# Free Theorems for Bidirectional Transformation

Janis Voigtländer

Technische Universität Dresden

GRACE-BX'08

# Free theorems: Example in Haskell

For every

$$g :: [\alpha] \to [\alpha]$$

# Free theorems: Example in Haskell

For every

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

and the standard function

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

# Free theorems: Example in Haskell

For every

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

and the standard function

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

holds, with arbitrary choice for $f$ and $l$,

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

# Free theorems: Example in Haskell

For every

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

and the standard function

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

holds, with arbitrary choice for $f$ and $l$,

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


- P. Wadler.
  Theorems for Free!
  In *Functional Programming Languages and Computer Architecture, Proceedings*. ACM Press, 1989.

# What does this have to do with Bidirectionalization?

Assume we are given some

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

# What does this have to do with Bidirectionalization?

Assume we are given some

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

and would like to produce from it a reasonable

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

# What does this have to do with Bidirectionalization?

Assume we are given some

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

and would like to produce from it a reasonable

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

with

$$\begin{aligned} \texttt{put}\ s\ (\texttt{get}\ s) &= s \\ \texttt{get}\ (\texttt{put}\ s\ v) &= v \\ &\vdots \end{aligned}$$

# What does this have to do with Bidirectionalization?

Assume we are given some

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

and would like to produce from it a reasonable

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

with

$$\begin{array}{rcl} \texttt{put}\ s\ (\texttt{get}\ s) & = & s \\ \texttt{get}\ (\texttt{put}\ s\ v) & = & v \\ & \vdots & \end{array}$$

Clearly, we need to be able to analyze get somehow.

# What have free theorems to do with Bidirectionalization?

How about doing this analysis by applying `get` to some input?

# What have free theorems to do with Bidirectionalization?

How about doing this analysis by applying `get` to some input?

Like:

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

# What have free theorems to do with Bidirectionalization?

How about doing this analysis by applying get to some input?

Like:

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

Using the free theorem

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

established earlier, the insights thus gained can be transferred to source lists other than $[0..n]$.

# What have free theorems to do with Bidirectionalization?

How about doing this analysis by applying get to some input?

Like:

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

Using the free theorem

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

established earlier, the insights thus gained can be transferred to source lists other than $[0..n]$.

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

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

# What have free theorems to do with Bidirectionalization?

How about doing this analysis by applying get to some input?

Like:

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

Using the free theorem

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

established earlier, the insights thus gained can be transferred to source lists other than $[0..n]$.

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

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

# What have free theorems to do with Bidirectionalization?

How about doing this analysis by applying get to some input?

Like:

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

Using the free theorem

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

established earlier, the insights thus gained can be transferred to source lists other than $[0..n]$.

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

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

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

# The resulting Bidirectionalization scheme (almost):

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

# The resulting Bidirectionalization scheme (almost):

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

# The resulting Bidirectionalization scheme (almost):

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

# The resulting Bidirectionalization scheme (almost):

$$
\begin{aligned}
&\texttt{put} :: [\alpha] \to [\alpha] \to [\alpha] \\
&\texttt{put}\ s\ v = \textbf{let}\ n \ \ = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\qquad s' \ = [0..n] \\
&\qquad\qquad\qquad g \ \ = \texttt{zip}\ s'\ s \\
&\qquad\qquad\qquad h \ \ = \texttt{zip}\ (\texttt{get}\ s')\ v \\
&\qquad\qquad\qquad h' \ = h + g \\
&\qquad\qquad\ \ \textbf{in}\ \ \texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ s'
\end{aligned}
$$

# The resulting Bidirectionalization scheme (almost):

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

# The resulting Bidirectionalization scheme (almost):

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

# The resulting Bidirectionalization scheme (almost):

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

# The resulting Bidirectionalization scheme (almost):

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

# The resulting Bidirectionalization scheme (almost):

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

For the full story, see:


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

# What I would like to tell you more about

Technical presentation:

- a constant-complement perspective on my method
  (rephrasing/deconstructing the POPL paper's approach)
- expanding the scope of semantic bidirectionalization
  by throwing in additional assumptions
- ideas for future work