Hilft `foldr/build` denn immer?

Mal ein anderes Beispiel:

```
fromTo :: (Ord α, Enum α) ⇒ α → α → [α]
fromTo n m = go n
      where go i = if i > m then []
                            else  i : (go (succ i))

zip :: [α] → [β] → [(α, β)]
zip []     []     = []
zip (a:as) (b:bs) = (a, b) : (zip as bs)
```

Und dann ein Ausdruck mit zwei Zwischenergebnissen:

```
zip (fromTo 1 10) (fromTo 'a' '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:

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

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`:

data Maybe α = Nothing | Just α

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`:

data Maybe α = Nothing | Just α

`unfoldr` :: $(\beta \to \text{Maybe}\ (\alpha, \beta)) \to \beta \to [\alpha]$

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`:

data Maybe α = Nothing | Just α

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

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`:

data Maybe α = Nothing | Just α

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

Damit zum Beispiel:

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

Und die `destroy`-Funktion...

Definition:

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

Und die `destroy`-Funktion...

Definition:

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

wobei:

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

Und die `destroy`-Funktion...

Definition:

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

wobei:

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

Dann zum Beispiel:

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

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

Aus dem problematischen Beispiel:

```
zip (fromTo 1 10) (fromTo 'a' 'j')
```

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

Aus dem problematischen Beispiel:

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

wird:

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

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

. . . und damit lässt sich mittels

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

wie folgt fortfahren:

$$\begin{aligned}&\texttt{destroy}\ (\lambda p\ x \to \texttt{destroy}\ (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ (\texttt{unfoldr}\ \mathit{step}_2\ \text{'a'}))\\ &\qquad (\texttt{unfoldr}\ \mathit{step}_1\ 1)\\ &\quad \textbf{where}\ \cdots\\ \rightsquigarrow\end{aligned}$$

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

. . . und damit lässt sich mittels

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

wie folgt fortfahren:

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

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

. . . und damit lässt sich mittels

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

wie folgt fortfahren:

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

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

. . . und damit lässt sich mittels

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

wie folgt fortfahren:

$$\begin{aligned}
&\texttt{destroy}\ (\lambda p\ x \to \texttt{destroy}\ (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ (\texttt{unfoldr}\ \mathit{step}_2\ \text{'a'}))\\
&\qquad\quad (\texttt{unfoldr}\ \mathit{step}_1\ 1)\\
&\quad \textbf{where}\ \cdots\\
&\rightsquigarrow\\
&\texttt{destroy}\ (\lambda p\ x \to (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ \mathit{step}_2\ \text{'a'})\\
&\qquad\quad (\texttt{unfoldr}\ \mathit{step}_1\ 1)\\
&\quad \textbf{where}\ \cdots\\
&\rightsquigarrow\\
&(\lambda p\ x \to (\lambda q\ y \to \mathit{zipD}\ p\ q\ x\ y)\ \mathit{step}_2\ \text{'a'})\ \mathit{step}_1\ 1\\
&\quad \textbf{where}\ \cdots\\
&\rightsquigarrow\\
&\mathit{zipD}\ \mathit{step}_1\ \mathit{step}_2\ 1\ \text{'a'}\\
&\quad \textbf{where}\ \cdots
\end{aligned}$$

Was heißt hier eigentlich „Dual“?

Zur Erinnerung:

```
foldr :: (α → β → β) → β → [α] → β
foldr f k []       = k
foldr f k (a : as) = f a (foldr f k as)

unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing      → []
                          Just (a, b') → a : (unfoldr f b')
```

Was heißt hier eigentlich „Dual“?

Zur Erinnerung:

```
foldr :: (α → β → β) → β → [α] → β
foldr f k []       = k
foldr f k (a : as) = f a (foldr f k as)

unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing     → []
                          Just (a, b') → a : (unfoldr f b')
```

Etwas „Transformiererei“ an `foldr`:

```
foldr :: (α → β → β) → β → [α] → β
foldr f k l = foldr' l f k
foldr' :: [α] → (α → β → β) → β → β
foldr' []       f k = k
foldr' (a : as) f k = f a (foldr' as f k)
```

Was heißt hier eigentlich „Dual“?

Zur Erinnerung:

```
foldr :: (α → β → β) → β → [α] → β
foldr f k []       = k
foldr f k (a : as) = f a (foldr f k as)

unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing     → []
                          Just (a, b') → a : (unfoldr f b')
```

Etwas „Transformiererei“ an `foldr`:

```
foldr :: (α → β → β) → β → [α] → β
foldr f k l = foldr' l f k
foldr' :: [α] → (α → β → β) → β → β
foldr' []       f k = k
foldr' (a : as) f k = f a (foldr' as f k)
foldr'' :: [α] → (Maybe (α, β) → β) → β
foldr'' []       f' = f' Nothing
foldr'' (a : as) f' = f' (Just (a , foldr'' as f'))
```

Was heißt hier eigentlich „Dual“?

Zur Erinnerung:

```
foldr :: (α → β → β) → β → [α] → β
foldr f k []       = k
foldr f k (a : as) = f a (foldr f k as)

unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing     → []
                          Just (a, b') → a : (unfoldr f b')
```

Etwas „Transformiererei“ an `foldr`:

```
foldr :: (α → β → β) → β → [α] → β
foldr f k l = foldr' l f k
foldr' :: [α] → (α → β → β) → β → β
foldr' l f k = foldr'' l (λp → case p of {Nothing     → k;
                                          Just (a, b) → f a b})
foldr'' :: [α] → (Maybe (α, β) → β) → β
foldr'' []       f' = f' Nothing
foldr'' (a : as) f' = f' (Just (a , foldr'' as f'))
```

Eine nützliche Sicht auf `foldr/build`

Zur Erinnerung:

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

Eine nützliche Sicht auf `foldr/build`

Zur Erinnerung:

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

und:

$$\texttt{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$$
$$\texttt{foldr}\ f\ k\ [] = k$$
$$\texttt{foldr}\ f\ k\ (a : as) = f\ a\ (\texttt{foldr}\ f\ k\ as)$$

Eine nützliche Sicht auf `foldr/build`

Zur Erinnerung:

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

und:

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

bzw.:

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

Eine nützliche Sicht auf `foldr/build`

Zur Erinnerung:

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

und:

$$\texttt{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$$
$$\texttt{foldr}\ f\ k\ [] = k$$
$$\texttt{foldr}\ f\ k\ (a : as) = f\ a\ (\texttt{foldr}\ f\ k\ as)$$

bzw.:

$$\texttt{foldr}' :: [\alpha] \to (\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta)$$
$$\texttt{foldr}'\ []\ f\ k = k$$
$$\texttt{foldr}'\ (a : as)\ f\ k = f\ a\ (\texttt{foldr}'\ as\ f\ k)$$

Eine nützliche Sicht auf `foldr/build`

Zur Erinnerung:

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

und:

$$\texttt{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$$
$$\texttt{foldr}\ f\ k\ [] = k$$
$$\texttt{foldr}\ f\ k\ (a : as) = f\ a\ (\texttt{foldr}\ f\ k\ as)$$

bzw.:

$$\texttt{foldr}' :: [\alpha] \to (\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta)$$
$$\texttt{foldr}'\ []\ f\ k = k$$
$$\texttt{foldr}'\ (a : as)\ f\ k = f\ a\ (\texttt{foldr}'\ as\ f\ k)$$

Eine nützliche Sicht auf `foldr/build`

Zur Erinnerung:

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

und:

$$\texttt{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$$
$$\texttt{foldr}\ f\ k\ [] = k$$
$$\texttt{foldr}\ f\ k\ (a : as) = f\ a\ (\texttt{foldr}\ f\ k\ as)$$

bzw.:

$$\texttt{foldr}' :: [\alpha] \to (\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta)$$
$$\texttt{foldr}'\ []\ f\ k = k$$
$$\texttt{foldr}'\ (a : as)\ f\ k = f\ a\ (\texttt{foldr}'\ as\ f\ k)$$

Es gelten:

$$\texttt{foldr}' \circ \texttt{build} = \texttt{id}$$
$$\texttt{build} \circ \texttt{foldr}' = \texttt{id}$$

Analog für `destroy`/`unfoldr`

Zur Erinnerung:

```
unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing      → []
                          Just (a, b') → a : (unfoldr f b')
```

Analog für `destroy/unfoldr`

Zur Erinnerung:

```
unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing     → []
                          Just (a, b') → a : (unfoldr f b')
```

und:

```
destroy :: (∀β. (β → Maybe (α, β)) → β → γ) → [α] → γ
destroy cons = cons match
    where match []       = Nothing
          match (a : as) = Just (a, as)
```

Analog für `destroy/unfoldr`

Zur Erinnerung:

```
unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing      → []
                          Just (a, b') → a : (unfoldr f b')

unfoldr' :: ([α] → γ) → (β → Maybe (α, β)) → β → γ
unfoldr' g f b = g (unfoldr f b)
```

und:

```
destroy :: (∀β. (β → Maybe (α, β)) → β → γ) → [α] → γ
destroy cons = cons match
    where match []       = Nothing
          match (a : as) = Just (a, as)
```

Analog für `destroy/unfoldr`

Zur Erinnerung:

```
unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing      → []
                          Just (a, b') → a : (unfoldr f b')

unfoldr' :: ([α] → γ) → (∀β. (β → Maybe (α, β)) → β → γ)
unfoldr' g f b = g (unfoldr f b)
```

und:

```
destroy :: (∀β. (β → Maybe (α, β)) → β → γ) → [α] → γ
destroy cons = cons match
    where match []       = Nothing
          match (a : as) = Just (a, as)
```

Analog für `destroy/unfoldr`

Zur Erinnerung:

```
unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing      → []
                          Just (a, b') → a : (unfoldr f b')

unfoldr' :: ([α] → γ) → (∀β. (β → Maybe (α, β)) → β → γ)
unfoldr' g f b = g (unfoldr f b)
```

und:

```
destroy :: (∀β. (β → Maybe (α, β)) → β → γ) → [α] → γ
destroy cons = cons match
    where match []       = Nothing
          match (a : as) = Just (a, as)
```

Analog für `destroy`/`unfoldr`

Zur Erinnerung:

```
unfoldr :: (β → Maybe (α, β)) → β → [α]
unfoldr f b = case f b of Nothing    → []
                          Just (a, b') → a : (unfoldr f b')

unfoldr' :: ([α] → γ) → (∀β. (β → Maybe (α, β)) → β → γ)
unfoldr' g f b = g (unfoldr f b)
```

und:

```
destroy :: (∀β. (β → Maybe (α, β)) → β → γ) → [α] → γ
destroy cons = cons match
    where match []       = Nothing
          match (a : as) = Just (a, as)
```

Es gelten:

$$\texttt{unfoldr}' \circ \texttt{destroy} = \texttt{id}$$

$$\texttt{destroy} \circ \texttt{unfoldr}' = \texttt{id}$$

Eine `destroy/build`-Regel?

Laut Definitionen ist

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

...

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:

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

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:

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

Warum dann nicht einfach

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

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:

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

Warum dann nicht einfach

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

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

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

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

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

ist:

$$\begin{array}{l}
\forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. \\
\quad \forall p :: \tau_1 \to \mathsf{Maybe}\ (\tau, \tau_1), q :: \tau_2 \to \mathsf{Maybe}\ (\tau, \tau_2). \\
\qquad (\forall (x, y) \in \mathcal{R}.\ (p\ x, q\ y) \in \mathsf{lift}_{\mathsf{Maybe}}(\mathsf{lift}_{(,)}(id, \mathcal{R}))) \\
\qquad \Rightarrow \forall (z, v) \in \mathcal{R}.\ cons\ p\ z = cons\ q\ v
\end{array}$$

Wo beginnen?

Das freie Theorem für

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

ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. \\
&\quad \forall p :: \tau_1 \to \text{Maybe}\ (\tau, \tau_1), q :: \tau_2 \to \text{Maybe}\ (\tau, \tau_2). \\
&\qquad (\forall (x, y) \in \mathcal{R}.\ (p\ x, q\ y) \in \text{lift}_{\text{Maybe}}(\text{lift}_{(,)}(id, \mathcal{R}))) \\
&\qquad \Rightarrow \forall (z, v) \in \mathcal{R}.\ cons\ p\ z = cons\ q\ v
\end{aligned}$$

Zur Erinnerung, wir wollen beweisen:

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

$$=$$

$$cons\ \texttt{id}\ (prod\ (\lambda a\ as \to \text{Just}\ (a, as))\ \text{Nothing})$$

Wo beginnen?

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

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

Zur Erinnerung, wir wollen beweisen:

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

Wo beginnen?

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2.\\
&\quad \forall p :: \tau_1 \to \text{Maybe}\ (\tau, \tau_1), q :: \tau_2 \to \text{Maybe}\ (\tau, \tau_2).\\
&\qquad (\forall x :: \tau_1.\ (p\ x, q\ (f\ x)) \in \text{lift}_{\text{Maybe}}(\text{lift}_{(,)}(id, f)))\\
&\qquad \Rightarrow \forall y :: \tau_1.\ cons\ p\ y = cons\ q\ (f\ y)
\end{aligned}$$

Zur Erinnerung, wir wollen beweisen:

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

Wo beginnen?

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2. \\
&\quad \forall p :: \tau_1 \to \text{Maybe}\ (\tau, \tau_1), q :: \tau_2 \to \text{Maybe}\ (\tau, \tau_2). \\
&\qquad (\forall x :: \tau_1.\ (p\ x, q\ (f\ x)) \in \text{lift}_{\text{Maybe}}(\text{lift}_{(,)}(id, f))) \\
&\qquad \Rightarrow \forall y :: \tau_1.\ cons\ p\ y = cons\ q\ (f\ y)
\end{aligned}$$

Zur Erinnerung, wir wollen beweisen:

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

$$=$$

$$cons\ \texttt{id}\ (prod\ (\lambda a\ as \to \text{Just}\ (a, as))\ \text{Nothing})$$

Wo beginnen?

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2.\\
&\quad \forall p :: \tau_1 \to \text{Maybe}\ (\tau, \tau_1), q :: \tau_2 \to \text{Maybe}\ (\tau, \tau_2).\\
&\qquad (\forall x :: \tau_1.\ (p\ x, q\ (f\ x)) \in \text{lift}_{\text{Maybe}}(\text{lift}_{(,)}(id, f)))\\
&\qquad \Rightarrow \forall y :: \tau_1.\ cons\ p\ y = cons\ q\ (f\ y)
\end{aligned}$$

Zur Erinnerung, wir wollen beweisen:

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

$$=$$

$$cons\ \texttt{id}\ (prod\ (\lambda a\ as \to \text{Just}\ (a, as))\ \text{Nothing})$$

Wo beginnen?

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2. \\
&\quad \forall p :: \tau_1 \to \text{Maybe}\ (\tau, \tau_1), q :: \tau_2 \to \text{Maybe}\ (\tau, \tau_2). \\
&\qquad (\forall x :: \tau_1.\ (p\ x, q\ (f\ x)) \in \text{lift}_{\text{Maybe}}(\text{lift}_{(,)}(id, f))) \\
&\qquad \Rightarrow \forall y :: \tau_1.\ cons\ p\ y = cons\ q\ (f\ y)
\end{aligned}$$

Zur Erinnerung, wir wollen beweisen:

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

$$=$$

$$cons\ \texttt{id}\ (prod\ (\lambda a\ as \to \text{Just}\ (a, as))\ \text{Nothing})$$

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

Wie weiter?

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

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

Wie weiter?

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

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

Das freie Theorem für

$$prod :: \forall\beta.\ (\tau \to \beta \to \beta) \to \beta \to \beta$$

ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. \\
&\quad \forall p :: \tau \to \tau_1 \to \tau_1, q :: \tau \to \tau_2 \to \tau_2. \\
&\qquad (\forall x :: \tau.\ \forall (y, z) \in \mathcal{R}.\ (p\ x\ y, q\ x\ z) \in \mathcal{R}) \\
&\qquad \Rightarrow \forall (v, w) \in \mathcal{R}.\ (prod\ p\ v, prod\ q\ w) \in \mathcal{R}
\end{aligned}$$

Wie weiter?

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. $f\ \ (prod\ (:)\ [\,]) = prod\ (\lambda a\ as \to \mathsf{Just}\ (a, as))\ \mathsf{Nothing}$

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2.\\
&\quad \forall p :: \tau \to \tau_1 \to \tau_1, q :: \tau \to \tau_2 \to \tau_2.\\
&\qquad (\forall x :: \tau.\ \forall y :: \tau_1.\ f\ (p\ x\ y) = q\ x\ (f\ y))\\
&\qquad \Rightarrow \forall z :: \tau_1.\ f\ (\,prod\ p\ z\,) = prod\ q\ (f\ z)
\end{aligned}$$

Wie weiter?

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. $f\ (prod\ (:)\ [\,]) = prod\ (\lambda a\ as \to \mathsf{Just}\ (a, as))\ \mathsf{Nothing}$

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2.\\
&\quad \forall p :: \tau \to \tau_1 \to \tau_1, q :: \tau \to \tau_2 \to \tau_2.\\
&\qquad (\forall x :: \tau.\ \forall y :: \tau_1.\ f\ (p\ x\ y) = q\ x\ (f\ y))\\
&\qquad \Rightarrow \forall z :: \tau_1.\ f\ (prod\ p\ z) = prod\ q\ (f\ z)
\end{aligned}$$

Wie weiter?

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. $f\ (prod\ (:)\ [\,]) = prod\ (\lambda a\ as \to \mathsf{Just}\ (a, as))\ \mathsf{Nothing}$

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2.\\
&\quad \forall p :: \tau \to \tau_1 \to \tau_1, q :: \tau \to \tau_2 \to \tau_2.\\
&\qquad (\forall x :: \tau.\ \forall y :: \tau_1.\ f\ (p\ x\ y) = q\ x\ (f\ y))\\
&\qquad \Rightarrow \forall z :: \tau_1.\ f\ (prod\ p\ z) = prod\ q\ (f\ z)
\end{aligned}$$

Wie weiter?

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. $f\ (prod\ (:)\ [\,]) = prod\ (\lambda a\ as \to \mathsf{Just}\ (a, as))\ \mathsf{Nothing}$

Das freie Theorem für

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

spezialisiert auf die Ebene von Funktionen, ist:

$$\begin{aligned}
&\forall \tau_1, \tau_2, f :: \tau_1 \to \tau_2.\\
&\quad \forall p :: \tau \to \tau_1 \to \tau_1, q :: \tau \to \tau_2 \to \tau_2.\\
&\qquad (\forall x :: \tau.\ \forall y :: \tau_1.\ f\ (p\ x\ y) = q\ x\ (f\ y))\\
&\qquad \Rightarrow \forall z :: \tau_1.\ f\ (prod\ p\ z) = prod\ q\ (f\ z)
\end{aligned}$$

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. $\forall x :: \tau, y :: [\tau].\ f\ ((:)\ x\ y) = (\lambda a\ as \to \mathrm{Just}\ (a, as))\ x\ (f\ y)$
3. $f\ [] = \mathrm{Nothing}$

Fast geschafft

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

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

Die letzten beiden Bedingungen lassen keine andere Wahl als:

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

Fast geschafft

Alles, was wir brauchen, ist:

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

Die letzten beiden Bedingungen lassen keine andere Wahl als:

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

Fast geschafft

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

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

Schließlich . . .

Wir haben:

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

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

Schließlich . . .

Wir haben:

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

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

Um zu zeigen, dass

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

Schließlich . . .

Wir haben:

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

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

Um zu zeigen, dass

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

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

$$\begin{array}{lll} \texttt{match}\ [\,] & = \mathsf{Nothing} & f\ [\,] = \mathsf{Nothing} \\ \texttt{match}\ (a : as) & = \mathsf{Just}\ \ (a, as) & f\ (x : y) = \mathsf{Just}\ \ (x, f\ y) \end{array}$$

Schließlich . . .

Wir haben:

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

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

Um zu zeigen, dass

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

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

$$\begin{array}{lll} \texttt{match}\ [\,] & = \mathsf{Nothing} & f\ [\,] = \mathsf{Nothing} \\ \texttt{match}\ (a : as) & = \mathsf{Just}\ \ (a, as) & f\ (x : y) = \mathsf{Just}\ \ (x, f\ y) \end{array}$$

Schließlich . . .

Wir haben:

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

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

Um zu zeigen, dass

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

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

$$\begin{array}{lll} \texttt{match}\ [\,] & = \mathsf{Nothing} & f\ [\,] = \mathsf{Nothing} \\ \texttt{match}\ (a : as) & = \mathsf{Just}\ \ (a, as) & f\ (x : y) = \mathsf{Just}\ \ (x, f\ y) \end{array}$$

Schließlich . . .

Wir haben:

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

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

Um zu zeigen, dass

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

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

$$\begin{array}{lll} \texttt{match}\ [\,] & = \mathsf{Nothing} & f\ [\,] = \mathsf{Nothing} \\ \texttt{match}\ (a : as) & = \mathsf{Just}\ (a, as) & f\ (x : y) = \mathsf{Just}\ (x, f\ y) \end{array}$$

Schließlich . . .

Wir haben:

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

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

Um zu zeigen, dass

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

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

$$\begin{array}{lll} \texttt{match}\ [\,] & = \mathsf{Nothing} & f\ [\,] = \mathsf{Nothing} \\ \texttt{match}\ (a : as) & = \mathsf{Just}\ \ (a, as) & f\ (x : y) = \mathsf{Just}\ \ (x, f\ y) \end{array}$$

Fertig!

Short Cut Deforestation auf anderen Datentypen?

Zum Beispiel:

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

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

Short Cut Deforestation auf anderen Datentypen?

Zum Beispiel:

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

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

Ziel:

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

Short Cut Deforestation auf anderen Datentypen?

Zum Beispiel:

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

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

Ziel:

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

Zunächst ein geeignetes `fold`:

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

Short Cut Deforestation auf anderen Datentypen?

Außerdem ein geeignetes `build`:

```
buildTree :: (∀β. (α → β) → (β → β → β) → β) → Tree α
buildTree prod = prod Leaf Node
```

Short Cut Deforestation auf anderen Datentypen?

Außerdem ein geeignetes `build`:

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

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

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

Short Cut Deforestation auf anderen Datentypen?

Außerdem ein geeignetes `build`:

```
buildTree :: (∀β. (α → β) → (β → β → β) → β) → Tree α
buildTree prod = prod Leaf Node
```

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

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

Nun:

```
incr :: Tree Int → Tree Int
incr t = buildTree (λl n → foldTree (λa → l (a + 1))
                                    (λt1 t2 → n t1 t2)
                                    t)
```

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:

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

Nun:

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

Und schließlich:

$$\texttt{incr}\ (\texttt{incr}\ t) \quad = \quad \cdots\ \texttt{foldTree}\ \cdots\ (\texttt{buildTree}\ (\lambda l\ n \to \cdots\ t))\ \cdots$$

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:

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

Nun:

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

Und schließlich:

$$\texttt{incr}\ (\texttt{incr}\ t) = \cdots\ \texttt{foldTree}\ \cdots\ (\texttt{buildTree}\ (\lambda l\ n \to \cdots\ t))\ \cdots$$
$$\rightsquigarrow\ \cdots$$

Circular Short Cut Deforestation [Fernandes et al. 2007]

Manchmal möchte man zusätzliche Ausgabe/Information erzeugen:

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

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:

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

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:

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

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:

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

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:

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

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

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:

$$\texttt{pfold} :: (\alpha \to \beta \to \delta \to \beta) \to (\delta \to \beta) \to ([\alpha], \delta) \to \beta$$

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]

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:

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

$$\texttt{normalise} :: ([\mathit{Int}], \mathit{Int}) \to [\mathit{Float}]$$
$$\texttt{normalise} = \texttt{pfold}\ \cdots$$

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:

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

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:

$$\texttt{pfold}\ h_1\ h_2\ (\texttt{buildp}\ g\ c)$$
$$\rightsquigarrow$$

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:

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

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:

```
buildp :: (∀β. (α → β → β) → β → γ → (β, δ)) → γ → ([α], δ)
buildp g c = g (:) [] c
```

Circular Short Cut Deforestation [Fernandes et al. 2007]

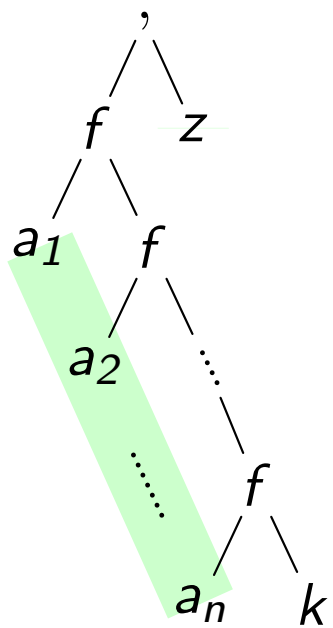
Zur Erzeugung:

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

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

Typ von g erzwingt (semantisch gesehen) folgende Form:

$\lambda f\ k\ c\ \to$



Circular Short Cut Deforestation [Fernandes et al. 2007]

Zur Verarbeitung:

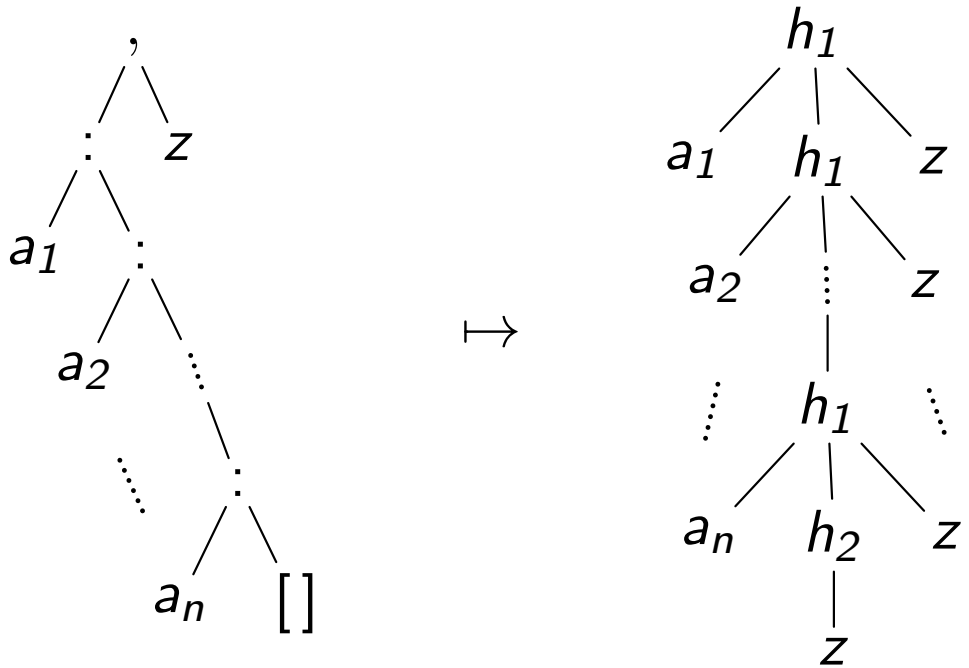
$$\texttt{pfold} :: (\alpha \to \beta \to \delta \to \beta) \to (\delta \to \beta) \to ([\alpha], \delta) \to \beta$$
$$\texttt{pfold}\ h_1\ h_2\ (as, z) = \texttt{foldr}\ (\lambda a\ b \to h_1\ a\ b\ z)\ (h_2\ z)\ as$$

Circular Short Cut Deforestation [Fernandes et al. 2007]

Zur Verarbeitung:

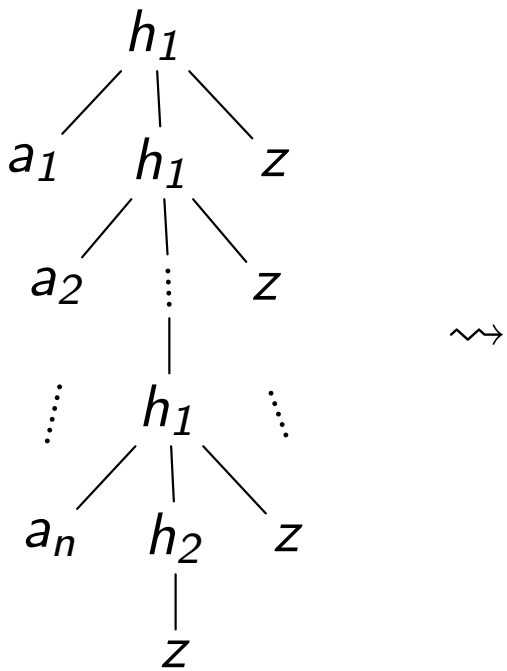
$$\texttt{pfold} :: (\alpha \to \beta \to \delta \to \beta) \to (\delta \to \beta) \to ([\alpha], \delta) \to \beta$$
$$\texttt{pfold}\ h_1\ h_2\ (as, z) = \texttt{foldr}\ (\lambda a\ b \to h_1\ a\ b\ z)\ (h_2\ z)\ as$$

Für ein konkretes Zwischenergebnis (`buildp` g c) bedeutet dies:



Circular Short Cut Deforestation [Fernandes et al. 2007]

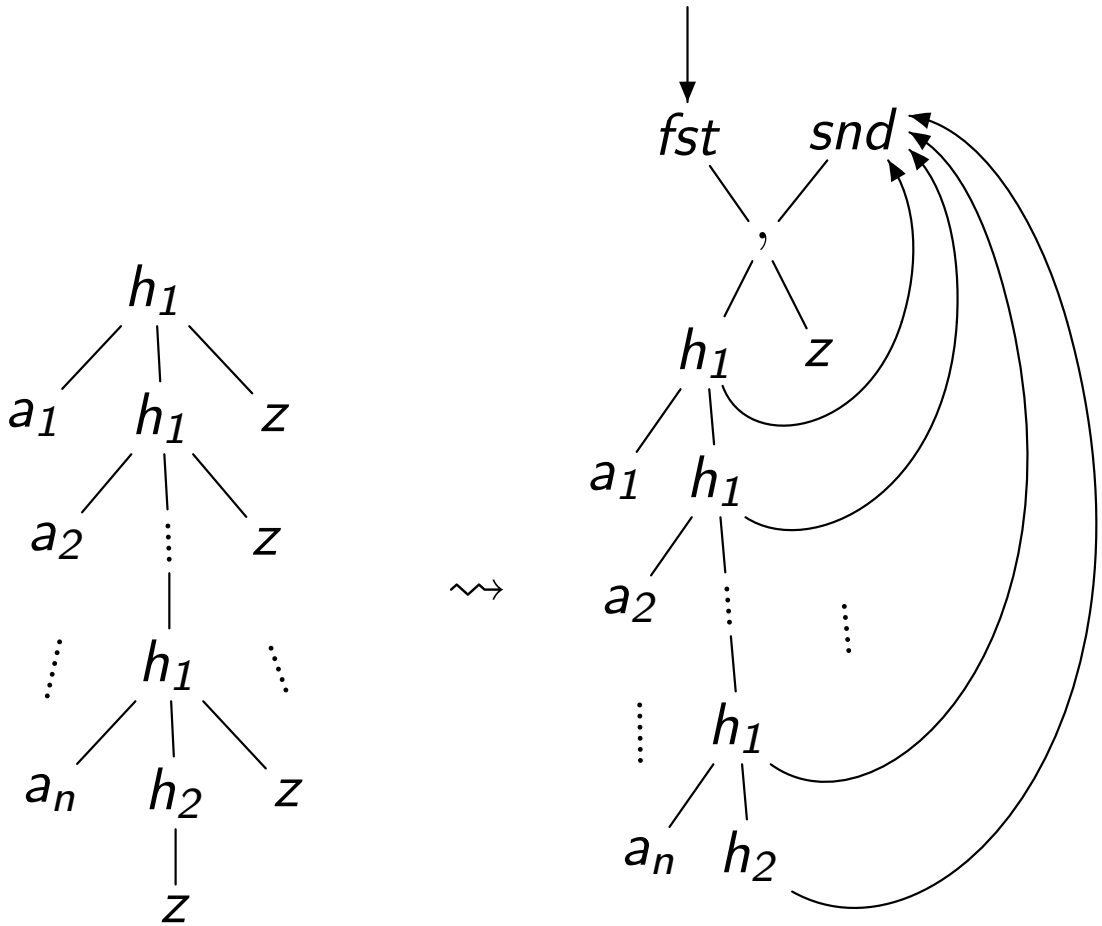
`pfold` $h_1\ h_2\ (g\ (:)\ []\ c) \rightsquigarrow$



$\rightsquigarrow$

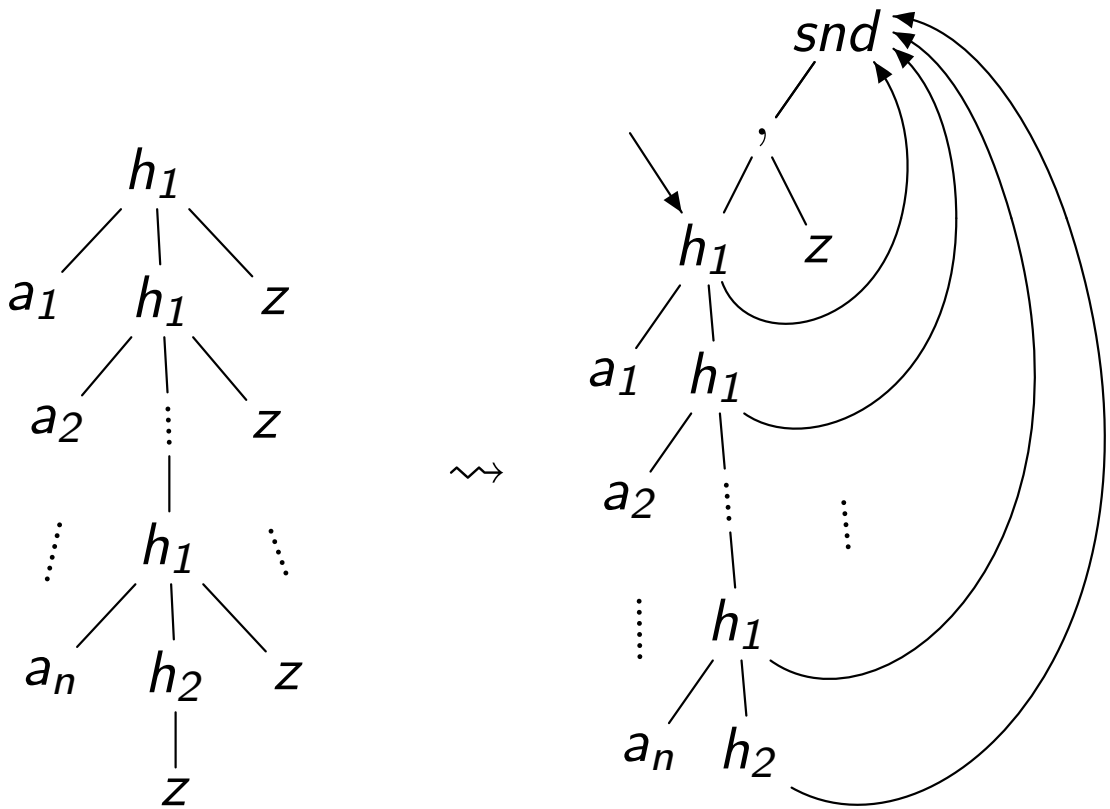
Circular Short Cut Deforestation [Fernandes et al. 2007]

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



Circular Short Cut Deforestation [Fernandes et al. 2007]

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



Circular Short Cut Deforestation [Fernandes et al. 2007]

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