New Applications of Parametricity

Janis Voigtländer

Technische Universität Dresden

ISS-AiPL'09

Recent Applications of Parametricity

Janis Voigtländer

Technische Universität Dresden

ISS-AiPL'09

A Recent Application~~s~~ of Parametricity

Janis Voigtländer

Technische Universität Dresden

ISS-AiPL'09

News About

A Recent Application~~s~~ of Parametricity

Janis Voigtländer

Technische Universität Dresden

ISS-AiPL'09

Parametric Polymorphism in Haskell

A standard function:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism 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:

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

Parametric Polymorphism in Haskell

A standard function:

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

Some invocations:

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

Parametric Polymorphism in Haskell

A standard function:

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

Some invocations:

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

Parametric Polymorphism in Haskell

A standard function:

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

Some invocations:

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

Parametric Polymorphism in Haskell

A standard function:

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

Some invocations:

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

Another Example

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

Another Example

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

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

$$\begin{array}{l} \texttt{reverse} :: [\alpha] \to [\alpha] \\ \texttt{reverse}\ [\,] \qquad\quad = [\,] \\ \texttt{reverse}\ (a : as) = (\texttt{reverse}\ as) + [a] \end{array}$$

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

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

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

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

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

For every choice of f and l:

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

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

Another Example

$$\begin{aligned} \texttt{reverse} &:: [\alpha] \to [\alpha] \\ \texttt{tail} &:: [\alpha] \to [\alpha] \\ \texttt{g} &:: [\alpha] \to [\alpha] \end{aligned}$$

For every choice of f and l:

$$\begin{aligned} \texttt{reverse}\ (\texttt{map}\ f\ l) &= \texttt{map}\ f\ (\texttt{reverse}\ l) \\ \texttt{tail}\ (\texttt{map}\ f\ l) &= \texttt{map}\ f\ (\texttt{tail}\ l) \\ \texttt{g}\ (\texttt{map}\ f\ l) &= \texttt{map}\ f\ (\texttt{g}\ l) \end{aligned}$$

Automatic Generation of Free Theorems

At `http://linux.tcs.inf.tu-dresden.de/~voigt/ft`:

This tool allows to generate free theorems for sublanguages of Haskell as described here.

The source code of the underlying library and a shell-based application using it is available here and here.

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]" or simply "filter":

```
g :: (a -> Bool) -> [a] -> [a]
```

Please choose a sublanguage of Haskell:

- (●) no bottoms (hence no general recursion and no selective strictness)
- ( ) general recursion but no selective strictness
- ( ) general recursion and selective strictness

Please choose a theorem style (without effect in the sublanguage with no bottoms):

- (●) equational
- ( ) inequational

Generate

Automatic Generation of Free Theorems

The theorem generated for functions of the type

```
g :: forall a . (a -> Bool) -> [a] -> [a]
```

in the sublanguage of Haskell with no bottoms is:

```
forall t1,t2 in TYPES, R in REL(t1,t2).
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall (x, y) in R. p x = q y)
   ==> (forall (z, v) in lift{[]}(R).
         (g p z, g q v) in lift{[]}(R))
```

The structural lifting occurring therein is defined as follows:

```
lift{[]}(R)
  = {([], [])}
  u {(x : xs, y : ys) |
       ((x, y) in R) && ((xs, ys) in lift{[]}(R))}
```

Reducing all permissible relation variables to functions yields:

```
forall t1,t2 in TYPES, f :: t1 -> t2.
 forall p :: t1 -> Bool.
  forall q :: t2 -> Bool.
   (forall x :: t1. p x = q (f x))
   ==> (forall y :: [t1]. map f (g p y) = g q (map f y))
```

Export as PDF   Show type instantiations   Enter a new type   Help page

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]

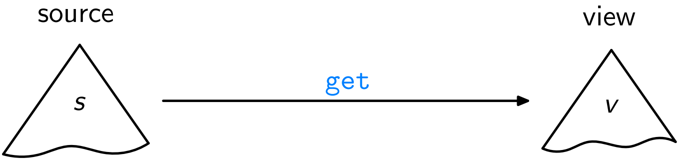
Some Applications

- 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]

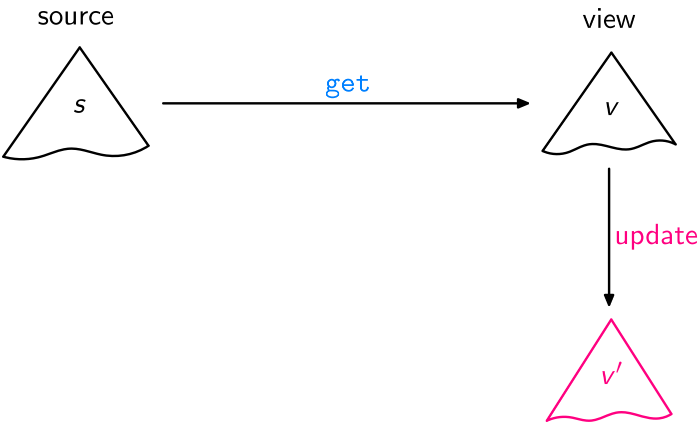
Some Applications

- 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]

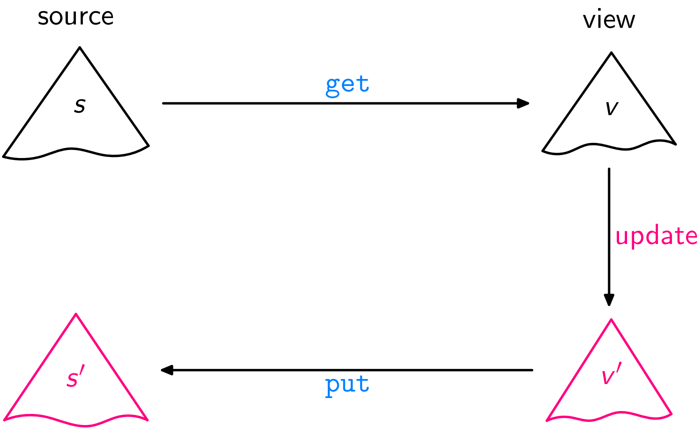
Bidirectional Transformation



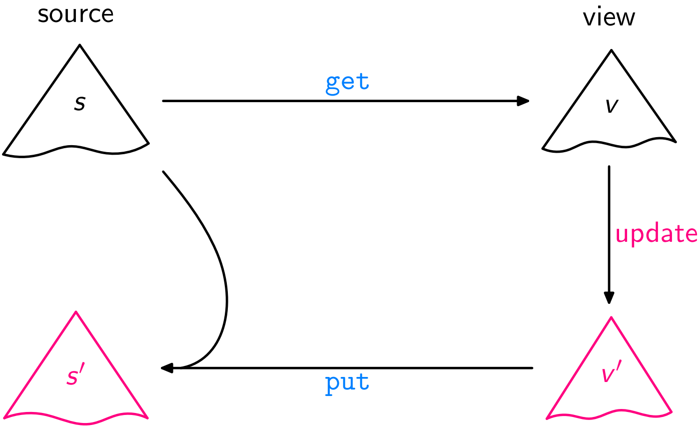
Bidirectional Transformation



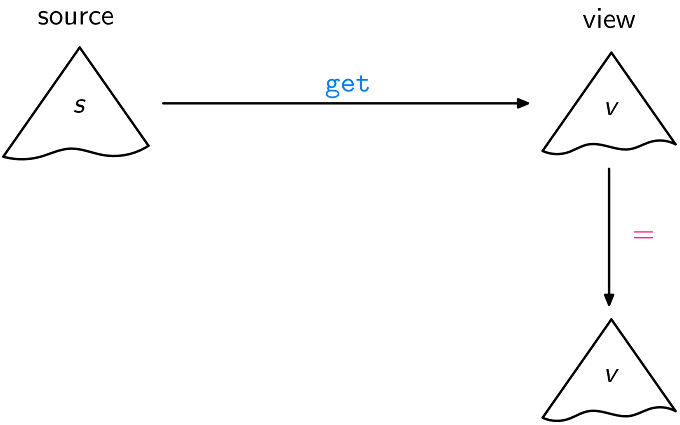
Bidirectional Transformation



Bidirectional Transformation

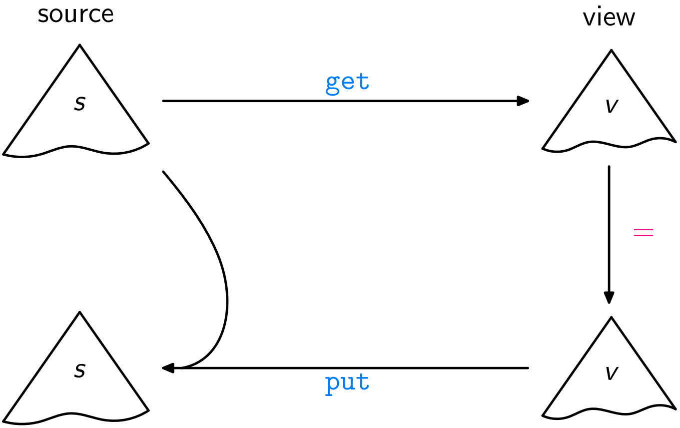


Bidirectional Transformation



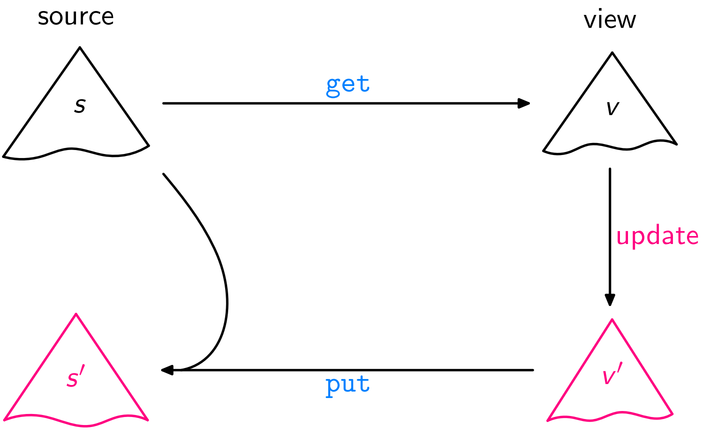
Acceptability / GetPut

Bidirectional Transformation



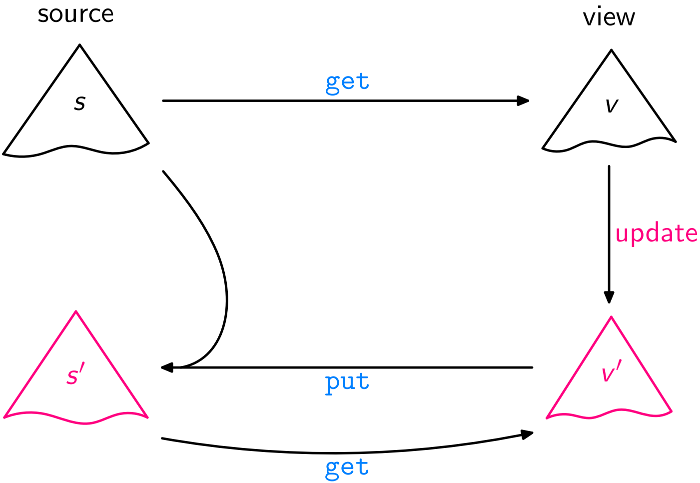
Acceptability / GetPut

Bidirectional Transformation



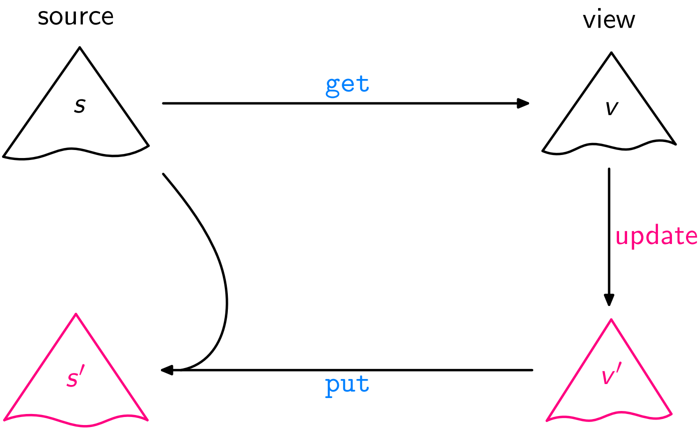
Consistency / PutGet

Bidirectional Transformation

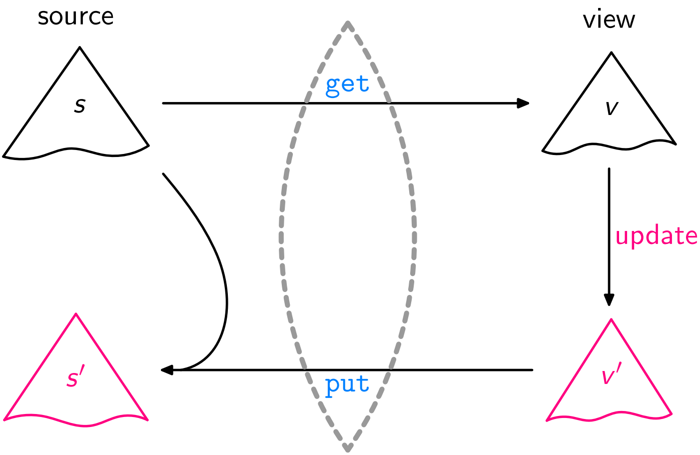


Consistency / PutGet

Bidirectional Transformation



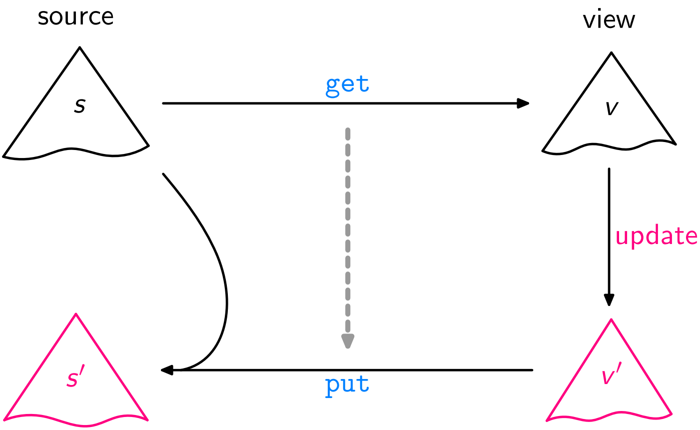
Bidirectional Transformation



Lenses, DSLs

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

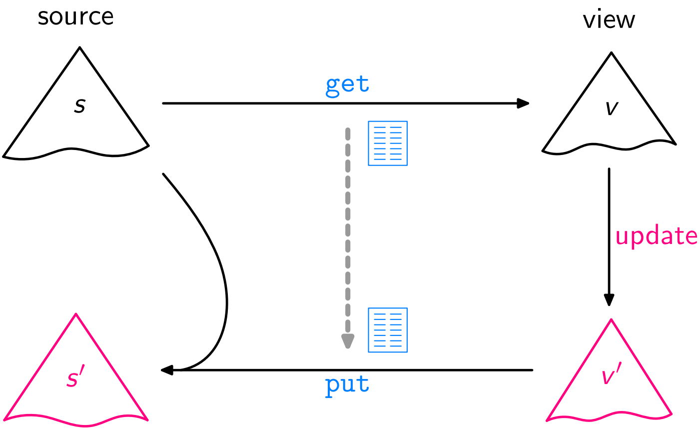
Bidirectional Transformation



Bidirectionalisation

[Matsuda et al., ICFP'07]

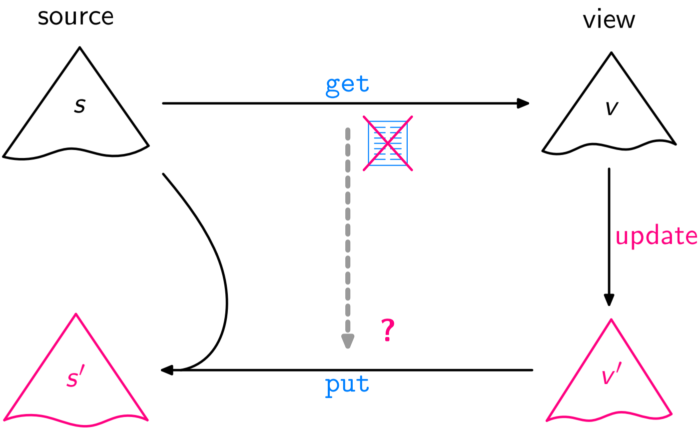
Bidirectional Transformation



Syntactic Bidirectionalisation

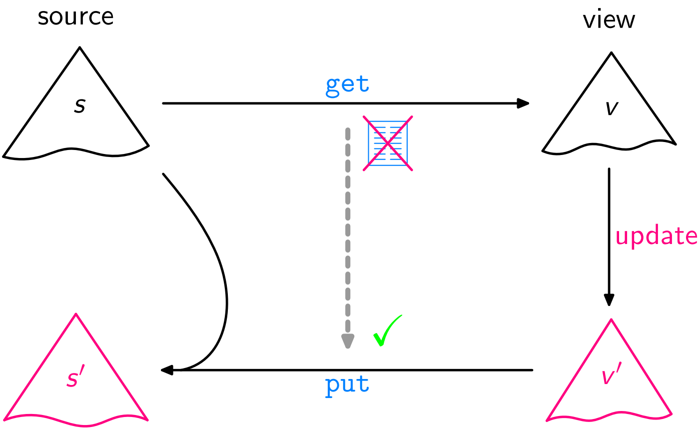
[Matsuda et al., ICFP'07]

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:

$$\text{“abc”} \xrightarrow{\texttt{tail}} \text{“bc”}$$

† “Bidirectionalization for free!”

Semantic Bidirectionalisation

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

Examples:

$$\begin{tikzcd}
\text{“abc”} \arrow[r, "\texttt{tail}"] & \text{“bc”} \arrow[d, "\text{update}"] \\
 & \text{“de”}
\end{tikzcd}$$

† “Bidirectionalization for free!”

Semantic Bidirectionalisation

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

Examples:

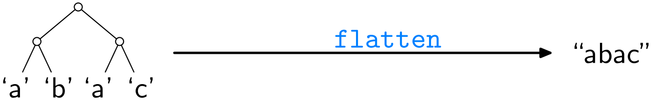
$$
\begin{tikzcd}
\text{“abc”} \arrow[r, "\texttt{tail}"] \arrow[d, bend left] & \text{“bc”} \arrow[d, "\text{update}"] \\
\text{“ade”} & \text{“de”} \arrow[l, "\texttt{bff tail}"]
\end{tikzcd}
$$

† “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, ...

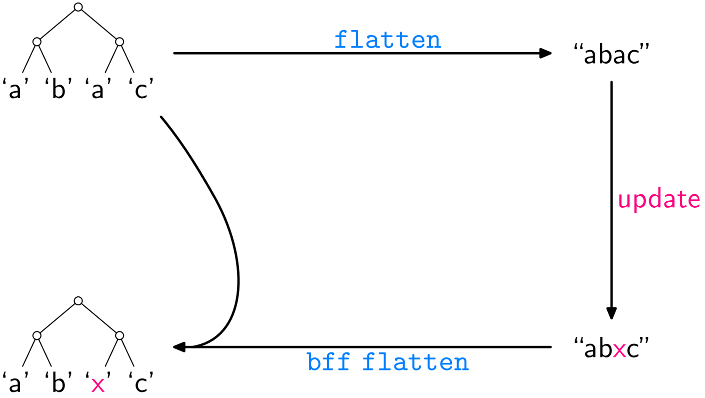
Examples:

† "Bidirectionalization for free!"

Semantic Bidirectionalisation

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

Examples:

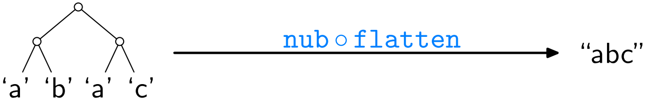


† "Bidirectionalization for free!"

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, ...

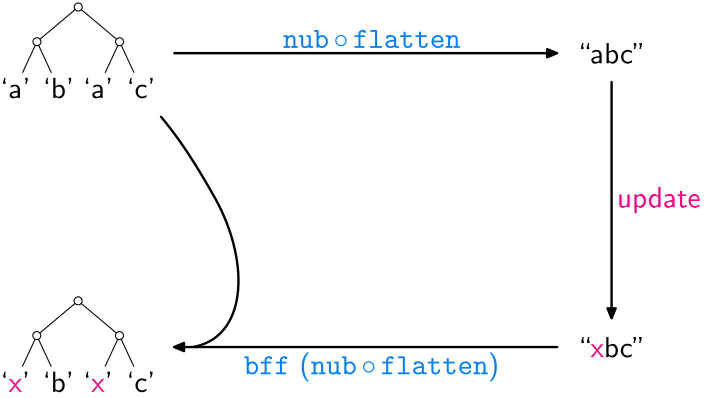
Examples:

† “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

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

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

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

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

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

we have

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

for arbitrary f and l.

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l.

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

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

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l.

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

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

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l.

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

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

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

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l.

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

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

for every $\mathtt{get} :: [\alpha] \to [\alpha]$.

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

In general, given

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

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

In general, given

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

define a V^C and

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

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

In general, given

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

define a V^C and

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

such that

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

is injective

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

In general, given

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

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:

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

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

In general, given

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

define a V^C and

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

such that

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

is injective and has an inverse

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

Then:

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

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

The Constant-Complement Approach

For our setting,

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

what should be V^C and

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

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,

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

what should be V^C and

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

To make

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

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

Candidates:

1. length of the source list

The Constant-Complement Approach

For 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

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

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

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

The Complement Function

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

For example:

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

The Complement Function

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

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

For example:

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

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

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:

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

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

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

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

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

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

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

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

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

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

For example:

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

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

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

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

For example:

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

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

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

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

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

For example:

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

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

To prove formally:

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

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

Altogether:

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

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

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

$$\begin{array}{l}
\texttt{put} :: [\alpha] \to [\alpha] \to [\alpha] \\
\texttt{put}\ s\ v' = \texttt{inv}\ (v', \texttt{compl}\ s)
\end{array}$$

“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`:

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

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

"Fusion"

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

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

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

“Fusion”

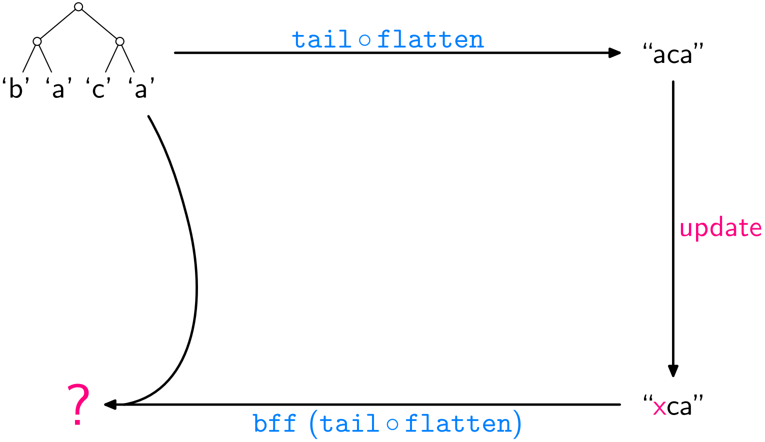
Inlining `compl` and `inv` into `put`:

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

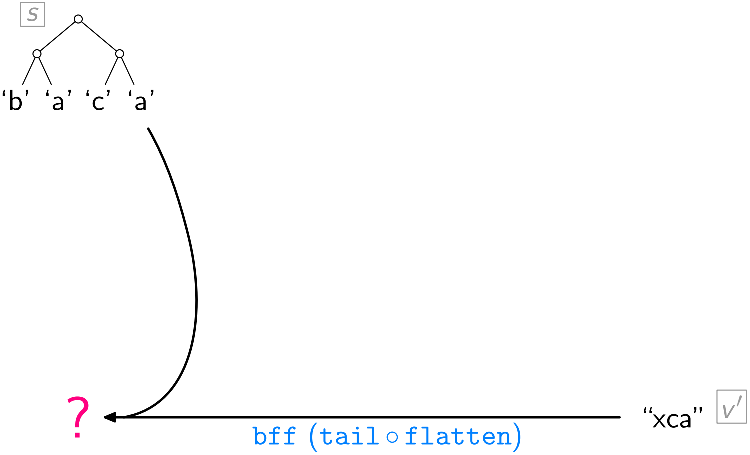
$$
\begin{aligned}
&\texttt{assoc}\ [\,]\qquad\ \ [\,] \qquad\ \ = [\,] \\
&\texttt{assoc}\ (i : is)\ (b : bs) = \textbf{let}\ m = \texttt{assoc}\ is\ bs \\
&\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ \textbf{case}\ \texttt{lookup}\ i\ m\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{Nothing}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

Actual code only slightly more elaborate!

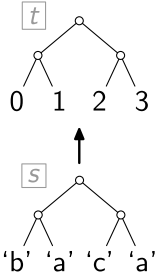
The Resulting Bidirectionalisation Method in Action



The Resulting Bidirectionalisation Method in Action

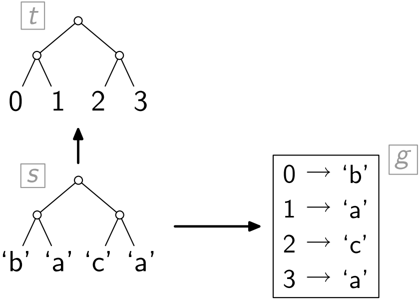


The Resulting Bidirectionalisation Method in Action



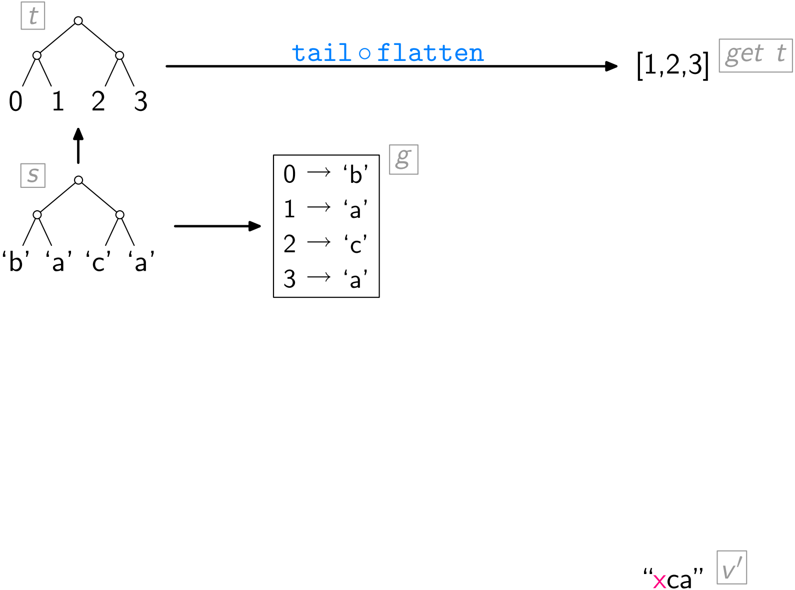
"xca" v'

The Resulting Bidirectionalisation Method in Action

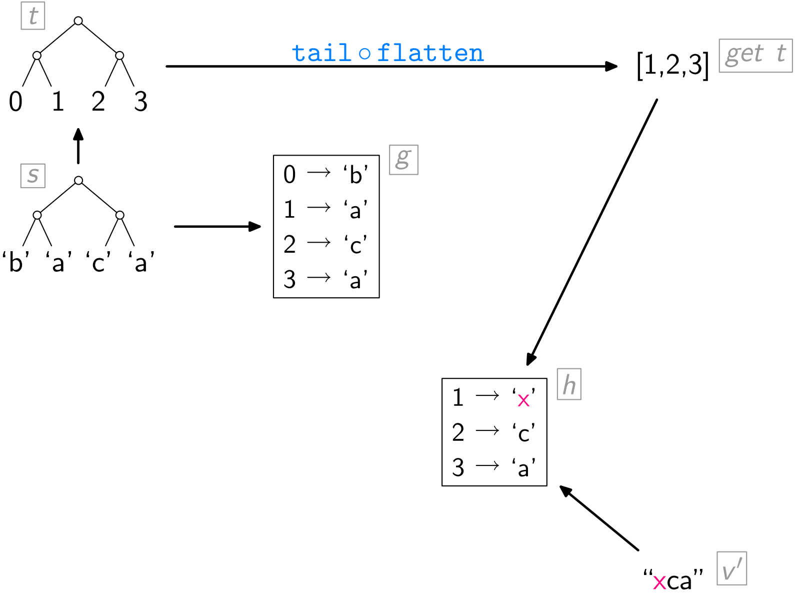


"xca" v'

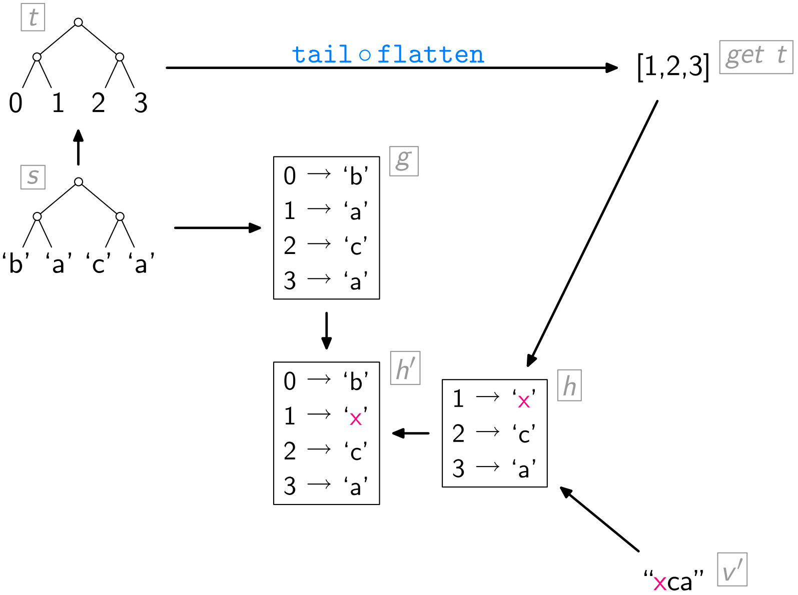
The Resulting Bidirectionalisation Method in Action



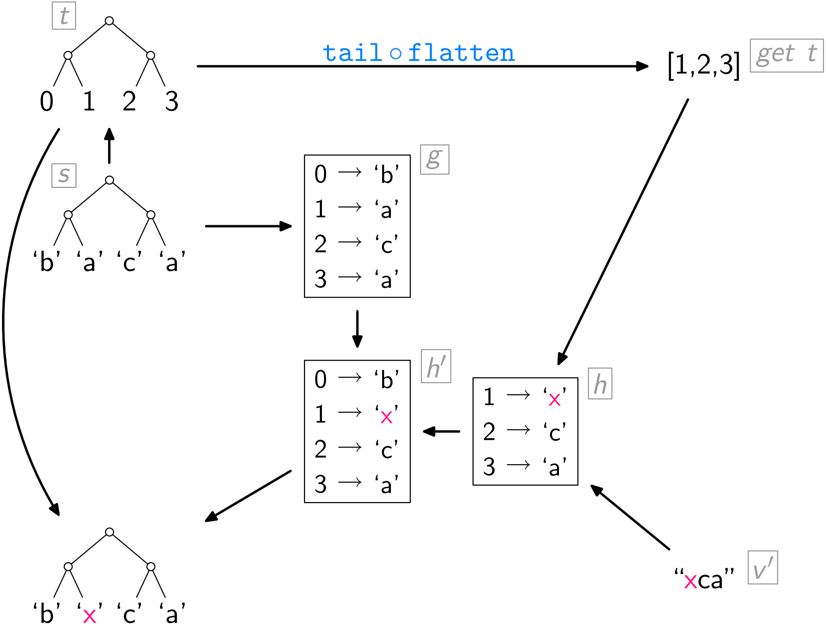
The Resulting Bidirectionalisation Method in Action



The Resulting Bidirectionalisation Method in Action



The Resulting Bidirectionalisation Method in Action



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:

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

Assuming Shape-Injectivity

So assume there is a function

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

with, for every source list s,

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

Assuming Shape-Injectivity

So assume there is a function

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

with, for every source list s,

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

Then:

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

Assuming Shape-Injectivity

So assume there is a function

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

with, for every source list s,

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

Then:

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

Assuming Shape-Injectivity

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

Assuming Shape-Injectivity

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

Assuming Shape-Injectivity

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

But how to obtain `shapeInv` ???

Assuming Shape-Injectivity

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

But how to obtain `shapeInv` ???

One possibility: provided by user.

Assuming Shape-Injectivity

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

But how to obtain `shapeInv` ???

One possibility: provided by user.

Another possibility: determined statically (dependent types?).

Assuming Shape-Injectivity

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

But how to obtain `shapeInv` ???

One possibility: provided by user.

Another possibility: determined statically (dependent types?).

Just for experimentation:

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

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” $= [(0, \text{'a'})]$

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" $= [(0, \text{'a'})]$

But not so in others:

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

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

Not Quite There, Yet

Works quite nicely in some cases:

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

But not so in others:

$\texttt{get} = \texttt{init} \quad \leadsto \quad \texttt{put}$ "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

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

Eliminating Indices

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

Eliminating Indices

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

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

Eliminating Indices

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

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

Eliminating Indices

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

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

Now:

$$\texttt{get} = \texttt{init} \quad \leadsto \quad \texttt{put}\ \text{"abcde"}\ \ \text{"xyz"} = \text{"xyze"}$$

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

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

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:

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

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:

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

More Examples

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

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

Then:

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

However:

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

More Examples

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

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

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.

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 III

J.C. Reynolds.
Types, abstraction and parametric polymorphism.
In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

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.

References IV

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

- 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.