligned}$$

More Examples

Let get = sieve with:

$$\texttt{sieve} :: [\alpha] \to [\alpha]$$
$$\texttt{sieve}\ (a : b : cs) = b : (\texttt{sieve}\ cs)$$
$$\texttt{sieve}\ \_ \qquad\qquad = []$$

Then:

$$\texttt{put}\ [1..8]\ [2, -4, 6, 8] \quad = \quad [1, 2, 3, -4, 5, 6, 7, 8]$$

More Examples

Let $\texttt{get} = \texttt{sieve}$ with:

$$\begin{aligned} &\texttt{sieve} :: [\alpha] \to [\alpha] \\ &\texttt{sieve}\ (a : b : cs) = b : (\texttt{sieve}\ cs) \\ &\texttt{sieve}\ \_ \qquad\qquad\ \ = [\,] \end{aligned}$$

Then:

$$\texttt{put}\ [1..8]\ [2, -4, 6, 8] \quad = \quad [1, 2, 3, -4, 5, 6, 7, 8]$$

$$\texttt{put}\ [1..8]\ [2, -4, 6] \quad = \quad [1, 2, 3, -4, 5, 6]$$

More Examples

Let $\texttt{get} = \texttt{sieve}$ with:

$$
\begin{aligned}
&\texttt{sieve} :: [\alpha] \to [\alpha] \\
&\texttt{sieve}\ (a : b : cs) = b : (\texttt{sieve}\ cs) \\
&\texttt{sieve}\ \_ \qquad\qquad\quad = [\,]
\end{aligned}
$$

Then:

$$
\begin{aligned}
\texttt{put}\ [1..8]\ [2,-4,6,8] &= [1,2,3,-4,5,6,7,8] \\
\texttt{put}\ [1..8]\ [2,-4,6] &= [1,2,3,-4,5,6] \\
\texttt{put}\ [1..8]\ [2,-4,6,8,10,12] &= [1,2,3,-4,5,6,7,8,\bot,10,\bot,12]
\end{aligned}
$$

More Examples

Let $\texttt{get} = \texttt{sieve}$ with:

$$\begin{aligned}
&\texttt{sieve} :: [\alpha] \to [\alpha] \\
&\texttt{sieve}\ (a : b : cs) = b : (\texttt{sieve}\ cs) \\
&\texttt{sieve}\ _ \qquad\qquad\quad = []
\end{aligned}$$

Then:

$$\begin{aligned}
\texttt{put}\ [1..8]\ [2, -4, 6, 8] &\quad=\quad [1, 2, 3, -4, 5, 6, 7, 8] \\
\texttt{put}\ [1..8]\ [2, -4, 6] &\quad=\quad [1, 2, 3, -4, 5, 6] \\
\texttt{put}\ [1..8]\ [2, -4, 6, 8, 10, 12] &\quad=\quad [1, 2, 3, -4, 5, 6, 7, 8, \bot, 10, \bot, 12]
\end{aligned}$$

However:

$$\texttt{put}\ [1..8]\ [0, 2, -4, 6, 8] \quad=\quad [1, 0, 3, 2, 5, -4, 7, 6, \bot, 8]$$

More Examples

Let $\texttt{get} = \texttt{sieve}$ with:

$$\begin{aligned}
&\texttt{sieve} :: [\alpha] \to [\alpha] \\
&\texttt{sieve}\ (a : b : cs) = b : (\texttt{sieve}\ cs) \\
&\texttt{sieve}\ _ \qquad\qquad\ \ = []
\end{aligned}$$

Then:

$$\begin{aligned}
\texttt{put}\ [1..8]\ [2, -4, 6, 8] &= [1, 2, 3, -4, 5, 6, 7, 8] \\
\texttt{put}\ [1..8]\ [2, -4, 6] &= [1, 2, 3, -4, 5, 6] \\
\texttt{put}\ [1..8]\ [2, -4, 6, 8, 10, 12] &= [1, 2, 3, -4, 5, 6, 7, 8, \bot, 10, \bot, 12]
\end{aligned}$$

However:

$$\texttt{put}\ [1..8]\ [0, 2, -4, 6, 8] = [1, 0, 3, 2, 5, -4, 7, 6, \bot, 8]$$

Whereas we might have preferred:

$$\texttt{put}\ [1..8]\ [0, 2, -4, 6, 8] = [\bot, 0, 1, 2, 3, -4, 5, 6, 7, 8]$$

Conclusion

Types:

- constrain the behaviour of programs

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- enable lightweight, semantic analysis methods

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- enable lightweight, semantic analysis methods

On the practical side:

- efficiency-improving program transformations

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- enable lightweight, semantic analysis methods

On the practical side:

- efficiency-improving program transformations
- applications in specific domains (more out there?)

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- enable lightweight, semantic analysis methods

On the practical side:

- efficiency-improving program transformations
- applications in specific domains (more out there?)

Bidirectionalisation in particular:

- hot topic (databases, models community, ...)

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- enable lightweight, semantic analysis methods

On the practical side:

- efficiency-improving program transformations
- applications in specific domains (more out there?)

Bidirectionalisation in particular:

- hot topic (databases, models community, ...)
- need a way to inject/exploit "user knowledge"

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- enable lightweight, semantic analysis methods

On the practical side:

- efficiency-improving program transformations
- applications in specific domains (more out there?)

Bidirectionalisation in particular:

- hot topic (databases, models community, ...)
- need a way to inject/exploit "user knowledge"

On the programming language side:

- push towards full programming languages

Conclusion

Types:

- constrain the behaviour of programs
- thus lead to interesting theorems about programs
- enable lightweight, semantic analysis methods

On the practical side:

- efficiency-improving program transformations
- applications in specific domains (more out there?)

Bidirectionalisation in particular:

- hot topic (databases, models community, . . .)
- need a way to inject/exploit "user knowledge"

On the programming language side:

- push towards full programming languages
- aim for exploiting more expressive type systems

References I

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

N.A. Day, J. Launchbury, and J. Lewis.
Logical abstractions in Haskell.
In *Haskell Workshop, Proceedings*. Technical Report UU-CS-1999-28, Utrecht University, 1999.

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.

References II

J.P. Fernandes, A. Pardo, and J. Saraiva.
A shortcut fusion rule for circular program calculation.
In *Haskell Workshop, Proceedings*, pages 95–106. ACM Press, 2007.

A. Gill, J. Launchbury, and S.L. Peyton Jones.
A short cut to deforestation.
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.

P. Johann and J. Voigtländer.
Free theorems in the presence of seq.
In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

References III

P. Johann and J. Voigtländer.
A family of syntactic logical relations for the semantics of Haskell-like languages.
Information and Computation, 207(2):341–368, 2009.

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.

F. Stenger and J. Voigtländer.
Parametricity for Haskell with imprecise error semantics.
In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009.

References IV

J. Svenningsson.
Shortcut fusion for accumulating parameters & zip-like functions.
In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM Press, 2002.

J. Voigtländer.
Much ado about two: A pearl on parallel prefix computation.
In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.

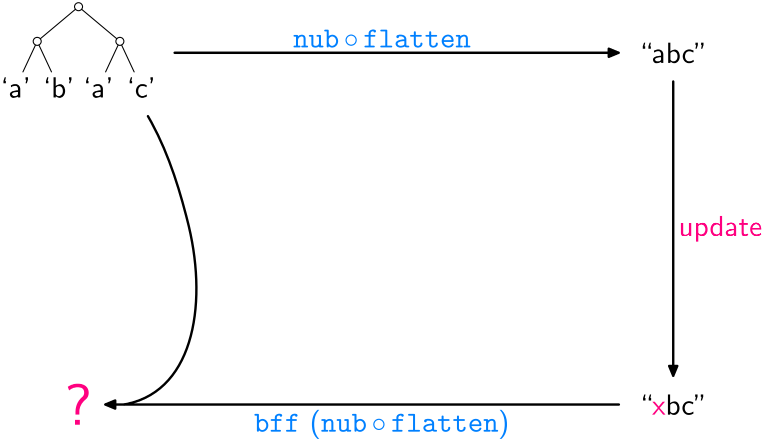
J. Voigtländer.
Bidirectionalization for free!
In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

References V

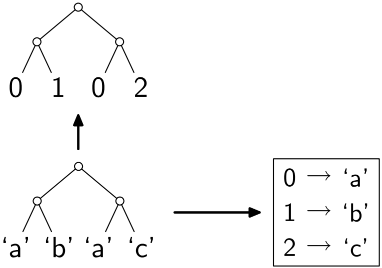
J. Voigtländer.
Free theorems involving type constructor classes.
In *International Conference on Functional Programming, Proceedings*. ACM Press, 2009.

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

Another Interesting Example (involving Eq type class)

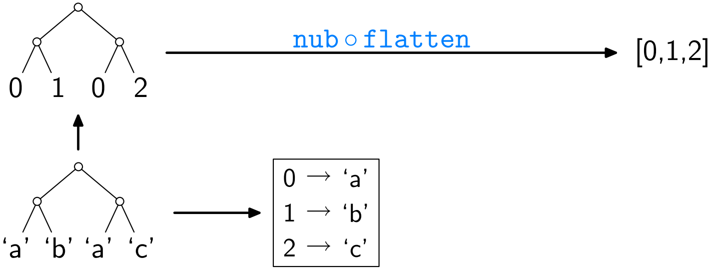


Another Interesting Example (involving Eq type class)



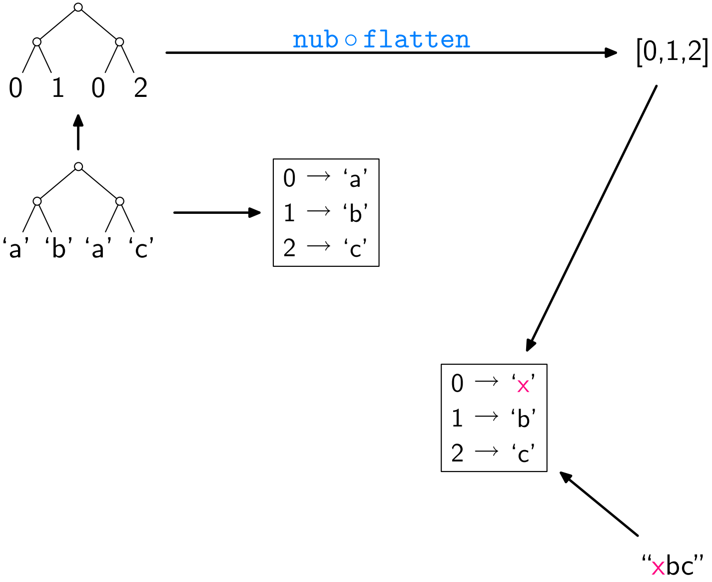
"xbc"

Another Interesting Example (involving Eq type class)

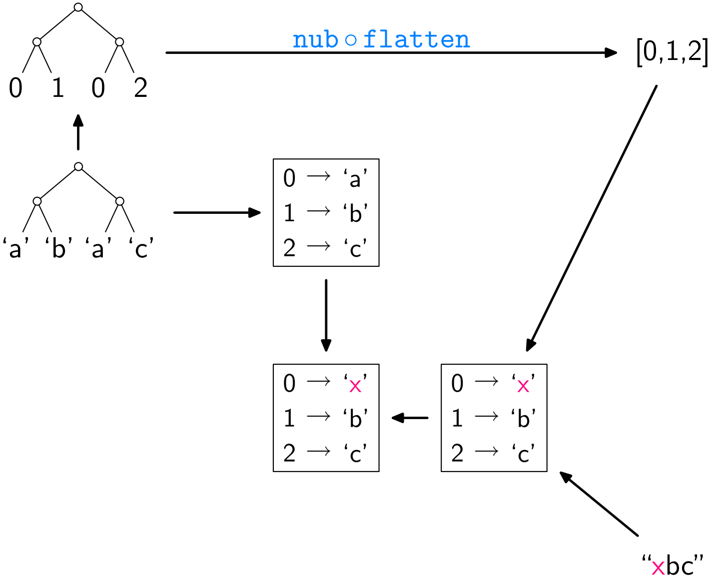


"xbc"

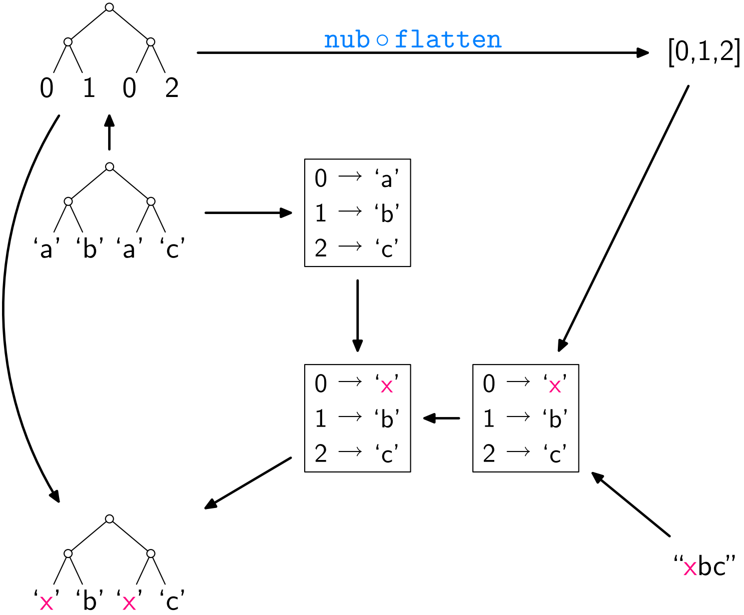
Another Interesting Example (involving Eq type class)



Another Interesting Example (involving Eq type class)



Another Interesting Example (involving Eq type class)



Why `g` (`map` f l) $=$ `map` f (`g` l), intuitively

- `g` $:: [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

Why `g` (`map` f l) $=$ `map` f (`g` l), intuitively

- `g` $:: [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

Why g (map f l) = map f (g l), intuitively

- g :: $[\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.

Why g (map f l) = map f (g l), intuitively

- g :: $[\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.

Why g (map f l) = map f (g l), intuitively

- g :: $[\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.
- The lists (map f l) and l always have equal length.

Why g (map f l) = map f (g l), intuitively

- g :: $[\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.
- The lists (map f l) and l always have equal length.
- g always chooses "the same" elements from (map f l) for output as it does from l,

Why `g` (`map` f l) = `map` f (`g` l), intuitively

- `g` :: $[\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.
- The lists (`map` f l) and l always have equal length.
- `g` always chooses "the same" elements from (`map` f l) for output as it does from l, except that in the former case it outputs their images under f.

Why $\texttt{g}$ ($\texttt{map}$ f l) = $\texttt{map}$ f ($\texttt{g}$ l), intuitively

- $\texttt{g} :: [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.
- The lists ($\texttt{map}$ f l) and l always have equal length.
- $\texttt{g}$ always chooses "the same" elements from ($\texttt{map}$ f l) for output as it does from l, except that in the former case it outputs their images under f.
- $\texttt{g}$ ($\texttt{map}$ f l) is equivalent to $\texttt{map}$ f ($\texttt{g}$ l).

Why g (map f l) = map f (g l), intuitively

- g :: $[\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l.
- The only means for this decision is to inspect the length of l.
- The lists (map f l) and l always have equal length.
- g always chooses "the same" elements from (map f l) for output as it does from l, except that in the former case it outputs their images under f.
- g (map f l) is equivalent to map f (g l).
- That is what we wanted to prove!

Revising Free Theorems

[Wadler, FPCA'89] : for every $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

Revising Free Theorems

[Wadler, FPCA'89] : for every $\mathtt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$,

$$\mathtt{g}\ p\ (\mathtt{map}\ f\ l) \quad = \quad \mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l)$$

- if f strict ($f\ \bot = \bot$).

Revising Free Theorems

[Wadler, FPCA'89] : for every g :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$g\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (g\ (p \circ f)\ l)$$

- if f strict ($f\ \bot = \bot$).

[Johann & V., POPL'04] : in presence of **seq**, if additionally:

- $p \neq \bot$,
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

Revising Free Theorems

[Wadler, FPCA'89] : for every $\texttt{g} :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$,

$$\texttt{g}\ p\ (\texttt{map}\ f\ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l)$$

- if f strict ($f\ \bot = \bot$).

[Johann & V., POPL'04] : in presence of **seq**, if additionally:

- $p \neq \bot$,
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

[Johann & V., I&C'09] : taking finite failures into account

Revising Free Theorems

[Wadler, FPCA'89] : for every g :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$,

$$g\ p\ (\mathtt{map}\ f\ l) \;=\; \mathtt{map}\ f\ (g\ (p \circ f)\ l)$$

- if f strict ($f\ \bot = \bot$).

[Johann & V., POPL'04] : in presence of **seq**, if additionally:

- $p \neq \bot$,
- f total ($\forall x \neq \bot.\ f\ x \neq \bot$).

[Johann & V., I&C'09] : taking finite failures into account

[Stenger & V., TLCA'09] : taking imprecise error semantics into account

⋮