Yesterday:

$$
\begin{array}{l}
\texttt{put} :: [\alpha] \to [\alpha] \to [\alpha] \\
\texttt{put}\ s\ v = \mathbf{let}\ \begin{array}[t]{lcl} n & = & (\texttt{length}\ s) - 1 \\ s' & = & [0..n] \\ g & = & \texttt{zip}\ s'\ s \\ h & = & \texttt{zip}\ (\texttt{get}\ s')\ v \\ h' & = & h +\!\!+\ g \end{array} \\
\qquad\qquad\quad \mathbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'
\end{array}
$$

For the full story, see:

- J. Voigtländer.
  Bidirectionalization for Free!
  In *Principles of Programming Languages, Proceedings*.
  ACM Press, 2009.

A Constant-Complement Perspective on Bidirectionalization for Free

Janis Voigtländer

Technische Universität Dresden

GRACE-BX'08

The Constant-Complement Approach

In general, given

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

The Constant-Complement Approach

In general, given

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

define a V' and

$$\texttt{compl} :: S \to V'$$

The Constant-Complement Approach

In general, given

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

define a V' and

$$\texttt{compl} :: S \to V'$$

such that

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

is injective

The Constant-Complement Approach

In general, given

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

define a V' and

$$\texttt{compl} :: S \rightarrow V'$$

such that

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

is injective and has an inverse

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

The Constant-Complement Approach

In general, given

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

define a V' and

$$\texttt{compl} :: S \to V'$$

such that

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

is injective and has an inverse

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

Then:

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

The Constant-Complement Approach

In general, given

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

define a V' and

$$\texttt{compl} :: S \to V'$$

such that

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

is injective and has an inverse

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

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' and

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

The Constant-Complement Approach

For our setting,

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

what should be V' and

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

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' and

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

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,

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

what should be V' and

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

To make

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

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

Candidates:

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

The Constant-Complement Approach

For our setting,

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

what should be V' and

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

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 with those.

The Complement Function

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

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

The Complement Function

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

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

For example:

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

The Complement Function

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

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

For example:

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

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

The Complement Function

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

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

For example:

$$\begin{aligned}
\texttt{get} = \texttt{tail} &\quad\rightsquigarrow\quad \texttt{compl}\ \text{“abcde”} = (5, [(0, \text{'a'})])\\
\texttt{get} = \texttt{take}\ 3 &\quad\rightsquigarrow\quad \texttt{compl}\ \text{“abcde”} = (5, [(3, \text{'d'}), (4, \text{'e'})])\\
\texttt{get} = \texttt{reverse} &\quad\rightsquigarrow\quad \texttt{compl}\ \text{“abcde”} = (5, [\,])
\end{aligned}$$

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

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

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

$$
\begin{aligned}
&\texttt{inv} :: [\alpha] \rightarrow (\mathsf{Int}, \mathsf{IntMap}\ \alpha) \rightarrow [\alpha] \\
&\texttt{inv}\ v\ (n+1, g') = \textbf{let}\ s' \ = [0..n] \\
&\qquad\qquad\qquad\qquad\quad\ \ h \ \ = \texttt{zip}\ (\texttt{get}\ s')\ v \\
&\qquad\qquad\qquad\qquad\quad\ \ h' \ = h +\!\!+\, g' \\
&\qquad\qquad\qquad\qquad\ \ \textbf{in}\ \ \texttt{map}\ (\lambda i \rightarrow \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'
\end{aligned}
$$

For example:

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

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

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

For example:

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

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

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

$$
\begin{array}{l}
\mathtt{inv} :: [\alpha] \to (\mathsf{Int}, \mathsf{IntMap}\ \alpha) \to [\alpha] \\
\mathtt{inv}\ v\ (n+1, g') = \textbf{let}\ s' \ = [0..n] \\
\qquad\qquad\qquad\qquad\quad\ \ h\ \ = \mathtt{zip}\ (\mathtt{get}\ s')\ v \\
\qquad\qquad\qquad\qquad\quad\ \ h' = h +\!\!+\ g' \\
\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \mathtt{map}\ (\lambda i \to \mathtt{fromJust}\ (\mathtt{lookup}\ i\ h'))\ s'
\end{array}
$$

For example:

$$
\begin{array}{lcl}
\mathtt{get} = \mathtt{tail} & \rightsquigarrow & \mathtt{inv}\ \text{“bcde”}\ (5, [(0, \text{'a'})]) = \text{“abcde”} \\
\mathtt{get} = \mathtt{take}\ 3 & \rightsquigarrow & \mathtt{inv}\ \text{“xyz”}\ (5, [(3, \text{'d'}), (4, \text{'e'})]) = \text{“xyzde”}
\end{array}
$$

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

Altogether:

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

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

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

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

“Fusion”

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

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

“Fusion”

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

$$
\begin{array}{l}
\mathtt{put} :: [\alpha] \to [\alpha] \to [\alpha] \\
\mathtt{put}\ s\ v = \textbf{let}\ \begin{array}[t]{lcl}
n & = & (\mathtt{length}\ s) - 1 \\
s' & = & [0..n] \\
g & = & \mathtt{zip}\ s'\ s \\
 & & \\
h & = & \mathtt{zip}\ (\mathtt{get}\ s')\ v \\
h' & = & h +\!\!+\ {\color{red} g}
\end{array} \\
\qquad\qquad\quad \textbf{in}\ \ \mathtt{map}\ (\lambda i \to \mathtt{fromJust}\ (\mathtt{lookup}\ i\ h'))\ s'
\end{array}
$$

"Fusion"

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

$$
\begin{array}{l}
\mathtt{put} :: [\alpha] \to [\alpha] \to [\alpha] \\
\mathtt{put}\ s\ v = \mathbf{let}\ \begin{array}[t]{ll}
n & = (\mathtt{length}\ s) - 1 \\
s' & = [0..n] \\
g & = \mathtt{zip}\ s'\ s \\
g' & = \mathtt{filter}\ (\lambda(i, \_) \to \mathtt{notElem}\ i\ (\mathrm{get}\ s'))\ g \\
h & = \mathtt{zip}\ (\mathrm{get}\ s')\ v \\
h' & = h +\!\!+\ g'
\end{array} \\
\qquad\qquad\quad \mathbf{in}\ \ \mathtt{map}\ (\lambda i \to \mathtt{fromJust}\ (\mathtt{lookup}\ i\ h'))\ s'
\end{array}
$$

“Fusion”

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

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

But the “decomposed” perspective via `compl` and `inv` better enables us to develop extensions of the technique!

Assuming Shape-Injectivity

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

Assuming Shape-Injectivity

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

But for many `get`-functions it is impossible to have two sources of different lengths whose views have the same length.

Assuming Shape-Injectivity

Our approach to making

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

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

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

But for many `get`-functions it is impossible to have two sources of different lengths whose views have the same length.

In these cases, recording the length of the original source leads to unnecessary restrictions.

For example:

$\mathtt{get} = \mathtt{tail} \quad \rightsquigarrow \quad \mathtt{put}$ "abcde" "xyz" fails

Assuming Shape-Injectivity

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

But for many `get`-functions it is impossible to have two sources of different lengths whose views have the same length.

In these cases, recording the length of the original source leads to unnecessary restrictions.

For example:

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

Assuming Shape-Injectivity

So assume there is a function

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

with, for every source list s,

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

Assuming Shape-Injectivity

So assume there is a function

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

with, for every source list s,

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

Then:

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

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:

$$\begin{array}{l}
\texttt{compl} :: [\alpha] \to \qquad \mathsf{IntMap}\ \alpha \\
\texttt{compl}\ s = \textbf{let}\ \begin{array}{ll} n & = (\texttt{length}\ s) - 1 \\ s' & = [0..n] \\ g & = \texttt{zip}\ s'\ s \\ g' & = \texttt{filter}\ (\lambda(i, _) \to \texttt{notElem}\ i\ (\texttt{get}\ s'))\ g \end{array} \\
\qquad\qquad \textbf{in} \qquad\quad g'
\end{array}$$

Assuming Shape-Injectivity

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

Assuming Shape-Injectivity

$$
\begin{array}{llll}
\texttt{inv} :: [\alpha] \to & \mathsf{IntMap}\ \alpha & \to [\alpha] \\
\texttt{inv}\ v & g' & = \textbf{let}\ n = \texttt{shapeInv}\ (\texttt{length}\ v) - 1 \\
 & & \qquad s' = [0..n] \\
 & & \qquad h = \texttt{zip}\ (\texttt{get}\ s')\ v \\
 & & \qquad h' = h +\!\!+\ g' \\
 & & \quad \textbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'
\end{array}
$$

Assuming Shape-Injectivity

$$
\begin{array}{llll}
\texttt{inv} :: [\alpha] \rightarrow & \text{IntMap } \alpha & \rightarrow [\alpha] & \\
\texttt{inv } v \quad g' & = \textbf{let } n & = \texttt{shapeInv} \ (\texttt{length } v) - 1 & \\
 & \qquad s' & = [0..n] & \\
 & \qquad h & = \texttt{zip} \ (\texttt{get } s') \ v & \\
 & \qquad h' & = h \mathbin{+\!\!+} g' & \\
 & \textbf{in } & \texttt{map} \ (\lambda i \rightarrow \texttt{fromJust} \ (\texttt{lookup } i \ h')) \ s' &
\end{array}
$$

But how to obtain `shapeInv` ???

Assuming Shape-Injectivity

$$
\begin{array}{lllll}
\texttt{inv} :: [\alpha] \to & \mathsf{IntMap}\ \alpha & \to [\alpha] & & \\
\texttt{inv}\ v & g' & = \textbf{let} & n & = \texttt{shapeInv}\ (\texttt{length}\ v) - 1 \\
 & & & s' & = [0..n] \\
 & & & h & = \texttt{zip}\ (\texttt{get}\ s')\ v \\
 & & & h' & = h +\!\!+\ g' \\
 & & \textbf{in} & \multicolumn{2}{l}{\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'}
\end{array}
$$

But how to obtain shapeInv ???

One possibility: provided by user.

Assuming Shape-Injectivity

$$
\begin{array}{l}
\texttt{inv} :: [\alpha] \to \text{IntMap}\ \alpha \to [\alpha] \\
\texttt{inv}\ v\ g' = \textbf{let}\ n = \texttt{shapeInv}\ (\texttt{length}\ v) - 1 \\
\qquad\qquad\qquad\quad s' = [0..n] \\
\qquad\qquad\qquad\quad h = \texttt{zip}\ (\texttt{get}\ s')\ v \\
\qquad\qquad\qquad\quad h' = h +\!\!+\ g' \\
\qquad\qquad\quad\ \textbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'
\end{array}
$$

But how to obtain `shapeInv` ???

One possibility: provided by user.

Another possibility:

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

Not Quite There, Yet

Works quite nicely in some cases:

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

Not Quite There, Yet

Works quite nicely in some cases:

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

Not Quite There, Yet

Works quite nicely in some cases:

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

But not so in others:

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

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 does not "collapse enough".

Not Quite There, Yet

Works quite nicely in some cases:

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

But not so in others:

$$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \begin{array}{l}\texttt{put}\ \text{“abcde”}\ \text{“xyz” fails, because} \\ \texttt{compl}\ \text{“abcde”} = [(4, \text{'e'})]\end{array}$$

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

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

Eliminating Indices

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

Eliminating Indices

$$
\begin{aligned}
&\texttt{compl} :: [\alpha] \to [\quad\;\; \alpha\;] \\
&\texttt{compl}\ s = \textbf{let}\ n\ \ = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\qquad\ \ s'\ \ = [0..n] \\
&\qquad\qquad\qquad\ \ g\ \ = \texttt{zip}\ s'\ s \\
&\qquad\qquad\qquad\ \ g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\text{get}\ s'))\ g \\
&\qquad\qquad\quad \textbf{in}\ \ \color{red}{\texttt{map}\ \texttt{snd}}\ g'
\end{aligned}
$$

Eliminating Indices

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

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

Eliminating Indices

$$
\begin{array}{l}
\texttt{compl} :: [\alpha] \rightarrow [\quad \alpha\ ] \\
\texttt{compl}\ s = \textbf{let}\ n\ \ = (\texttt{length}\ s) - 1 \\
\qquad\qquad\qquad\ s' \ = [0..n] \\
\qquad\qquad\qquad\ g\ \ = \texttt{zip}\ s'\ s \\
\qquad\qquad\qquad\ g' = \texttt{filter}\ (\lambda(i, \_) \rightarrow \texttt{notElem}\ i\ (\texttt{get}\ s'))\ g \\
\qquad\qquad \textbf{in}\ \ \texttt{map}\ \texttt{snd}\ g'
\end{array}
$$

$$
\begin{array}{l}
\texttt{inv} :: [\alpha] \rightarrow [\quad \alpha\ ] \rightarrow [\alpha] \\
\texttt{inv}\ v\ c\ = \textbf{let}\ n\ \ = \texttt{shapeInv}\ (\texttt{length}\ v) - 1 \\
\qquad\qquad\qquad\ s' \ = [0..n] \\
\qquad\qquad\qquad\ h\ \ = \texttt{zip}\ (\texttt{get}\ s')\ v \\
\qquad\qquad\qquad\ g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \rightarrow \texttt{notElem}\ i\ (\texttt{get}\ s'))\ s')\ c \\
\qquad\qquad\qquad\ h' = h +\!\!+\ g' \\
\qquad\qquad \textbf{in}\ \ \texttt{map}\ (\lambda i \rightarrow \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'
\end{array}
$$

Eliminating Indices

$$
\begin{array}{l}
\texttt{compl} :: [\alpha] \to [\quad \alpha\ ] \\
\texttt{compl}\ s = \textbf{let}\ n \ = (\texttt{length}\ s) - 1 \\
\qquad\qquad\qquad\ \ s' = [0..n] \\
\qquad\qquad\qquad\ \ g\ = \texttt{zip}\ s'\ s \\
\qquad\qquad\qquad\ \ g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ s'))\ g \\
\qquad\qquad\quad \textbf{in}\ \ \texttt{map}\ \texttt{snd}\ g'
\end{array}
$$

$$
\begin{array}{l}
\texttt{inv} :: [\alpha] \to [\quad \alpha\ ] \to [\alpha] \\
\texttt{inv}\ v\ c\ = \textbf{let}\ n\ = \texttt{shapeInv}\ (\texttt{length}\ v) - 1 \\
\qquad\qquad\qquad\ \ s' = [0..n] \\
\qquad\qquad\qquad\ \ h\ = \texttt{zip}\ (\texttt{get}\ s')\ v \\
\qquad\qquad\qquad\ \ g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \to \texttt{notElem}\ i\ (\texttt{get}\ s'))\ s')\ c \\
\qquad\qquad\qquad\ \ h' = h \mathbin{+\!\!+} g' \\
\qquad\qquad\quad \textbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'
\end{array}
$$

Now:

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

More Examples

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

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

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 $\mathtt{get} = \mathtt{sieve}$ with:

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

Then:

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

More Examples

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

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

Then:

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

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:

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

However:

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

More Examples

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

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

Then:

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

However:

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

Whereas we might have preferred:

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