Hilft `foldr`/`build` denn immer?

Mal ein anderes Beispiel:

$$
\begin{array}{l}
\texttt{fromTo} :: (\mathsf{Ord}\ \alpha, \mathsf{Enum}\ \alpha) \Rightarrow \alpha \to \alpha \to [\alpha] \\
\texttt{fromTo}\ n\ m = go\ n \\
\quad \textbf{where}\ go\ i = \textbf{if}\ i > m\ \textbf{then}\ [\,] \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ i : (go\ (\texttt{succ}\ i))
\end{array}
$$

$$
\begin{array}{l}
\texttt{zip} :: [\alpha] \to [\beta] \to [(\alpha, \beta)] \\
\texttt{zip}\ [\,] \qquad\quad [\,] \qquad\quad = [\,] \\
\texttt{zip}\ (a : as)\ (b : bs) = (a, b) : (\texttt{zip}\ as\ bs)
\end{array}
$$

Und dann ein Ausdruck mit zwei Zwischenergebnissen:

$$\texttt{zip}\ (\texttt{fromTo}\ 1\ 10)\ (\texttt{fromTo}\ '\mathrm{a}'\ '\mathrm{j}')$$

Was nun?

The Dual of Short Cut Deforestation [Svenningsson 2002]

Lösung:

1. Schreibe `fromTo` mittels `unfoldr`.
2. Schreibe `zip` mittels `destroy`.
3. Benutze folgende Regel:

$$\texttt{destroy}\ \mathit{cons}\ (\texttt{unfoldr}\ f\ \ b) \quad \rightsquigarrow \quad \mathit{cons}\ f\ \ b$$

Zunächst `unfoldr`:

$$\textbf{data}\ \text{Maybe}\ \alpha = \text{Nothing} \mid \text{Just}\ \alpha$$

$$\begin{array}{l}
\texttt{unfoldr} :: (\beta \to \text{Maybe}\ (\alpha, \beta)) \to \beta \to [\alpha] \\
\texttt{unfoldr}\ f\ \ b = \textbf{case}\ f\ \ b\ \textbf{of}\ \begin{array}[t]{ll} \text{Nothing} & \to [\,] \\ \text{Just}\ (a, b') & \to a : (\texttt{unfoldr}\ f\ \ b') \end{array}
\end{array}$$

Damit zum Beispiel:

$$\begin{array}{l}
\texttt{fromTo} :: (\text{Ord}\ \alpha, \text{Enum}\ \alpha) \Rightarrow \alpha \to \alpha \to [\alpha] \\
\texttt{fromTo}\ n\ \ m = \texttt{unfoldr}\ \mathit{step}\ n \\
\qquad \textbf{where}\ \mathit{step}\ i = \textbf{if}\ \ i > m\ \textbf{then}\ \text{Nothing} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \text{Just}\ (i, \texttt{succ}\ \ i)
\end{array}$$

Und die destroy-Funktion…

Definition:

$$\texttt{destroy} :: (\forall\beta.\ (\beta \to \mathsf{Maybe}\ (\alpha, \beta)) \to \beta \to \gamma) \to [\alpha] \to \gamma$$
$$\texttt{destroy}\ \mathit{cons} = \mathit{cons}\ \texttt{match}$$

wobei:

$$\begin{aligned}
&\texttt{match} :: [\alpha] \to \mathsf{Maybe}\ (\alpha, [\alpha])\\
&\texttt{match}\ [\,] \qquad\quad = \mathsf{Nothing}\\
&\texttt{match}\ (a : as) = \mathsf{Just}\ (a, as)
\end{aligned}$$

Dann zum Beispiel:

$$\begin{aligned}
&\texttt{zip} :: [\alpha] \to [\beta] \to [(\alpha, \beta)]\\
&\texttt{zip}\ as\ bs = \texttt{destroy}\ (\lambda p\ x \to \texttt{destroy}\ (\lambda q\ y \to zipD\ p\ q\ x\ y)\ bs)\ as\\
&\quad \textbf{where}\ zipD = \lambda p\ q\ x\ y \to\\
&\qquad\qquad \textbf{case}\ (p\ x, q\ y)\ \textbf{of}\\
&\qquad\qquad\quad (\mathsf{Nothing}\quad, \mathsf{Nothing}\quad) \to [\,]\\
&\qquad\qquad\quad (\mathsf{Just}\ (a, x'), \mathsf{Just}\ (b, y')) \to (a, b) : (zipD\ p\ q\ x'\ y')
\end{aligned}$$

Und warum ist das jetzt „besser“ als `foldr/build`?

Aus dem problematischen Beispiel:

$$\texttt{zip}\ (\texttt{fromTo}\ 1\ 10)\ (\texttt{fromTo}\ '\mathrm{a}'\ '\mathrm{j}')$$

wird:

$$
\begin{array}{l}
\texttt{destroy}\ (\lambda p\ x \to \texttt{destroy}\ (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ (\texttt{unfoldr}\ \mathit{step}_2\ '\mathrm{a}'))\\
\qquad\qquad (\texttt{unfoldr}\ \mathit{step}_1\ 1)\\
\quad \textbf{where}\ \mathit{zipD} = \lambda p\ q\ x\ y \to\\
\qquad\qquad\qquad \textbf{case}\ (p\ x, q\ y)\ \textbf{of}\\
\qquad\qquad\qquad\quad (\mathrm{Nothing}\qquad, \mathrm{Nothing}\qquad) \to [\,]\\
\qquad\qquad\qquad\quad (\mathrm{Just}\ (a, x'), \mathrm{Just}\ (b, y')) \to (a, b) : (\mathit{zipD}\ p\ q\ x'\ y')\\
\qquad\quad \mathit{step}_1\ i = \textbf{if}\ i > 10\ \textbf{then}\ \mathrm{Nothing}\\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathrm{Just}\ (i, \texttt{succ}\ i)\\
\qquad\quad \mathit{step}_2\ i = \textbf{if}\ i > '\mathrm{j}'\ \textbf{then}\ \mathrm{Nothing}\\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathrm{Just}\ (i, \texttt{succ}\ i)
\end{array}
$$

Und warum ist das jetzt „besser“ als foldr/build?

…und damit lässt sich mittels

$$\mathtt{destroy}\ \mathit{cons}\ (\mathtt{unfoldr}\ f\ b) \quad\rightsquigarrow\quad \mathit{cons}\ f\ b$$

wie folgt fortfahren:

$$\begin{array}{l}
\mathtt{destroy}\ (\lambda p\ x \to \mathtt{destroy}\ (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ (\mathtt{unfoldr}\ \mathit{step}_2\ '\mathrm{a}'))\\
\qquad\qquad (\mathtt{unfoldr}\ \mathit{step}_1\ 1)\\
\quad \textbf{where}\ \cdots\\
\rightsquigarrow\\
\mathtt{destroy}\ (\lambda p\ x \to (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ \mathit{step}_2\ '\mathrm{a}')\\
\qquad\qquad (\mathtt{unfoldr}\ \mathit{step}_1\ 1)\\
\quad \textbf{where}\ \cdots\\
\rightsquigarrow\\
(\lambda p\ x \to (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ \mathit{step}_2\ '\mathrm{a}')\ \mathit{step}_1\ 1\\
\quad \textbf{where}\ \cdots\\
\rightsquigarrow\\
\mathit{zipD}\ \mathit{step}_1\ \mathit{step}_2\ 1\ '\mathrm{a}'\\
\quad \textbf{where}\ \cdots
\end{array}$$

Was heißt hier eigentlich „Dual“?

Zur Erinnerung:

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

$$
\begin{aligned}
&\texttt{unfoldr} :: (\beta \to \text{Maybe}\ (\alpha, \beta)) \to \beta \to [\alpha]\\
&\texttt{unfoldr}\ f\ \ b = \textbf{case}\ f\ \ b\ \textbf{of}\ \text{Nothing} \qquad \to [\,]\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \text{Just}\ (a, b') \to a : (\texttt{unfoldr}\ f\ \ b')
\end{aligned}
$$

Etwas „Transformiererei“ an $\texttt{foldr}$:

$$
\begin{aligned}
&\texttt{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta\\
&\texttt{foldr}\ f\ \ k\ \ l = \texttt{foldr}'\ \ l\ f\ \ k\\
&\texttt{foldr}' :: [\alpha] \to (\alpha \to \beta \to \beta) \to \beta \to \beta\\
&\texttt{foldr}'\ \ l\ \ f\ \ k = \texttt{foldr}''\ \ l\ (\lambda p \to \textbf{case}\ p\ \textbf{of}\ \{\text{Nothing} \qquad \to k;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \text{Just}\ (a, b) \to f\ \ a\ \ b\})\\
&\texttt{foldr}'' :: [\alpha] \to (\text{Maybe}\ (\alpha, \beta) \to \beta) \to \beta\\
&\texttt{foldr}''\ [\,] \qquad\quad f' = f'\ \text{Nothing}\\
&\texttt{foldr}''\ (a : as)\ f' = f'\ (\text{Just}\ (a\ , \texttt{foldr}''\ \ as\ \ f'))
\end{aligned}
$$

Eine nützliche Sicht auf foldr/build

Zur Erinnerung:

$$\mathtt{build} :: (\forall \beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha]$$
$$\mathtt{build}\ \mathit{prod} = \mathit{prod}\ (:)\ [\,]$$

und:

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

bzw.:

$$\begin{aligned}
&\mathtt{foldr}' :: [\alpha] \to (\forall \beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta)\\
&\mathtt{foldr}'\ [\,] \qquad\quad f\ k = k\\
&\mathtt{foldr}'\ (a : as)\ f\ k = f\ a\ (\mathtt{foldr}'\ as\ f\ k)
\end{aligned}$$

Es gelten:

$$\begin{aligned}
\mathtt{foldr}' \circ \mathtt{build} &= \mathtt{id}\\
\mathtt{build} \circ \mathtt{foldr}' &= \mathtt{id}
\end{aligned}$$

Analog für destroy/unfoldr

Zur Erinnerung:

$$\texttt{unfoldr} :: (\beta \to \text{Maybe } (\alpha, \beta)) \to \beta \to [\alpha]$$
$$\begin{aligned}\texttt{unfoldr}\ f\ b = \textbf{case}\ f\ b\ \textbf{of}\ &\text{Nothing} &&\to [\,] \\ &\text{Just } (a, b') &&\to a : (\texttt{unfoldr}\ f\ b')\end{aligned}$$

$$\texttt{unfoldr}' :: ([\alpha] \to \gamma) \to (\forall \beta.\ (\beta \to \text{Maybe } (\alpha, \beta)) \to \beta \to \gamma)$$
$$\texttt{unfoldr}'\ g\ f\ b = g\ (\texttt{unfoldr}\ f\ b)$$

und:

$$\texttt{destroy} :: (\forall \beta.\ (\beta \to \text{Maybe } (\alpha, \beta)) \to \beta \to \gamma) \to [\alpha] \to \gamma$$
$$\begin{aligned}&\texttt{destroy}\ \mathit{cons} = \mathit{cons}\ \texttt{match} \\ &\qquad \textbf{where}\ \texttt{match}\ [\,] &&= \text{Nothing} \\ &\qquad\qquad\ \ \texttt{match}\ (a : \mathit{as}) &&= \text{Just } (a, \mathit{as})\end{aligned}$$

Es gelten:

$$\begin{aligned}\texttt{unfoldr}' \circ \texttt{destroy} &= \texttt{id} \\ \texttt{destroy} \circ \texttt{unfoldr}' &= \texttt{id}\end{aligned}$$

Eine destroy/build-Regel?

Laut Definitionen ist

$$\texttt{destroy}\ \mathit{cons}\ (\texttt{build}\ \mathit{prod})$$

das Gleiche wie

$$\mathit{cons}\ \texttt{match}\ (\mathit{prod}\ (:)\ [\,])\,,$$

wobei:

$$\begin{aligned}\texttt{match}\ [\,] &= \text{Nothing}\\ \texttt{match}\ (a : \mathit{as}) &= \text{Just}\ (a, \mathit{as})\end{aligned}$$

Warum dann nicht einfach

$$\begin{gathered}\texttt{destroy}\ \mathit{cons}\ (\texttt{build}\ \mathit{prod})\\ \rightsquigarrow\\ \mathit{cons}\ \texttt{id}\ (\mathit{prod}\ (\lambda a\ \mathit{as} \rightarrow \text{Just}\ (a, \mathit{as}))\ \text{Nothing})\quad ?\end{gathered}$$

Erhält diese Regel die Semantik?

Beweis der Korrektheit

Alles, was wir über *cons* und *prod* wissen, sind ihre Typen:

$$\mathit{cons} :: \forall\beta.\ (\beta \to \mathsf{Maybe}\ (\tau, \beta)) \to \beta \to \tau'$$

und

$$\mathit{prod} :: \forall\beta.\ (\tau \to \beta \to \beta) \to \beta \to \beta$$

Aber dies sollte doch wohl reichen, Dank freier Theoreme?

Im Folgenden, eine Beweisskizze.

Wo beginnen?

Das freie Theorem für

$$cons :: \forall \beta.\ (\beta \rightarrow \mathrm{Maybe}\ (\tau, \beta)) \rightarrow \beta \rightarrow \tau'\ ,$$

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{array}{l} \forall \tau_1, \tau_2, f :: \tau_1 \rightarrow \tau_2. \\ \quad \forall p :: \tau_1 \rightarrow \mathrm{Maybe}\ (\tau, \tau_1), q :: \tau_2 \rightarrow \mathrm{Maybe}\ (\tau, \tau_2). \\ \qquad\quad (\forall x :: \tau_1.\ (p\ x, q\ (f\ x)) \in \mathrm{lift}_{\mathrm{Maybe}}(\mathrm{lift}_{(,)}(id, f))) \\ \qquad \Rightarrow \forall y :: \tau_1.\ \boxed{cons\ p\ y} = \boxed{cons\ q\ (f\ y)} \end{array}$$

Zur Erinnerung, wir wollen beweisen:

$$\boxed{cons\ \texttt{match}\ (prod\ (:)\ [\,])}$$
$$=$$
$$\boxed{cons\ \texttt{id}\ (prod\ (\lambda a\ as \rightarrow \mathrm{Just}\ (a, as))\ \mathrm{Nothing})}$$

Wie weiter?

Alles, was wir brauchen, ist eine Funktion f so dass:

1. $\forall x :: [\tau].\ (\texttt{match}\ x, \texttt{id}\ (f\ x)) \in \mathrm{lift}_{\mathrm{Maybe}}(\mathrm{lift}_{(,)}(id, f))$
2. $f\ (prod\ (:)\ [\,])= prod\ (\lambda a\ as \rightarrow \mathrm{Just}\ (a, as))\ \mathrm{Nothing}$

Wie weiter?

Alles, was wir brauchen, ist eine Funktion f so dass:

1. $\forall x :: [\tau].\ (\texttt{match}\ x, \texttt{id}\ (f\ x)) \in \mathrm{lift}_{\mathrm{Maybe}}(\mathrm{lift}_{(,)}(id, f))$

2. $f\ (\boxed{prod\ (:)\ [\,]}) = \boxed{prod\ (\lambda a\ as \rightarrow \mathrm{Just}\ (a, as))\ \mathrm{Nothing}}$

Das freie Theorem für

$$prod :: \forall \beta.\ (\tau \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta\,,$$

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{array}{l}
\forall \tau_1, \tau_2, f :: \tau_1 \rightarrow \tau_2. \\
\quad \forall p :: \tau \rightarrow \tau_1 \rightarrow \tau_1, q :: \tau \rightarrow \tau_2 \rightarrow \tau_2. \\
\qquad\quad (\forall x :: \tau.\ \forall y :: \tau_1.\ f\ (p\ x\ y) = q\ x\ (f\ y)) \\
\qquad \Rightarrow \forall z :: \tau_1.\ f\ (\boxed{prod\ p\ z}) = \boxed{prod\ q\ (f\ z)}
\end{array}$$

Fast geschafft

Alles, was wir brauchen, ist eine Funktion f so dass:

1. $\forall x :: [\tau].\ (\texttt{match}\ x, \texttt{id}\ (f\ x)) \in \mathsf{lift}_{\mathsf{Maybe}}(\mathsf{lift}_{(,)}(id, f))$
2. $\forall x :: \tau, y :: [\tau].\ f\ ((:)\ x\ y) = (\lambda a\ as \to \mathsf{Just}\ (a, as))\ x\ (f\ y)$
3. $f\ []\ = \mathsf{Nothing}$

Fast geschafft

Alles, was wir brauchen, ist eine Funktion f so dass:

1. $\forall x :: [\tau].\ (\texttt{match}\ x, \texttt{id}\ (f\ \ x)) \in \mathrm{lift}_{\mathrm{Maybe}}(\mathrm{lift}_{(,)}(id, f))$
2. $\forall x :: \tau, y :: [\tau].\ f\ ((:)\ \ x\ \ y) = (\lambda a\ as \rightarrow \mathrm{Just}\ (a, as))\ \ x\ (f\ \ y)$
3. $f\ \ [\,] = \mathrm{Nothing}$

Die letzten beiden Bedingungen lassen keine andere Wahl als:

$$\begin{aligned} f\ [\,] \quad &= \mathrm{Nothing} \\ f\ (x : y) &= \mathrm{Just}\ (x, f\ \ y) \end{aligned}$$

Fast geschafft

1. $\forall x :: [\tau].\ (\texttt{match}\ x, \texttt{id}\ (f\ x)) \in \mathrm{lift}_{\mathrm{Maybe}}(\mathrm{lift}_{(,)}(id, f))$?

$$\begin{aligned} f\ [\,] &= \mathrm{Nothing} \\ f\ (x : y) &= \mathrm{Just}\ (x, f\ y) \end{aligned}$$

Schließlich ...

Wir haben:

$$\text{lift}_{\text{Maybe}}(\text{lift}_{(,)}(id, f)) = \{(\text{ Nothing }, \text{ Nothing })\} \cup \{(\text{ Just } x_1 \text{ , Just } y_1 \text{ }) \mid (x_1, y_1) \in \text{lift}_{(,)}(id, f)\}$$

$$\text{lift}_{(,)}(id, f) = \{(\,(x_1, x_2)\,,\,(y_1, y_2)\,) \mid x_1 = y_1 \wedge f\ x_2 = y_2\}$$

Um zu zeigen, dass

$$\forall x :: [\tau].\ (\texttt{match}\ x, \texttt{id}\ (f\ x)) \in \text{lift}_{\text{Maybe}}(\text{lift}_{(,)}(id, f))\,,$$

untersuchen wir alle Fälle für Eingaben von `match` und f, unter Verwendung der Definitionen:

$$\begin{aligned} \texttt{match}\ [] &= \text{Nothing} & \qquad f\ [] &= \text{Nothing} \\ \texttt{match}\ (a : as) &= \text{Just}\ \ (a, as) & \qquad f\ (x : y) &= \text{Just}\ \ (x, f\ y) \end{aligned}$$

Fertig!

Short Cut Deforestation auf anderen Datentypen?

Zum Beispiel:

$$\textbf{data}\ \text{Tree}\ a\ =\ \text{Leaf}\ a \mid \text{Node}\ (\text{Tree}\ a)\ (\text{Tree}\ a)$$

$$\begin{aligned}
&\texttt{incr} :: \text{Tree Int} \to \text{Tree Int}\\
&\texttt{incr}\ (\text{Leaf}\ a) \qquad\ \ = \text{Leaf}\ (a+1)\\
&\texttt{incr}\ (\text{Node}\ t_1\ t_2) = \text{Node}\ (\texttt{incr}\ t_1)\ (\texttt{incr}\ t_2)
\end{aligned}$$

Ziel:

$$\texttt{incr}\ (\texttt{incr}\ t) \ \rightsquigarrow\ ?$$

Zunächst ein geeignetes `fold`:

$$\begin{aligned}
&\texttt{foldTree} :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \text{Tree}\ \alpha \to \beta\\
&\texttt{foldTree}\ l\ n\ (\text{Leaf}\ a) \qquad\ \ = l\ a\\
&\texttt{foldTree}\ l\ n\ (\text{Node}\ t_1\ t_2) = n\ (\texttt{foldTree}\ l\ n\ t_1)\ (\texttt{foldTree}\ l\ n\ t_2)
\end{aligned}$$

Short Cut Deforestation auf anderen Datentypen?

Außerdem ein geeignetes `build`:

$$\texttt{buildTree} :: (\forall\beta.\ (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \beta) \to \text{Tree}\ \alpha$$
$$\texttt{buildTree}\ \mathit{prod} = \mathit{prod}\ \text{Leaf Node}$$

Sowie eine `foldTree`/`buildTree`-Regel:

$$\texttt{foldTree}\ l\ n\ (\texttt{buildTree}\ \mathit{prod}) \quad \leadsto \quad \mathit{prod}\ l\ n$$

Nun:

$$\begin{aligned} &\texttt{incr} :: \text{Tree Int} \to \text{Tree Int} \\ &\texttt{incr}\ t = \texttt{buildTree}\ (\lambda l\ n \to \texttt{foldTree}\ (\lambda a \to l\ (a+1)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda t_1\ t_2 \to n\ t_1\ t_2) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad t) \end{aligned}$$

Und schließlich:

$$\begin{aligned} \texttt{incr}\ (\texttt{incr}\ t) \ &= \ \cdots\ \texttt{foldTree}\ \cdots\ (\texttt{buildTree}\ (\lambda l\ n \to \cdots\ t))\ \cdots \\ &\leadsto\ \cdots \end{aligned}$$

Circular Short Cut Deforestation [Fernandes et al. 2007]

Manchmal möchte man zusätzliche Ausgabe/Information erzeugen:

$$\texttt{filterAndCount} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to ([\alpha], \mathsf{Int})$$

und diese auch weiterverarbeiten:

$$\texttt{normalise} :: ([\mathsf{Int}], \mathsf{Int}) \to [\mathsf{Float}]$$

Zur Erzeugung:

$$\texttt{buildp} :: (\forall \beta.\ (\alpha \to \beta \to \beta) \to \beta \to \gamma \to (\beta, \delta)) \to \gamma \to ([\alpha], \delta)$$
$$\texttt{buildp}\ g\ c = g\ (:)\ [\,]\ c$$

Zur Verarbeitung:

$$\texttt{pfold} :: (\alpha \to \beta \to \delta \to \beta) \to (\delta \to \beta) \to ([\alpha], \delta) \to \beta$$
$$\texttt{pfold}\ h_1\ h_2\ (as, z) = \texttt{foldr}\ (\lambda a\ b \to h_1\ a\ b\ z)\ (h_2\ z)\ as$$

Circular Short Cut Deforestation [Fernandes et al. 2007]

Zur Erzeugung:

$$\texttt{buildp} :: (\forall \beta.\ (\alpha \to \beta \to \beta) \to \beta \to \gamma \to (\beta, \delta)) \to \gamma \to ([\alpha], \delta)$$
$$\texttt{buildp}\ g\ c = g\ (:)\ [\,]\ c$$

Zur Verarbeitung:

$$\texttt{pfold} :: (\alpha \to \beta \to \delta \to \beta) \to (\delta \to \beta) \to ([\alpha], \delta) \to \beta$$
$$\texttt{pfold}\ h_1\ h_2\ (as, z) = \texttt{foldr}\ (\lambda a\ b \to h_1\ a\ b\ z)\ (h_2\ z)\ as$$

Fusions-Regel:

$$\texttt{pfold}\ h_1\ h_2\ (\texttt{buildp}\ g\ c)$$
$$\rightsquigarrow$$
$$\textbf{let}\ (b, z) = g\ (\lambda a\ b \to h_1\ a\ b\ z)\ (h_2\ z)\ c\ \textbf{in}\ b$$

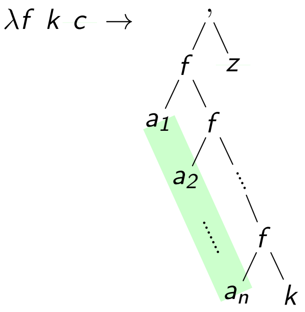
Circular Short Cut Deforestation [Fernandes et al. 2007]

Zur Erzeugung:

$$\texttt{buildp} :: \boxed{(\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \gamma \to (\beta,\delta))} \to \gamma \to ([\alpha],\delta)$$

$$\texttt{buildp}\ \boxed{g}\ c = g\ (:)\ []\ c$$

Typ von g erzwingt (semantisch gesehen) folgende Form:

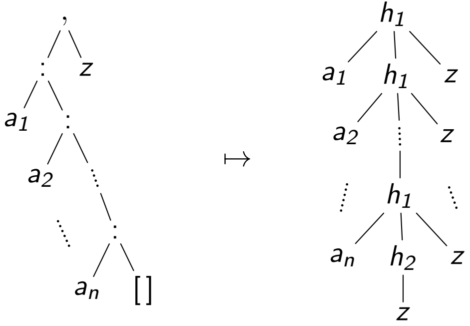


Circular Short Cut Deforestation [Fernandes et al. 2007]

Zur Verarbeitung:

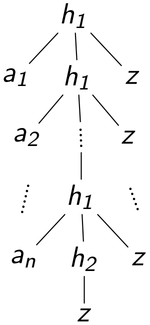
$$\texttt{pfold} :: (\alpha \to \beta \to \delta \to \beta) \to (\delta \to \beta) \to ([\alpha], \delta) \to \beta$$
$$\texttt{pfold}\ h_1\ h_2\ (as, z) = \texttt{foldr}\ (\lambda a\ b \to h_1\ a\ b\ z)\ (h_2\ z)\ as$$

Für ein konkretes Zwischenergebnis ($\texttt{buildp}\ g\ c$) bedeutet dies:

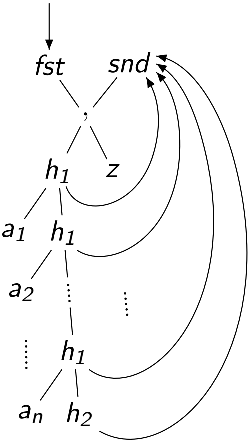


Circular Short Cut Deforestation [Fernandes et al. 2007]

$\texttt{pfold}\ h_1\ h_2\ (g\ (:)\ [\,]\ c) \rightsquigarrow \textbf{let}\ (b, z) = g\ (\lambda a\ b \to h_1\ a\ b\ z)\ (h_2\ z)\ c\ \textbf{in}\ b$

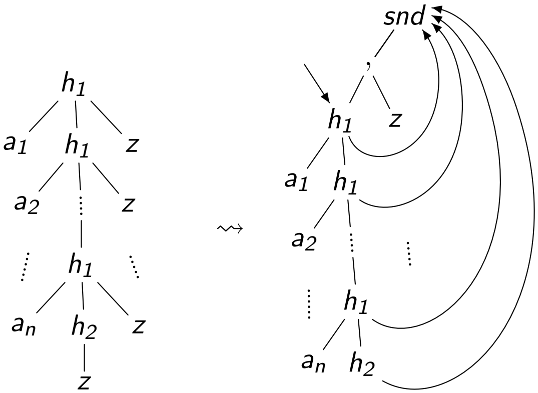


$\rightsquigarrow$



Circular Short Cut Deforestation [Fernandes et al. 2007]

$\texttt{pfold}\ h_1\ h_2\ (g\ (:)\ [\,]\ c) \rightsquigarrow \textbf{let}\ (b, z) = g\ (\lambda a\ b \rightarrow h_1\ a\ b\ z)\ (h_2\ z)\ c\ \textbf{in}\ b$



Circular Short Cut Deforestation [Fernandes et al. 2007]

`pfold` $h_1\ h_2\ (g\ (:)\ [\,]\ c) \rightsquigarrow \textbf{let}\ (b, z) = g\ (\lambda a\ b \rightarrow h_1\ a\ b\ z)\ (h_2\ z)\ c\ \textbf{in}\ b$