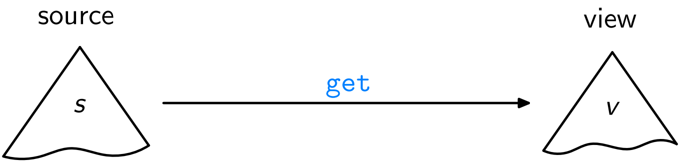
Semantic Bidirectionalization and the Constant-Complement Perspective

Janis Voigtländer

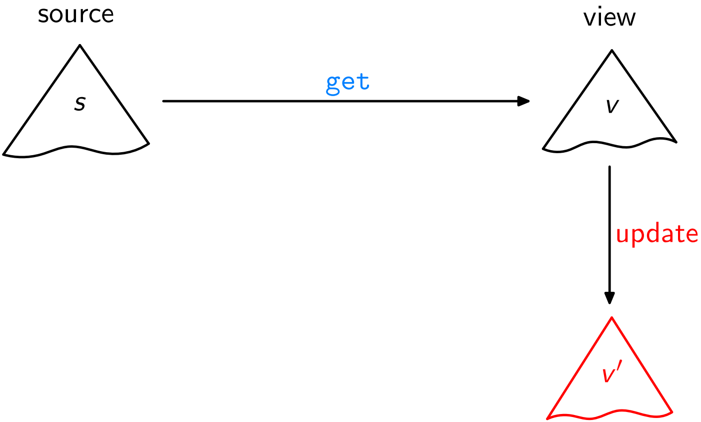
University of Bonn

BT-in-ABC'10

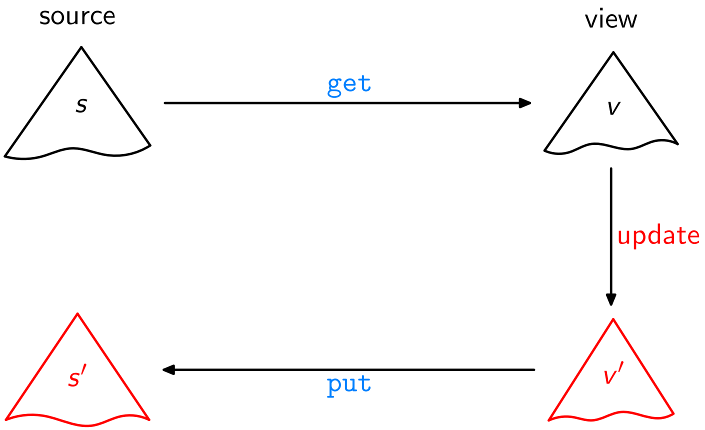
Bidirectional Transformation



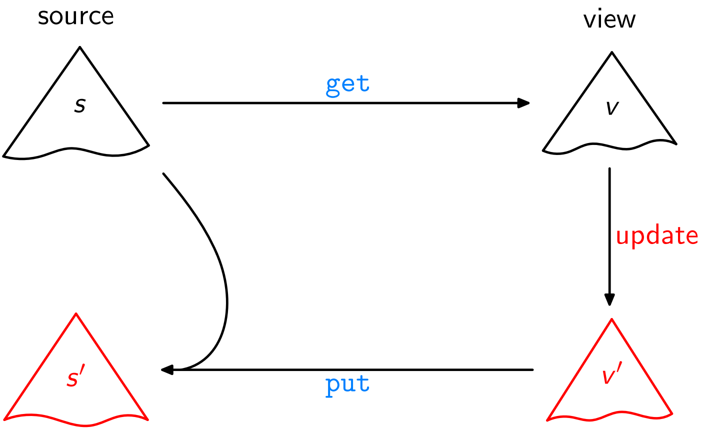
Bidirectional Transformation



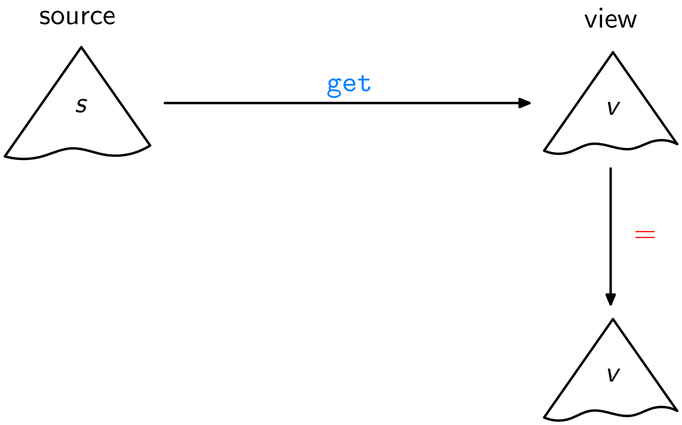
Bidirectional Transformation



Bidirectional Transformation

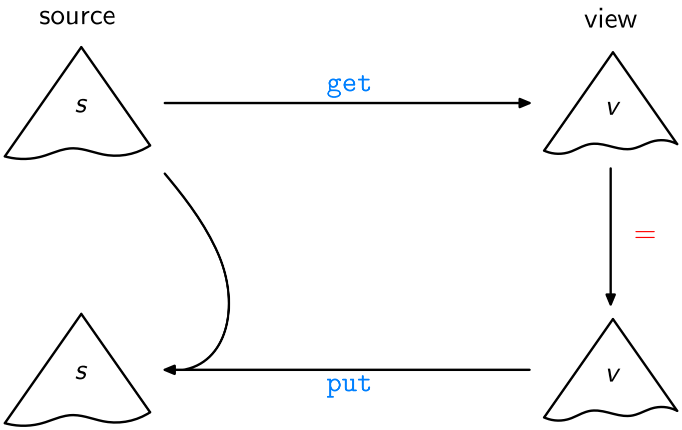


Bidirectional Transformation



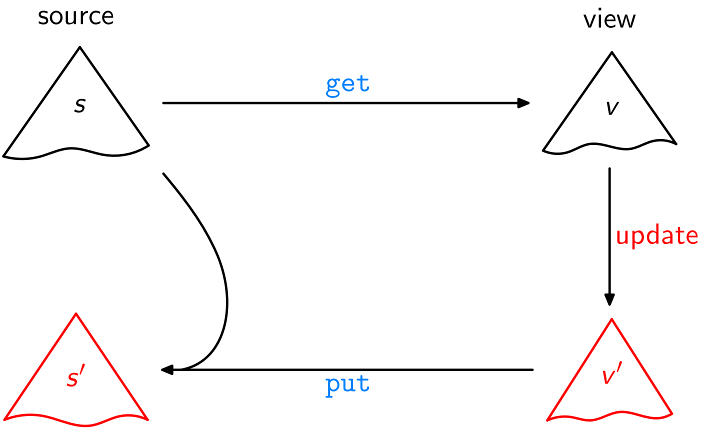
Acceptability / GetPut

Bidirectional Transformation



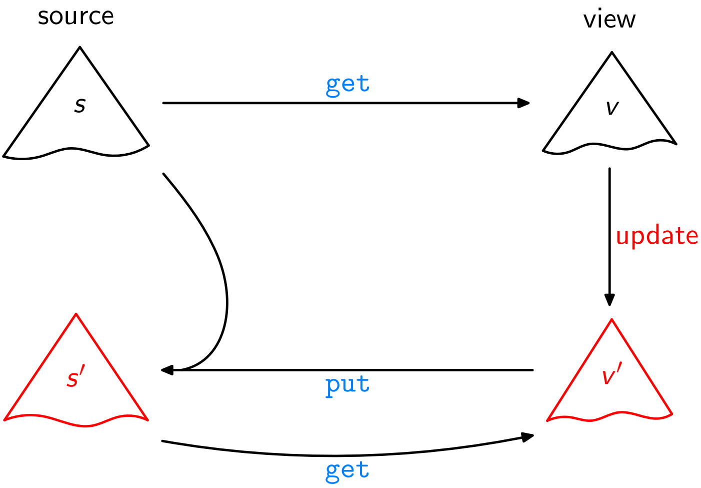
Acceptability / GetPut

Bidirectional Transformation



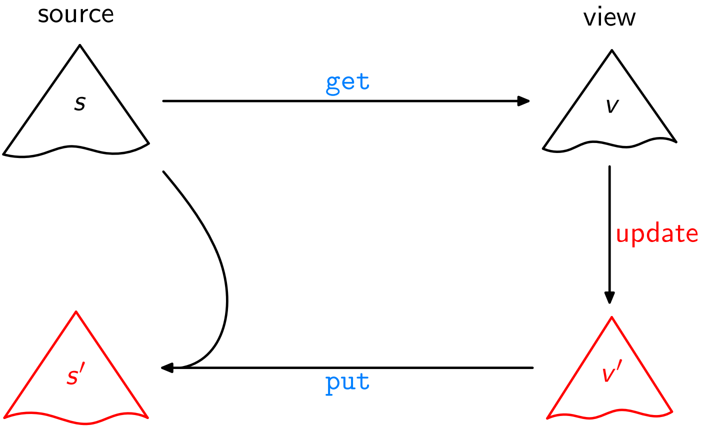
Consistency / PutGet

Bidirectional Transformation

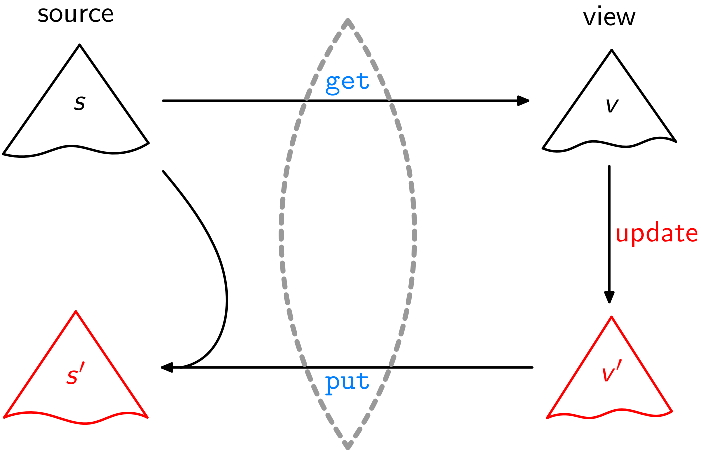


Consistency / PutGet

Bidirectional Transformation



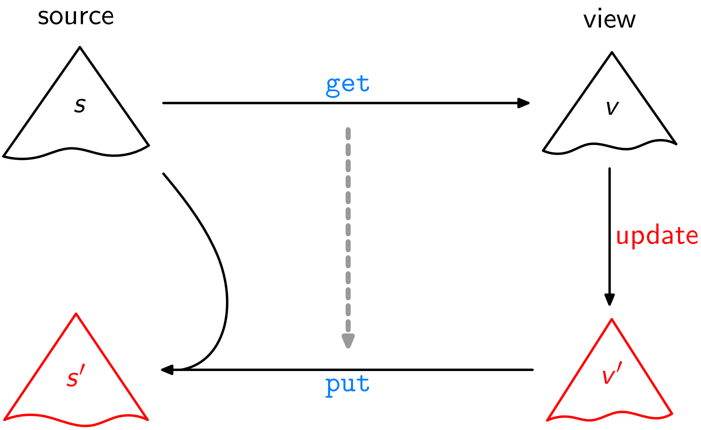
Bidirectional Transformation



Lenses, DSLs

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

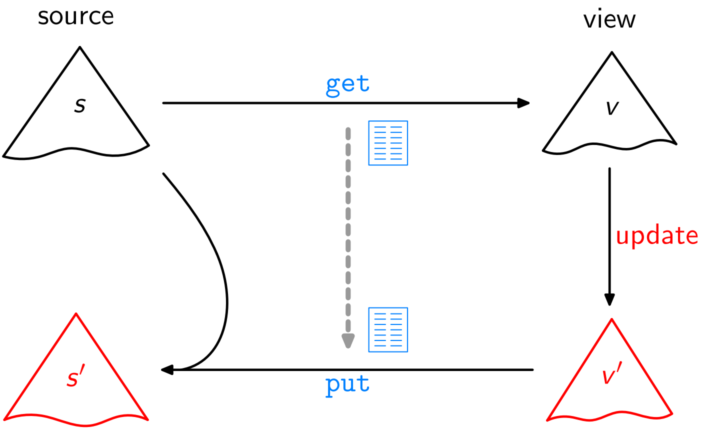
Bidirectional Transformation



Bidirectionalization

[Matsuda et al., ICFP'07]

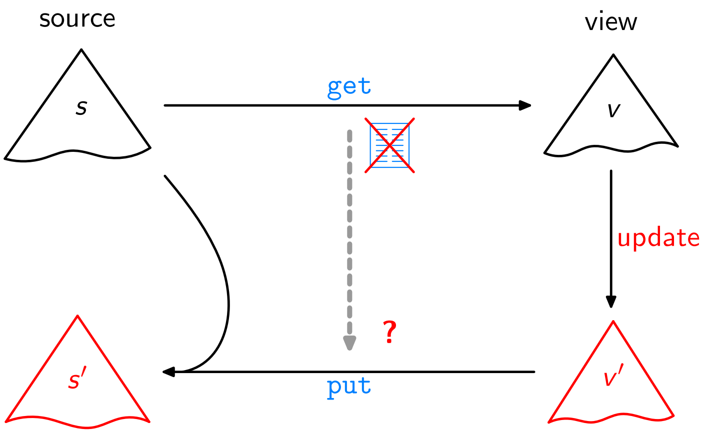
Bidirectional Transformation



Syntactic Bidirectionalization

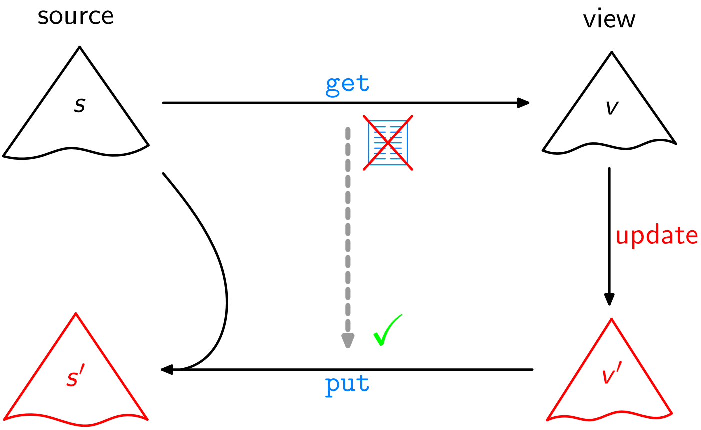
[Matsuda et al., ICFP'07]

Bidirectional Transformation



Semantic Bidirectionalization

Bidirectional Transformation



Semantic Bidirectionalization

[V., POPL'09]

Semantic Bidirectionalization

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

Semantic Bidirectionalization

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

[†] "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:

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

† “Bidirectionalization for free!”

Semantic Bidirectionalization

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

Examples:

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

† “<u>B</u>idirectionalization <u>f</u>or <u>f</u>ree!”

Semantic Bidirectionalization

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

Examples:

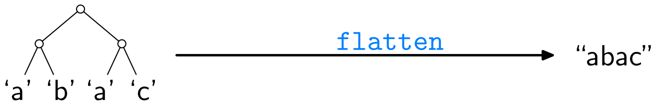
$$\begin{tikzcd}
\text{``abc''} \arrow[rrr, "\texttt{tail}"] \arrow[d, bend left] & & & \text{``bc''} \arrow[d, "\text{update}"] \\
\text{``ade''} & & & \text{``de''} \arrow[lll, "\texttt{bff tail}"]
\end{tikzcd}$$

† "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:



† "Bidirectionalization for free!"

Semantic Bidirectionalization

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

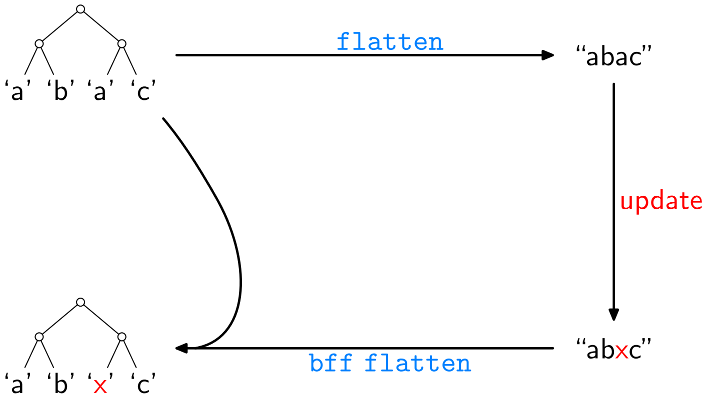
Examples:

† "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:

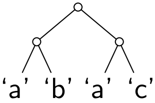


† "Bidirectionalization for free!"

Semantic Bidirectionalization

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

Examples:



$\xrightarrow{\texttt{nub} \circ \texttt{flatten}}$ "abc"

† "Bidirectionalization for free!"

Semantic Bidirectionalization

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

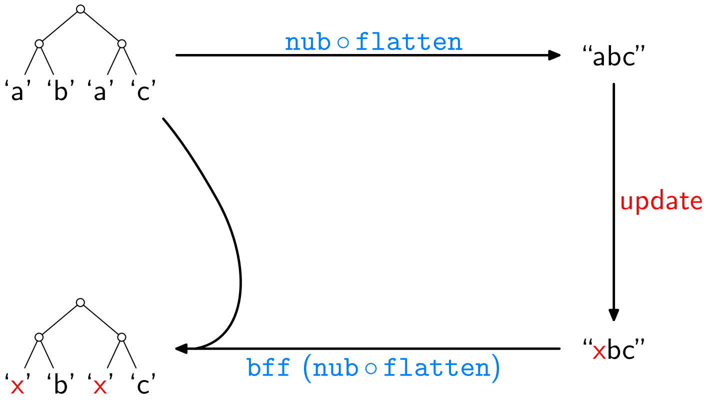
Examples:

[†] “Bidirectionalization for free!”

Semantic Bidirectionalization

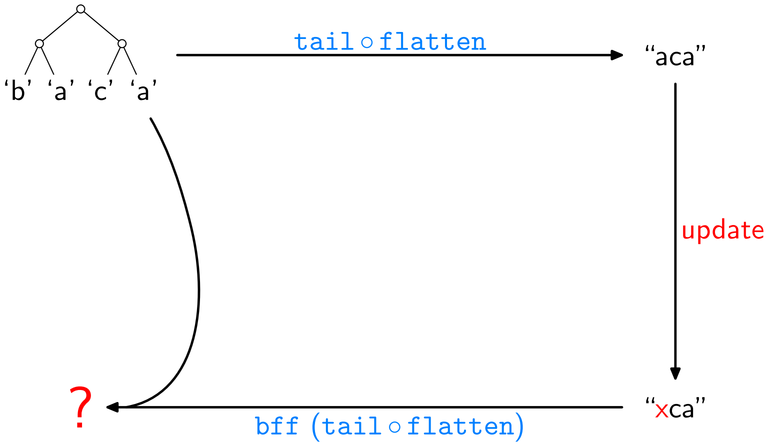
Aim: Write a higher-order function `bff`† such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

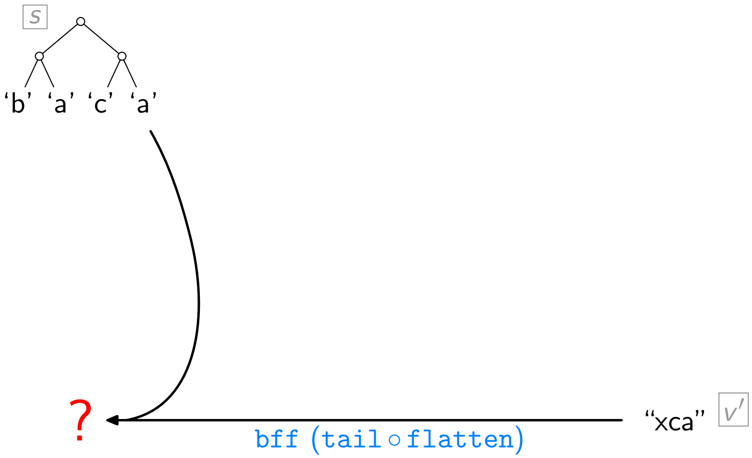


† "Bidirectionalization for free!"

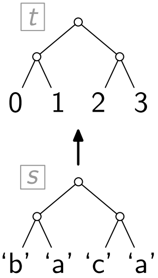
Overview of the Bidirectionalization Method



Overview of the Bidirectionalization Method

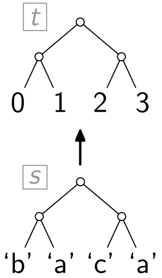


Overview of the Bidirectionalization Method



"xca" $\boxed{v'}$

Overview of the Bidirectionalization Method

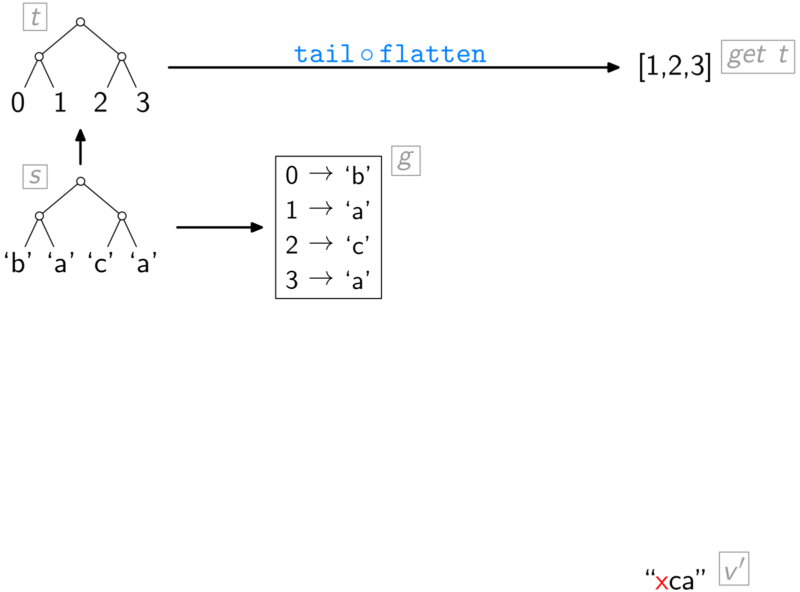


0	$\rightarrow$	‘b’
1	$\rightarrow$	‘a’
2	$\rightarrow$	‘c’
3	$\rightarrow$	‘a’

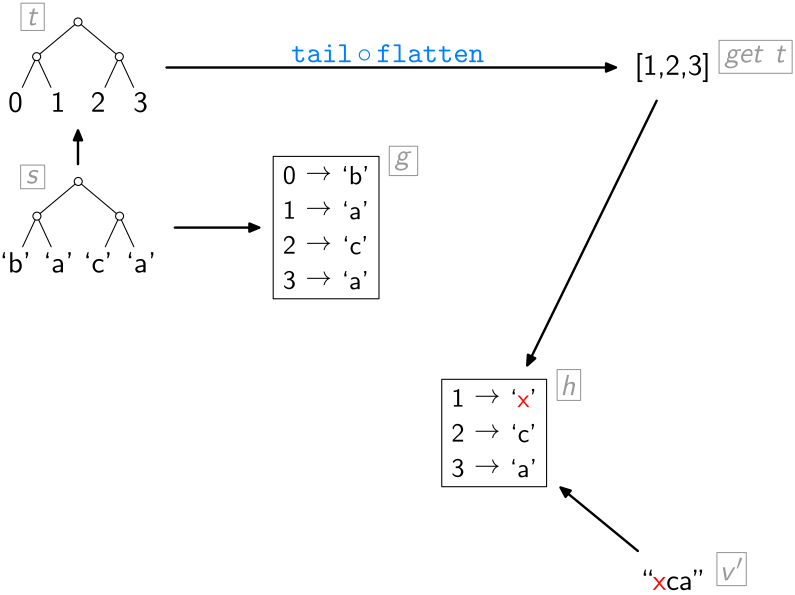
g

“xca” v'

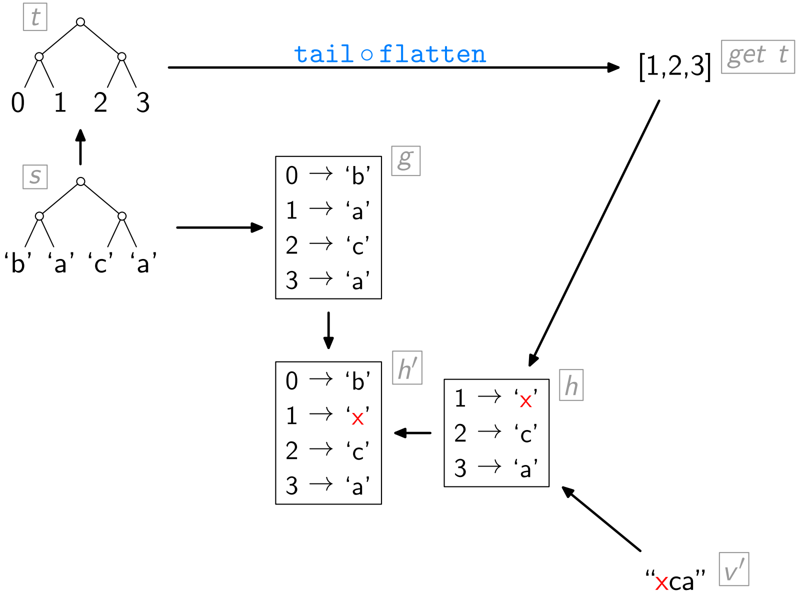
Overview of the Bidirectionalization Method



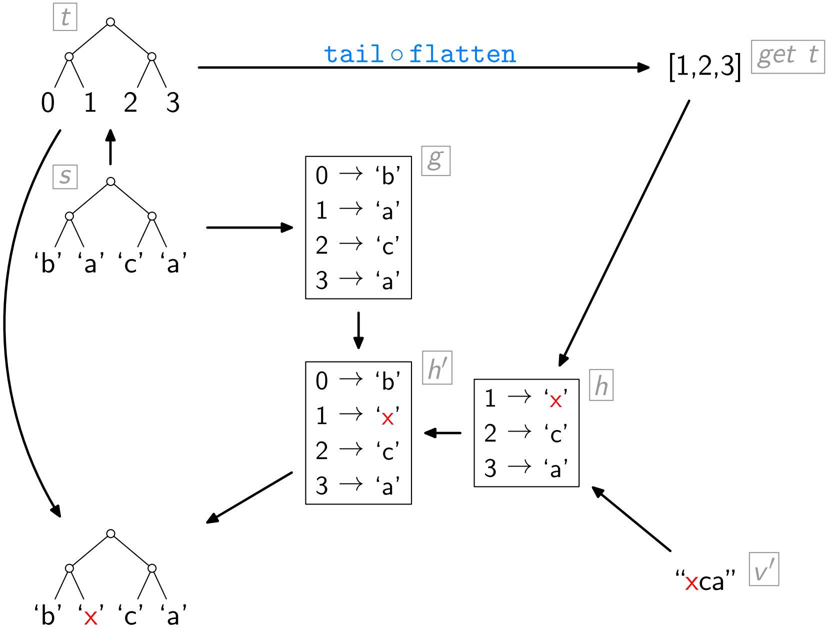
Overview of the Bidirectionalization Method



Overview of the Bidirectionalization Method



Overview of the Bidirectionalization Method



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

In general, given

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

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

In general, given

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

define a V^C and

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

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

In general, given

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

define a V^C and

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

such that

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

is injective

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

In general, given

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

define a V^C and

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

such that

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

is injective and has an inverse

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

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

In general, given

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

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:

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

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

In general, given

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

define a V^C and

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

such that

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

is injective and has an inverse

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

Then:

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

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

The Constant-Complement Approach

For a very simple setting,

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

what should be V^C and

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

The Constant-Complement Approach

For a very simple setting,

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

what should be V^C and

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

To make

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

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

The Constant-Complement Approach

For a very simple setting,

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

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 a very simple 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 a very simple setting,

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

what should be V^C and

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

To make

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

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

Candidates:

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

For the moment, be maximally conservative.

The Complement Function

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

The Complement Function

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

For example:

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

The Complement Function

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

For example:

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

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

The Complement Function

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

For example:

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

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

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

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

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

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

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

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

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

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

For example:

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

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

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

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

For example:

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

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

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

Correctness

To prove formally:

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

Correctness

To prove formally:

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

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

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

we have, for arbitrary f and l,

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

Correctness

To prove formally:

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

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

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

we have, for arbitrary f and l,

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

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

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

Correctness

To prove formally:

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

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

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

we have, for arbitrary f and l,

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

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

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

Correctness

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

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

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

we have, for arbitrary f and l,

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

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

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

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

Correctness

To prove formally:

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

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

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

we have, for arbitrary f and l,

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

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

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

Altogether, So Far:

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

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

“Fusion”

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

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

"Fusion"

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

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

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

"Fusion"

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

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

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

"Fusion"

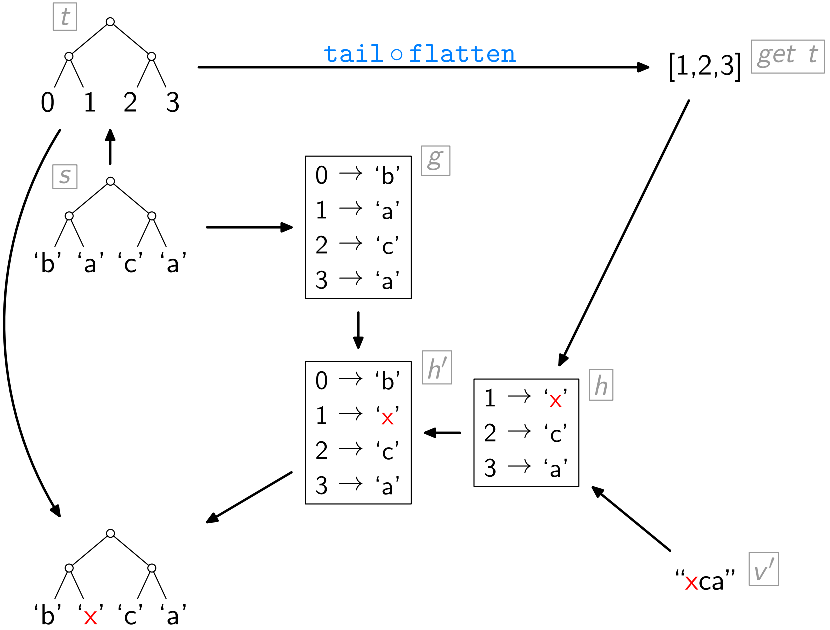
Inlining `compl` and `inv` into `put`, plus some clever rewriting:

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

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

Actual code only slightly more elaborate!

Overview of the Bidirectionalization Method



Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.

Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.
- For example, `bff tail` "abcde" "xyz" ...

Extending the Technique

Major Problem:

- Shape-affecting updates lead to failure.
- For example, `bff tail` "abcde" "xyz" ...

Analysis as to Why:

- Our approach to making

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

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

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

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

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

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

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

- Being maximally conservative this way often does not "collapse enough".
- For example:

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

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

with, for every source list s,

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

Then:

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

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:

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

Assuming Shape-Injectivity

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

Assuming Shape-Injectivity

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

Assuming Shape-Injectivity

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

But how to obtain `shapeInv` ???

Assuming Shape-Injectivity

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

But how to obtain $\texttt{shapeInv}$???

Just for experimentation:

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

Some Tests

Works quite nicely in some cases:

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

Some Tests

Works quite nicely in some cases:

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

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

Some Tests

Works quite nicely in some cases:

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

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

But not so in others:

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

Some Tests

Works quite nicely in some cases:

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

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

But not so in others:

$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{put}$ "abcde" "abc" = "abc"

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

Some Tests

Works quite nicely in some cases:

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

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

But not so in others:

$\texttt{get} = \texttt{take}\ 3 \quad \rightsquigarrow \quad \texttt{put}$ "abcde" "abc" = "abc"

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

Better:

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

Conclusion

[V., POPL'09]:

- very lightweight, easy access to bidirectionality
- full treatment of equality and ordering constraints
- proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Conclusion

[V., POPL'09]:

- very lightweight, easy access to bidirectionality
- full treatment of equality and ordering constraints
- proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Here:

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

Conclusion

[V., POPL'09]:

- very lightweight, easy access to bidirectionality
- full treatment of equality and ordering constraints
- proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Here:

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

Outlook:

- . . . could also be a way to inject/exploit "user knowledge"

Conclusion

[V., POPL'09]:

- very lightweight, easy access to bidirectionality
- full treatment of equality and ordering constraints
- proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Here:

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

Outlook:

- ... could also be a way to inject/exploit "user knowledge"
- combination with syntactic bidirectionalization à la [Matsuda et al., ICFP'07] is work in progress

Conclusion

[V., POPL'09]:

- very lightweight, easy access to bidirectionality
- full treatment of equality and ordering constraints
- proofs, using free theorems and equational reasoning
- a datatype-generic account of the whole story

Here:

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

Outlook:

- ... could also be a way to inject/exploit "user knowledge"
- combination with syntactic bidirectionalization à la [Matsuda et al., ICFP'07] is work in progress
- efficiency issues untackled so far, ...

References I

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

J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt.
Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem.
ACM Transactions on Programming Languages and Systems, 29(3):17, 2007.

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

References II

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

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