Utrecht University

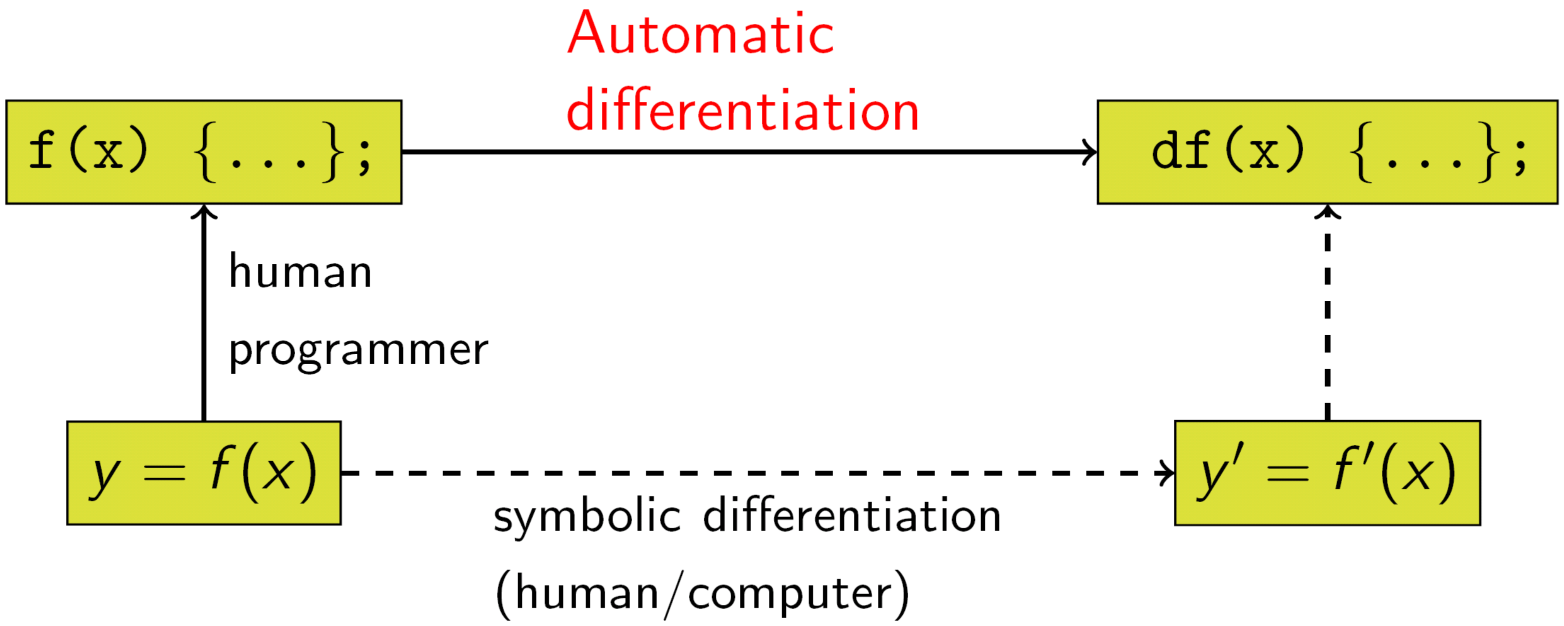
18-12-2022

Efficient dual numbers reverse automatic differentiation by well-known program transformations

Paper by Tom Smeding and Matthijs Vákár
Presentation by Simon van Hus

Automatic Differentiation

- Manual Differentiation
- Symbolic Differentiation
- Numeric Differentiation
- Automatic Differentiation



Automatic Differentiation

- Manual Differentiation
- Symbolic Differentiation
- Numeric Differentiation
- Automatic Differentiation
  - **Forward Mode**

$$
\begin{aligned}
\frac{\partial y}{\partial x} &= \frac{\partial y}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} \\
&= \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x} \right) \\
&= \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \left(\frac{\partial w_{n-2}}{\partial w_{n-3}} \frac{\partial w_{n-3}}{\partial x} \right) \right) \\
&= \cdots
\end{aligned}
$$

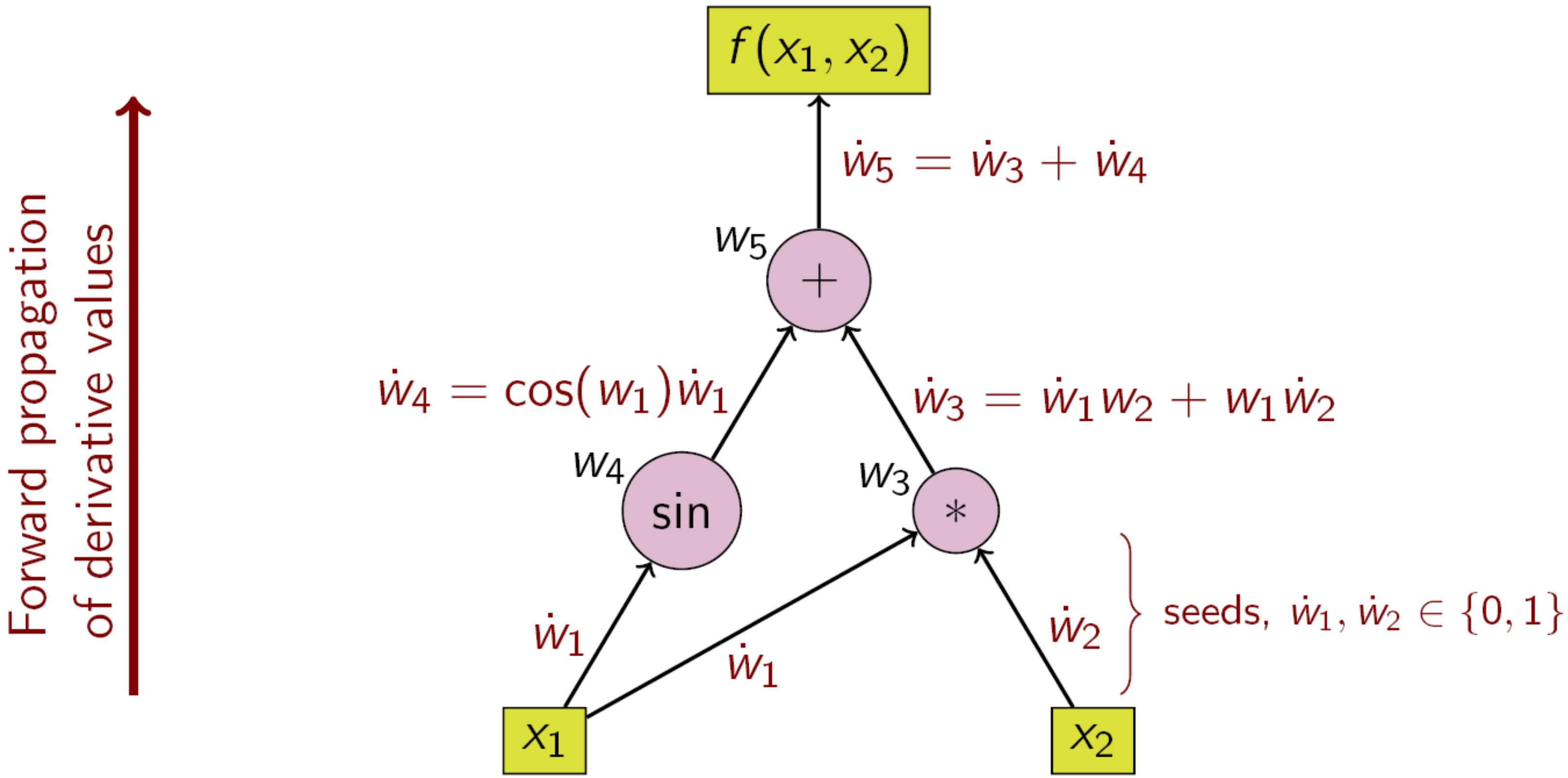
Automatic Differentiation

- Manual Differentiation
- Symbolic Differentiation
- Numeric Differentiation
- Automatic Differentiation
  - **Forward Mode**
  - **Reverse Mode**

$$
\begin{aligned}
\frac{\partial y}{\partial x} &= \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} \\
&= \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} \\
&= \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} \\
&= \cdots
\end{aligned}
$$

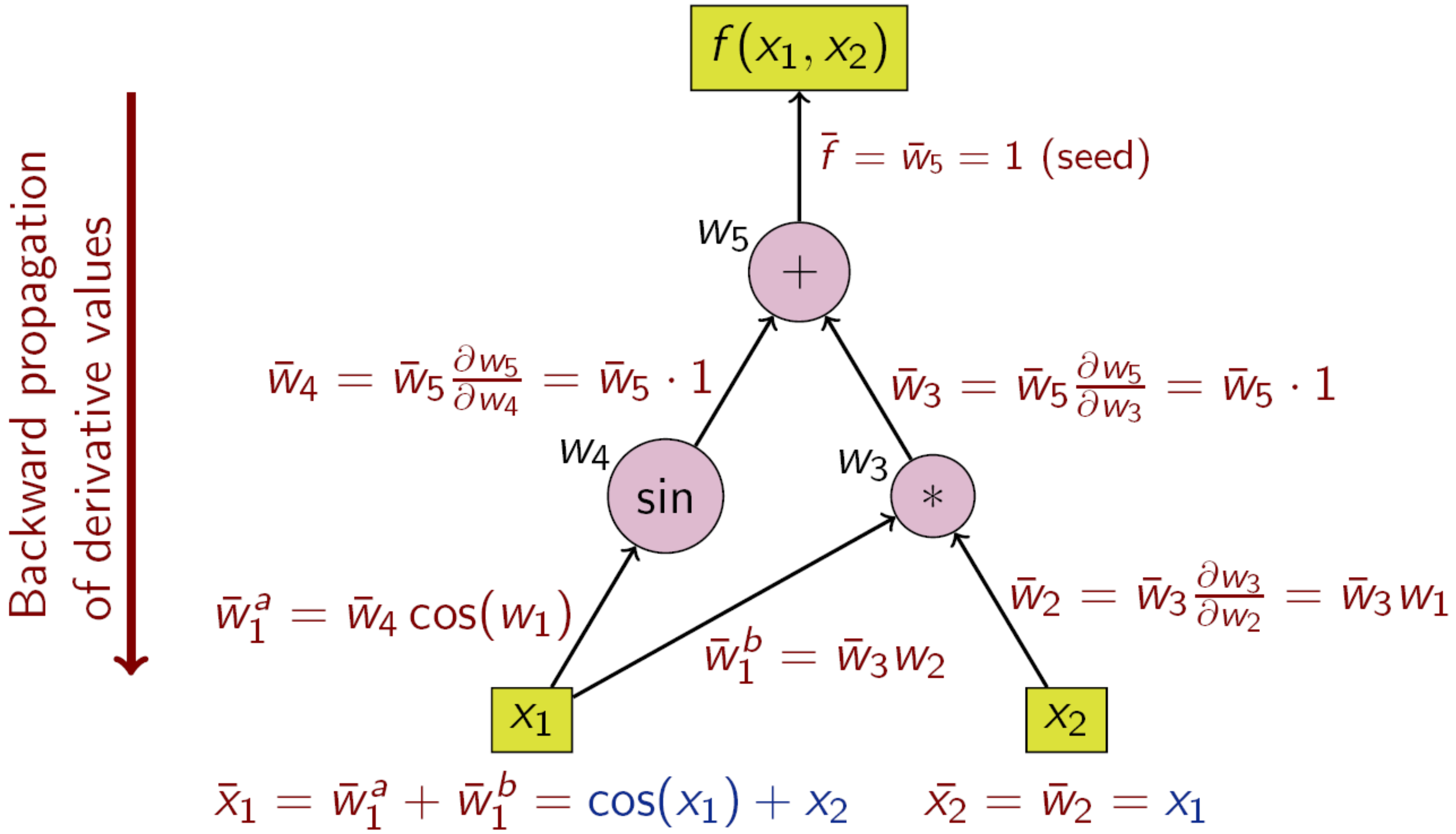
Dual Numbers

- Forward Mode
- **Derivatives**



Dual Numbers

- Forward Mode
- **Derivatives**
- Reverse Mode
- **Adjoints**



Naïve Reverse AD

- Source and Target Languages

Types: $\sigma, \tau ::= \mathbb{R} \mid () \mid (\sigma, \tau) \mid \sigma \to \tau \mid \mathsf{Int}$

Terms:

$$
\begin{array}{lll}
s, t ::= & x \mid () \mid (s, t) \mid \mathrm{fst}(t) \mid \mathrm{snd}(t) \mid s\ t \mid \lambda(x : \tau).\ t \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ t & \\
\mid & r & \text{(literal } \mathbb{R} \text{ values)} \\
\mid & op(t_1, \ldots, t_n) & (op \in \mathrm{Op}_n, \text{ primitive operation application } (\mathbb{R}^n \to \mathbb{R}))
\end{array}
$$

Fig. 4. The source language of all variants of this paper's reverse AD transformation. Int, the type of integers, is added as an example of a type that AD does not act upon.

Types:

$$
\begin{array}{lll}
\sigma, \tau ::= & \mathbb{R} \mid () \mid (\sigma, \tau) \mid \sigma \to \tau \mid \mathsf{Int} & \\
\mid & \sigma \multimap \tau & \text{(linear functions)}
\end{array}
$$

Terms:

$$
\begin{array}{lll}
s, t ::= & x \mid () \mid (s, t) \mid \mathrm{fst}(t) \mid \mathrm{snd}(t) \mid s\ t \mid \lambda(x : \tau).\ t \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ t \mid r \mid op(t_1, \ldots, t_n) & \\
\mid & \underline{\lambda}(z : \tau).\ b & \text{(linear lambda abstraction (} \tau \text{ a type without function arrows))}
\end{array}
$$

Linear function bodies:

$$
\begin{array}{lll}
b ::= & () \mid (b, b') \mid \mathrm{fst}(b) \mid \mathrm{snd}(b) & \text{(tupling)} \\
\mid & z & \text{(reference to } \underline{\lambda}\text{-bound variable)} \\
\mid & x\ b & \text{(linear function application; } x : \sigma \multimap \tau \text{ is an identifier)} \\
\mid & \partial_i op(x_1, \ldots, x_n)(b) & (op \in \mathrm{Op}_n, \ i\text{'th partial derivative of } op \ (\mathbb{R}^n \to \mathbb{R})) \\
\mid & b + b' & \text{(elementwise addition of results)} \\
\mid & \underline{0} & \text{(zero of result type)}
\end{array}
$$

Fig. 5. The target language of the unoptimised variant of the reverse AD transformation. Components that are also in the source language (Fig. 4) are set in grey.

Naïve Reverse AD

- Source and Target Languages
- Linear Functions

$$f(0) = 0$$

$$f(x) + f(y) = f(x+y)$$

Types: $\sigma, \tau \ ::= \ \mathbb{R} \mid () \mid (\sigma, \tau) \mid \sigma \to \tau \mid \mathsf{Int}$

Terms: $s, t \ ::= \ x \mid () \mid (s, t) \mid \mathrm{fst}(t) \mid \mathrm{snd}(t) \mid s\ t \mid \lambda(x : \tau).\ t \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ t$

$\mid\ r$ (literal $\mathbb{R}$ values)

$\mid\ op(t_1, \ldots, t_n)$ ($op \in \mathrm{Op}_n$, primitive operation application ($\mathbb{R}^n \to \mathbb{R}$))

Fig. 4. The source language of all variants of this paper's reverse AD transformation. Int, the type of integers, is added as an example of a type that AD does not act upon.

Types: $\sigma, \tau \ ::= \ \mathbb{R} \mid () \mid (\sigma, \tau) \mid \sigma \to \tau \mid \mathsf{Int}$

$\mid\ \sigma \multimap \tau$ (linear functions)

Terms: $s, t \ ::= \ x \mid () \mid (s, t) \mid \mathrm{fst}(t) \mid \mathrm{snd}(t) \mid s\ t \mid \lambda(x : \tau).\ t \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ t \mid r \mid op(t_1, \ldots, t_n)$

$\mid\ \underline{\lambda}(z : \tau).\ b$ (linear lambda abstraction (τ a type without function arrows))

Linear function bodies:

$b \ ::= \ () \mid (b, b') \mid \mathrm{fst}(b) \mid \mathrm{snd}(b)$ (tupling)

$\mid\ z$ (reference to $\underline{\lambda}$-bound variable)

$\mid\ x\ b$ (linear function application; $x : \sigma \multimap \tau$ is an identifier)

$\mid\ \partial_i op(x_1, \ldots, x_n)(b)$ ($op \in \mathrm{Op}_n$, i'th partial derivative of op ($\mathbb{R}^n \to \mathbb{R}$))

$\mid\ b + b'$ (elementwise addition of results)

$\mid\ \underline{0}$ (zero of result type)

Fig. 5. The target language of the unoptimised variant of the reverse AD transformation. Components that are also in the source language (Fig. 4) are set in grey.

Naïve Reverse AD

- Source and Target Languages
- Linear Functions
- Source Code Transformation

On types: $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \mathbb{R} \multimap c) \qquad \mathbf{D}^1_c[()] = () \qquad \mathbf{D}^1_c[(\sigma,\tau)] = (\mathbf{D}^1_c[\sigma], \mathbf{D}^1_c[\tau])$

$$\mathbf{D}^1_c[\sigma \to \tau] = \mathbf{D}^1_c[\sigma] \to \mathbf{D}^1_c[\tau] \qquad \mathbf{D}^1_c[\mathsf{Int}] = \mathsf{Int}$$

On environments: $\mathbf{D}^1_c[\varepsilon] = \varepsilon \qquad \mathbf{D}^1_c[\Gamma, x : \tau] = \mathbf{D}^1_c[\Gamma], x : \mathbf{D}^1_c[\tau]$

On terms:

If $\Gamma \vdash t : \tau$ then $\mathbf{D}^1_c[\Gamma] \vdash \mathbf{D}^1_c[t] : \mathbf{D}^1_c[\tau]$

$$\mathbf{D}^1_c[x : \tau] = x : \mathbf{D}^1_c[\tau] \qquad \mathbf{D}^1_c[()] = ()$$

$$\mathbf{D}^1_c[(s,t)] = (\mathbf{D}^1_c[s], \mathbf{D}^1_c[t]) \qquad \mathbf{D}^1_c[\mathrm{fst}(t)] = \mathrm{fst}(\mathbf{D}^1_c[t])$$

$$\mathbf{D}^1_c[\mathrm{snd}(t)] = \mathrm{snd}(\mathbf{D}^1_c[t]) \qquad \mathbf{D}^1_c[s\ t] = \mathbf{D}^1_c[s]\ \mathbf{D}^1_c[t]$$

$$\mathbf{D}^1_c[\lambda(x:\tau).\ t] = \lambda(x : \mathbf{D}^1_c[\tau]).\ \mathbf{D}^1_c[t] \qquad \mathbf{D}^1_c[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] = \mathbf{let}\ x : \mathbf{D}^1_c[\tau] = \mathbf{D}^1_c[s]\ \mathbf{in}\ \mathbf{D}^1_c[t]$$

$$\mathbf{D}^1_c[r] = (r, \underline{\lambda}(z : \mathbb{R}).\ \underline{0})$$

$$\begin{aligned}\mathbf{D}^1_c[op(t_1,\ldots,t_n)] = {} & \mathbf{let}\ (x_1,d_1) = \mathbf{D}^1_c[t_1]\ \mathbf{in}\ \ldots\ \mathbf{in}\ \mathbf{let}\ (x_n,d_n) = \mathbf{D}^1_c[t_n] \\ & \mathbf{in}\ (op(x_1,\ldots,x_n) \\ & \quad ,\underline{\lambda}(z:\mathbb{R}).\ d_1\ (\partial_1 op(x_1,\ldots,x_n)(z)) + \cdots + d_n\ (\partial_n op(x_1,\ldots,x_n)(z)))\end{aligned}$$

Fig. 6. The naive code transformation from the source (Fig. 4) to the target (Fig. 5) language. The cases where $\mathbf{D}^1_c$ just maps homomorphically over the source language are set in gray.

Naïve Reverse AD

- Source and Target Languages
- Linear Functions
- Source Code Transformation
- API Wrapper

$$
\begin{aligned}
&\mathrm{Interleave}^1_\tau &&: \forall c.\ (\tau, \tau \multimap c) \to \mathbf{D}^1_c[\tau]\\
&\mathrm{Interleave}^1_\mathbb{R} &&= \lambda(x, d).\ (x, d)\\
&\mathrm{Interleave}^1_{()} &&= \lambda((), d).\ ()\\
&\mathrm{Interleave}^1_{(\sigma,\tau)} &&= \lambda((x, y), d).\ (\mathrm{Interleave}^1_\sigma\ (x, \underline{\lambda}(z : \sigma).\ d\ (z, \underline{0}))\\
& && \qquad\qquad\qquad ,\mathrm{Interleave}^1_\tau\ (y, \underline{\lambda}(z : \tau).\ d\ (\underline{0}, z)))\\
&\mathrm{Interleave}^1_{\mathsf{Int}} &&= \lambda(n, d).\ n\\
&\mathrm{Interleave}^1_{\sigma\to\tau} &&= \text{not defined!}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Deinterleave}^1_\tau &&: \forall c.\ \mathbf{D}^1_c[\tau] \to (\tau, \tau \multimap c)\\
&\mathrm{Deinterleave}^1_\mathbb{R} &&= \lambda(x, d).\ (x, d)\\
&\mathrm{Deinterleave}^1_{()} &&= \lambda().\ ((), \underline{\lambda}(z : ()).\ \underline{0})\\
&\mathrm{Deinterleave}^1_{(\sigma,\tau)} &&= \lambda(x, y).\ \mathbf{let}\ (x_1, x_2) = \mathrm{Deinterleave}^1_\sigma\ x\\
& && \qquad\qquad \mathbf{in\ let}\ (y_1, y_2) = \mathrm{Deinterleave}^1_\tau\ y\\
& && \qquad\qquad \mathbf{in}\ ((x_1, y_1), \underline{\lambda}(z : (\sigma, \tau)).\ x_2\ (\mathrm{fst}(z)) + y_2\ (\mathrm{snd}(z)))\\
&\mathrm{Deinterleave}^1_{\mathsf{Int}} &&= \lambda n.\ (n, \underline{\lambda}(z : \mathsf{Int}).\ \underline{0})\\
&\mathrm{Deinterleave}^1_{\sigma\to\tau} &&= \text{not defined!}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Wrap}^1 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma))\\
&\mathrm{Wrap}^1[\lambda(x : \sigma).\ t] = \lambda(x : \sigma).\ \mathbf{let}\ x : \mathbf{D}^1_\sigma[\sigma] = \mathrm{Interleave}^1_\sigma\ (x, \mathrm{id})\ \mathbf{in}\ \mathrm{Deinterleave}^1_\tau\ (\mathbf{D}^1_\sigma[t])
\end{aligned}
$$

Fig. 7. Wrapper around $\mathbf{D}^1_c$ of Fig. 6.

Linear Factoring

- Duplicate backpropagator calls

$$
\begin{array}{ll}
\lambda(x_0 : \mathbb{R}). & \lambda(x_0 : \mathbb{R}, dx_0 : \mathbb{R} \multimap \mathbb{R}). \\
\quad \textbf{let } x_1 = x_0 + x_0 & \quad \textbf{let } (x_1, dx_1) = (x_0 + x_0, \lambda(d : \mathbb{R}).\ dx_0\ d + dx_0\ d) \\
\quad \textbf{in let } x_2 = x_1 + x_1 & \quad \textbf{in let } (x_2, dx_2) = (x_1 + x_1, \lambda(d : \mathbb{R}).\ dx_1\ d + dx_1\ d) \\
\qquad \vdots \qquad\qquad\qquad \rightsquigarrow & \qquad \vdots \\
\quad \textbf{in let } x_n = x_{n-1} + x_{n-1} & \quad \textbf{in let } (x_n, dx_n) = (x_{n-1} + x_{n-1}, \lambda(d : \mathbb{R}).\ dx_{n-1}\ d + dx_{n-1}\ d) \\
\quad \textbf{in } x_n & \quad \textbf{in } (x_n, dx_n)
\end{array}
$$

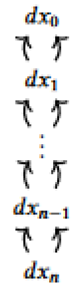


Fig. 2. Left: an example showing how naive dual-numbers reverse AD can result in exponential blow-up when applied to a program with sharing. Right: the dependency graph of the backpropagators dx_i.

Linear Factoring

- Duplicate backpropagator calls

$$
\begin{aligned}
f_1 &= \lambda(z : \mathbb{R}).\ (0, (z, 0)) \\
f_2 &= \lambda(z : \mathbb{R}).\ f_1\ (2 \cdot z) + f_1\ (3 \cdot z) \\
f_3 &= \lambda(z : \mathbb{R}).\ f_2\ (4 \cdot z) + f_1\ (5 \cdot z) \\
f_4 &= \lambda(z : \mathbb{R}).\ f_2\ z + f_3\ (2 \cdot z)
\end{aligned}
$$

$$
\begin{tikzcd}
 & f_3 \arrow[r] \arrow[d] & f_1 \\
f_4 \arrow[ur] \arrow[r] & f_2 \arrow[ur, bend left] \arrow[ur, bend right] &
\end{tikzcd}
$$

Linear Factoring

- Duplicate backpropagator calls
- Linear Factoring

$$
\begin{aligned}
f_1 &= \lambda(z:\mathbb{R}).\ (0,(z,0)) \\
f_2 &= \lambda(z:\mathbb{R}).\ f_1\ (2\cdot z) + f_1\ (3\cdot z) \\
f_3 &= \lambda(z:\mathbb{R}).\ f_2\ (4\cdot z) + f_1\ (5\cdot z) \\
f_4 &= \lambda(z:\mathbb{R}).\ f_2\ z + f_3\ (2\cdot z)
\end{aligned}
$$

$$
\begin{tikzcd}
& f_3 \arrow[r] \arrow[d] & f_1 \\
f_4 \arrow[ur] \arrow[r] & f_2 \arrow[ur, bend left] \arrow[ur, bend right] &
\end{tikzcd}
$$

$$f(x) + f(y) = f(x+y)$$

Linear Factoring

- Duplicate backpropagator calls
- Linear Factoring
- Execution Order

$$\text{Staged}\, c = (c, \text{Map Int}\ (\mathbb{R} \multimap \text{Staged}\, c, \mathbb{R}))$$

$$\begin{aligned} &\text{SCall} &&: (\text{Int}, \mathbb{R} \multimap \text{Staged}\, c) \to \mathbb{R} \multimap \text{Staged}\, c \\ &\text{SCall}\ (i, f)\ x = (\underline{0}, \{i \mapsto (f, x)\}) && \end{aligned}$$

Linear Factoring

- Duplicate backpropagator calls
- Linear Factoring
- Execution Order
- Resolving “Staged c”

$$
\begin{array}{l}
\text{SResolve } (c : \sigma, m : \text{Map Int } (\mathbb{R} \multimap \text{Staged } \sigma, \mathbb{R})) \coloneqq \\
\quad \textbf{if } m \text{ is empty } \textbf{then } c \\
\qquad\qquad\quad \textbf{else let } i = \text{highest key in } m \\
\qquad\qquad\qquad \textbf{in let } (f, a) = \text{lookup } i \text{ in } m \\
\qquad\qquad\qquad \textbf{in let } m' = \text{delete } i \text{ from } m \\
\qquad\qquad\qquad \textbf{in } \text{SResolve } (f\ a +_{\text{Staged}} (c, m'))
\end{array}
$$

Linear Factoring

- Duplicate backpropagator calls
- Linear Factoring
- Execution Order
- Resolving “Staged c”
- New Transformation

On types:

$$\mathbf{D}^2_c[\mathbb{R}] = (\mathbb{R}, (\mathsf{Int}, \mathbb{R} \multimap \mathsf{Staged}\, c)) \qquad \mathbf{D}^2_c[()] = () \qquad \mathbf{D}^2_c[(\sigma,\tau)] = (\mathbf{D}^2_c[\sigma], \mathbf{D}^2_c[\tau])$$

$$\mathbf{D}^2_c[\sigma \to \tau] = \mathbf{D}^2_c[\sigma] \to \mathsf{Int} \to (\mathbf{D}^2_c[\tau], \mathsf{Int}) \qquad \mathbf{D}^2_c[\mathsf{Int}] = \mathsf{Int}$$

On terms:

If $\Gamma \vdash t : \tau$ then $\mathbf{D}^2_c[\Gamma] \vdash \mathbf{D}^2_c[t] : \mathsf{Int} \to (\mathbf{D}^2_c[\tau], \mathsf{Int})$

$$\mathbf{D}^2_c[x : \tau] = \lambda i.\ (x : \mathbf{D}^2_c[\tau], i)$$

$$\mathbf{D}^2_c[(s,t)] = \lambda i.\ \mathbf{let}\ (x, i') = \mathbf{D}^2_c[s]\ i\ \mathbf{in}\ \mathbf{let}\ (y, i'') = \mathbf{D}^2_c[t]\ i'\ \mathbf{in}\ ((x,y), i'')$$

$$\mathbf{D}^2_c[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] = \lambda i.\ \mathbf{let}\ (x : \mathbf{D}^2_c[\tau], i') = \mathbf{D}^2_c[s]\ i\ \mathbf{in}\ \mathbf{D}^2_c[t]\ i'$$

etc.

$$\mathbf{D}^2_c[r] = \lambda i.\ ((r, (i, \underline{\lambda}(z : \mathbb{R}).\ 0_{\mathsf{Staged}})), i + 1)$$

$$\begin{aligned}\mathbf{D}^2_c[op(t_1, \ldots, t_n)] = {} & \\ & \lambda i.\ \mathbf{let}\ ((x_1, d_1), i_1) = \mathbf{D}^2_c[t_1]\ i\ \mathbf{in}\ \ldots\ \mathbf{in}\ \mathbf{let}\ ((x_n, d_n), i_n) = \mathbf{D}^2_c[t_n]\ i_{n-1} \\ & \quad \mathbf{in}\ ((op(x_1, \ldots, x_n), (i_n, \underline{\lambda}(z : \mathbb{R}).\ \mathsf{SCall}\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) +_{\mathsf{Staged}} \cdots +_{\mathsf{Staged}} \\ & \qquad \mathsf{SCall}\ d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))) \\ & \quad , i_n + 1)\end{aligned}$$

Changed wrapper:

$$\mathsf{Wrap}^2 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma))$$

$$\begin{aligned}\mathsf{Wrap}^2[\lambda(x : \sigma).\ t] = \lambda(x : \sigma).\ & \mathbf{let}\ (x : \mathbf{D}^2_\sigma[\sigma], i) = \mathsf{Interleave}^2_\sigma\ (x, \mathsf{SCotan})\ 0 \\ & \mathbf{in}\ \mathbf{let}\ (y, d) = \mathsf{Deinterleave}^2_\tau\ (\mathsf{fst}(\mathbf{D}^2_\sigma[t]\ i)) \\ & \mathbf{in}\ (y, \underline{\lambda}(z : \tau).\ \mathsf{SResolve}\ (d\ z)) \qquad \textit{— see main text for SResolve}\end{aligned}$$

$$\begin{aligned}\mathsf{Interleave}^2_\tau &: \forall c.\ (\tau, \tau \multimap \mathsf{Staged}\, c) \to \mathsf{Int} \to (\mathbf{D}^2_c[\tau], \mathsf{Int}) \\ \mathsf{Interleave}^2_\mathbb{R} &= \lambda(x, d).\ \lambda i.\ ((x, (i, d)), i + 1) \\ \mathsf{Interleave}^2_{()} &= \lambda((), d).\ \lambda i.\ ((), i) \\ \mathsf{Interleave}^2_{(\sigma,\tau)} &= \lambda((x, y), d).\ \lambda i.\ \mathbf{let}\ (x', i') = \mathsf{Interleave}^2_\sigma\ (x, \underline{\lambda}(z : \sigma).\ d\ (z, \underline{0}))\ i \\ & \qquad \mathbf{in}\ \mathbf{let}\ (y', i'') = \mathsf{Interleave}^2_\tau\ (y, \underline{\lambda}(z : \tau).\ d\ (\underline{0}, z))\ i' \\ & \qquad \mathbf{in}\ ((x', y'), i'') \\ \mathsf{Interleave}^2_{\mathsf{Int}} &= \lambda(n, d).\ \lambda i.\ (n, i)\end{aligned}$$

Linear Factoring

- Duplicate backpropagator calls
- Linear Factoring
- Execution Order
- Resolving “Staged c”
- New Transformation

$\mathbf{D}^2_c[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] = \lambda i.\ \mathbf{let}\ (x : \mathbf{D}^2_c[\tau], i') = \mathbf{D}^2_c[s]\ i\ \mathbf{in}\ \mathbf{D}^2_c[t]\ i'$

etc.

$$\mathbf{D}^2_c[r] = \lambda i.\ ((r, (i, \underline{\lambda}(z : \mathbb{R}).\ 0_{\text{Staged}})), i + 1)$$

$$\begin{aligned}
&\mathbf{D}^2_c[op(t_1, \ldots, t_n)] = \\
&\quad \lambda i.\ \mathbf{let}\ ((x_1, d_1), i_1) = \mathbf{D}^2_c[t_1]\ i\ \mathbf{in}\ \ldots\ \mathbf{in}\ \mathbf{let}\ ((x_n, d_n), i_n) = \mathbf{D}^2_c[t_n]\ i_{n-1} \\
&\qquad \mathbf{in}\ ((op(x_1, \ldots, x_n), (i_n, \underline{\lambda}(z : \mathbb{R}).\ \text{SCall}\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) +_{\text{Staged}} \cdots +_{\text{Staged}} \\
&\qquad\qquad \text{SCall}\ d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))) \\
&\qquad\quad , i_n + 1)
\end{aligned}$$

Changed wrapper:

$$\begin{aligned}
&\text{Wrap}^2 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma)) \\
&\text{Wrap}^2[\lambda(x : \sigma).\ t] = \lambda(x : \sigma).\ \mathbf{let}\ (x : \mathbf{D}^2_\sigma[\sigma], i) = \text{Interleave}^2_\sigma\ (x, \text{SCotan})\ 0 \\
&\qquad \mathbf{in\ let}\ (y, d) = \text{Deinterleave}^2_\tau\ (\text{fst}(\mathbf{D}^2_\sigma[t]\ i)) \\
&\qquad \mathbf{in}\ (y, \underline{\lambda}(z : \tau).\ \text{SResolve}\ (d\ z)) \qquad \text{— \textit{see main text for SResolve}}
\end{aligned}$$

$$\begin{aligned}
&\text{Interleave}^2_\tau : \forall c.\ (\tau, \tau \multimap \text{Staged}\, c) \to \text{Int} \to (\mathbf{D}^2_c[\tau], \text{Int}) \\
&\text{Interleave}^2_{\mathbb{R}} = \lambda(x, d).\ \lambda i.\ ((x, (i, d)), i + 1) \\
&\text{Interleave}^2_{()} = \lambda((), d).\ \lambda i.\ ((), i) \\
&\text{Interleave}^2_{(\sigma,\tau)} = \lambda((x, y), d).\ \lambda i.\ \mathbf{let}\ (x', i') = \text{Interleave}^2_\sigma\ (x, \underline{\lambda}(z : \sigma).\ d\ (z, \underline{0}))\ i \\
&\qquad \mathbf{in\ let}\ (y', i'') = \text{Interleave}^2_\tau\ (y, \underline{\lambda}(z : \tau).\ d\ (\underline{0}, z))\ i' \\
&\qquad \mathbf{in}\ ((x', y'), i'') \\
&\text{Interleave}^2_{\text{Int}} = \lambda(n, d).\ \lambda i.\ (n, i)
\end{aligned}$$

$\text{Deinterleave}^2_\tau$ gets type $\forall c.\ \mathbf{D}^2_c[\tau] \to (\tau, \tau \multimap \text{Staged}\, c)$ and ignores the new Int in $\mathbf{D}^2_c[\mathbb{R}]$.
$\underline{0}$ changes to 0_{Staged} and $(+)$ changes to $(+_{\text{Staged}})$.

Staged interface:

$$\text{Staged}\, c = (c, \text{Map Int}\ (\mathbb{R} \multimap \text{Staged}\, c, \mathbb{R}))$$

$$\begin{aligned}
&0_{\text{Staged}} : \text{Staged}\, c && (+_{\text{Staged}}) : \text{Staged}\, c \to \text{Staged}\, c \to \text{Staged}\, c \\
&0_{\text{Staged}} = (\underline{0}, \{\}) && (c, m) +_{\text{Staged}} (c', m') = (c + c', m \cup m') \quad \text{— \textit{with linear factoring}} \\
&\text{SCotan} : c \multimap \text{Staged}\, c && \text{SCall} : (\text{Int}, \mathbb{R} \multimap \text{Staged}\, c) \to \mathbb{R} \multimap \text{Staged}\, c \\
&\text{SCotan}\ c = (c, \{\}) && \text{SCall}\ (i, f)\ x = (\underline{0}, \{i \mapsto (f, x)\})
\end{aligned}$$

Fig. 8. The monadically transformed code transformation (from Fig. 4 to Fig. 5 plus Staged operations), based on Fig. 6. Parts of $\mathbf{D}^2_c$ and Interleave^2 that were simply lifted to monadic code are set in grey.

Current Complexity

- Primal Pass

$$\mathrm{Wrap}^2[P]\ x$$

$$\begin{aligned}
\mathrm{Interleave}^2 &: O(\mathrm{size}(x)) \\
\mathbf{D}^2_\sigma[P] &: O(\mathrm{cost}(P\ x)) \\
\mathrm{Deinterleave}^2 &: O(\mathrm{size}(P\ x))
\end{aligned}$$

$$\mathrm{total} : O(\mathrm{cost}(P\ x) + \mathrm{size}(x))$$

Current Complexity

- Primal Pass
- Reverse Pass

$$\mathrm{snd}(\mathrm{Wrap}^2[P]\ x)\ dy$$

Current Complexity

- Primal Pass
- Reverse Pass
- **Expensive Monoid Operations**

$$O(\mathrm{cost}(P\ x) + \mathrm{size}(x))$$

$$\boxed{\mathrm{snd}(\mathrm{Wrap}^2[P]\ x)}\ dy$$

$$(+_{\mathrm{Staged}}) \quad : \mathrm{Staged}\, c \to \mathrm{Staged}\, c \to \mathrm{Staged}\, c$$
$$(c, m) +_{\mathrm{Staged}} (c', m') = (c + c', m \cup m') \quad \text{— with linear factoring}$$

Current Complexity

- Primal Pass
- Reverse Pass
- **Expensive Monoid Operations**
- **Map operations in SResolve**

$$
\begin{array}{l}
\mathrm{SResolve}\ (c : \sigma, m : \mathrm{Map\ Int}\ (\mathbb{R} \multimap \mathrm{Staged}\ \sigma, \mathbb{R})) := \\
\quad \textbf{if}\ m\ \text{is empty}\ \textbf{then}\ c \\
\qquad\qquad\qquad \textbf{else let}\ i = \text{highest key in}\ m \\
\qquad\qquad\qquad\quad \textbf{in let}\ (f, a) = \text{lookup}\ i\ \text{in}\ m \\
\qquad\qquad\qquad\quad \textbf{in let}\ m' = \text{delete}\ i\ \text{from}\ m \\
\qquad\qquad\qquad\quad \textbf{in}\ \mathrm{SResolve}\ (f\ \ a +_{\mathrm{Staged}} (c, m'))
\end{array}
$$

Current Complexity

- Primal Pass
- Reverse Pass
- **Expensive Monoid Operations**
- **Map operations in SResolve**
- **SCall in backpropagators**

$$(\underline{\lambda}(z:\mathbb{R}).\mathrm{SCotan}(0,\ldots,0,z,0,\ldots,0)) : O(\mathrm{size}(x))$$
$$(\underline{\lambda}(z:\mathbb{R}).0_{\mathrm{Staged}}) : O(\mathrm{size}(x))$$
$$(\underline{\lambda}(z:\mathbb{R}).\mathrm{SCall}\ d_1\ (\partial_1 op(\ldots)(z)) +_{\mathrm{Staged}} \cdots +_{\mathrm{Staged}}\ \mathrm{SCall}\ d_n\ (\partial_n op(\ldots)(z))) : O(\mathrm{size}(x))$$

Staged interface:

$\mathrm{Staged}\,c = (c, \mathrm{Map\ Int}\ (\mathbb{R} \multimap \mathrm{Staged}\,c, \mathbb{R}))$

$0_{\mathrm{Staged}} : \mathrm{Staged}\,c$

$0_{\mathrm{Staged}} = (\underline{0}, \{\})$

$(+_{\mathrm{Staged}}) : \mathrm{Staged}\,c \to \mathrm{Staged}\,c \to \mathrm{Staged}\,c$

$(c, m) +_{\mathrm{Staged}} (c', m') = (c + c', m \cup m')$ — *with linear factoring*

$\mathrm{SCotan} : c \multimap \mathrm{Staged}\,c$

$\mathrm{SCotan}\ c = (c, \{\})$

$\mathrm{SCall} : (\mathrm{Int}, \mathbb{R} \multimap \mathrm{Staged}\,c) \to \mathbb{R} \multimap \mathrm{Staged}\,c$

$\mathrm{SCall}\ (i, f)\ x = (\underline{0}, \{i \mapsto (f, x)\})$

Utrecht University

Cayley Transformation

- Monoid Isomorphism

$$(M, 0, +)$$
$$(M \to M, \mathrm{id}, \circ)$$

Cayley Transformation

- Monoid Isomorphism
- Transforming Staged c

$$(M, 0, +)$$
$$(M \to M, \mathrm{id}, \circ)$$

$$(\text{Staged } c, 0_{\text{Staged}}, +_{\text{Staged}})$$
$$(\text{Staged } c \to \text{Staged } c, \mathrm{id}, \circ)$$

$$\text{Staged } c = (c, \text{Map Int } (\mathbb{R} \multimap (\text{Staged } c \to \text{Staged } c), \mathbb{R}))$$

$$\begin{aligned}
&\text{SCall} &&: (\text{Int}, \mathbb{R} \multimap (\text{Staged } c \to \text{Staged } c)) \to \mathbb{R} \multimap (\text{Staged } c \to \text{Staged } c)\\
&\text{SCotan} &&: c \multimap (\text{Staged } c \to \text{Staged } c)\\
&\text{SResolve} &&: (\text{Staged } c \to \text{Staged } c) \multimap c
\end{aligned}$$

$$\begin{aligned}
\text{SCall } (i, f)\ x\ (c, m) = (c, &\text{if } i \notin m \text{ then insert } i \mapsto (f, x) \text{ into } m\\
&\text{else update } m \text{ at } i \text{ with } (\lambda(_, x').\ (f, x + x')))
\end{aligned}$$

$$\begin{aligned}
&\text{SCotan} : (c \to c) \multimap (\text{Staged } c \to \text{Staged } c)\\
&\text{SCotan } f\ (c, m) = (f\ c, m)
\end{aligned}$$

Cayley Transformation

- Monoid Isomorphism
- Transforming Staged c
- New transformation

On types:

$$\mathbf{D}^3_c[\mathbb{R}] = (\mathbb{R}, (\mathsf{Int}, \mathbb{R} \multimap (\mathsf{Staged}\, c \to \mathsf{Staged}\, c))) \qquad \mathbf{D}^3_c[()] = () \qquad \mathbf{D}^3_c[(\sigma,\tau)] = (\mathbf{D}^3_c[\sigma], \mathbf{D}^3_c[\tau])$$

$$\mathbf{D}^3_c[\sigma \to \tau] = \mathbf{D}^3_c[\sigma] \to \mathsf{Int} \to (\mathbf{D}^3_c[\tau], \mathsf{Int}) \qquad \mathbf{D}^3_c[\mathsf{Int}] = \mathsf{Int}$$

On terms:

If $\Gamma \vdash t : \tau$ then $\mathbf{D}^3_c[\Gamma] \vdash \mathbf{D}^3_c[t] : \mathsf{Int} \to (\mathbf{D}^3_c[\tau], \mathsf{Int})$

Same as $\mathbf{D}^2_c$, except with 'id' in place of 0_{Staged} and '$\circ$' in place of $(+_{\mathsf{Staged}})$.

Changed wrapper:

$$\begin{aligned}
&\mathrm{Wrap}^3 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma))\\
&\mathrm{Wrap}^3[\lambda(x:\sigma).\ t] = \lambda(x:\sigma).\ \mathbf{let}\ (x : \mathbf{D}^3_\sigma[\sigma], i) = \mathrm{Interleave}^3_\sigma\ (x, \mathrm{SCotan})\ 0\\
&\qquad\qquad\qquad\qquad\qquad\ \mathbf{in\ let}\ (y,d) = \mathrm{Deinterleave}^3_\tau\ (\mathrm{fst}(\mathbf{D}^3_\sigma[t]\ i))\\
&\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\ (y, \underline{\lambda}(z:\tau).\ \mathrm{SResolve}\ (d\ z))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Interleave}^3_\tau &&: \forall c.\ (\tau, (\tau \to \tau) \multimap (\mathsf{Staged}\, c \to \mathsf{Staged}\, c)) \to \mathsf{Int} \to (\mathbf{D}^3_c[\tau], \mathsf{Int})\\
&\mathrm{Interleave}^3_{\mathbb{R}} &&= \lambda(x,d).\ \lambda i.\ ((x, (i, \underline{\lambda}(z:\mathbb{R}).\ d\ (\lambda(a:\mathbb{R}).\ z + a)))\\
& && \qquad\qquad\quad , i+1)\\
&\mathrm{Interleave}^3_{()} &&= \lambda((),d).\ \lambda i.\ ((), i)\\
&\mathrm{Interleave}^3_{(\sigma,\tau)} &&= \lambda((x,y),d).\ \lambda i.\\
& &&\quad \mathbf{let}\ (x', i') = \mathrm{Interleave}^3_\sigma\ (x, \lambda(f : \sigma \to \sigma).\ d\ (\lambda((v,w) : (\sigma,\tau)).\ (f\ v, w)))\ i\\
& &&\quad \mathbf{in\ let}\ (y', i'') = \mathrm{Interleave}^3_\tau\ (y, \lambda(f : \tau \to \tau).\ d\ (\lambda((v,w) : (\sigma,\tau)).\ (v, f\ w)))\ i'\\
& &&\quad \mathbf{in}\ ((x', y'), i'')\\
&\mathrm{Interleave}^3_{\mathsf{Int}} &&= \lambda(n,d).\ \lambda i.\ (n, i)
\end{aligned}$$

$$\mathrm{Deinterleave}^3_\tau : \forall c.\ \mathbf{D}^3_c[\tau] \to (\tau, \tau \multimap (\mathsf{Staged}\, c \to \mathsf{Staged}\, c))$$

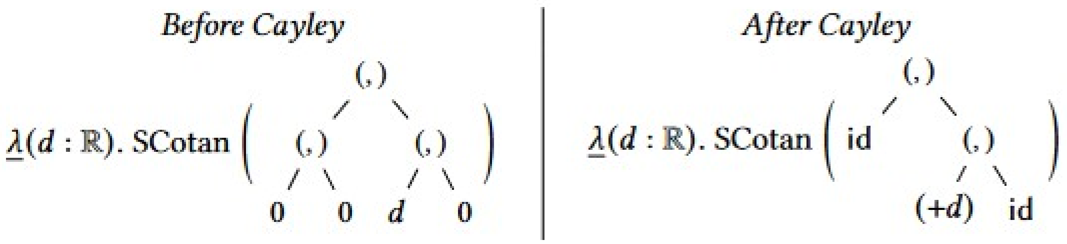
(Same as $\mathrm{Deinterleave}^2$ in Fig. 8, except with id and $(\circ)$ in place of 0_{Staged} and $(+_{\mathsf{Staged}})$)

Fig. 9. The Cayley-transformed code transformation, based on Fig. 8. Grey parts are unchanged.

Utrecht University

Cayley Transformation

- Monoid Isomorphism
- Transforming Staged c
- New transformation
- Remaining Problem



Efficient Gradient Updates

- Only collect scalars

$$\text{Staged}\, c = (c, \text{Map Int}\ (\mathbb{R} \multimap (\text{Staged}\, c \rightarrow \text{Staged}\, c), \mathbb{R}))$$

$$\text{Staged}\, c = (\text{Map Int}\ \mathbb{R}, \text{Map Int}\ (\mathbb{R} \multimap (\text{Staged}\, c \rightarrow \text{Staged}\, c), \mathbb{R}))$$

Efficient Gradient Updates

- Only collect scalars
- Remaining problems:
- **Expensive Monoid Operations**
- **Map Operations in SResolve**
- **SCall in backpropagators**

$$O((\mathrm{cost}(P\ x) + \mathrm{size}(x)) \log(\mathrm{cost}(P\ x) + \mathrm{size}(x)))$$

Efficient Gradient Updates

- Only collect scalars
- Remaining problems:
  - ~~**Expensive Monoid Operations**~~
  - **Map Operations in SResolve**
  - ~~**SCall in backpropagators**~~

$$O((\text{cost}(P\ x) + \text{size}(x)) \log(\text{cost}(P\ x) + \text{size}(x)))$$

Utrecht University

Mutable Arrays

- Map operations in SResolve

$$
\begin{aligned}
&\mathrm{SResolve}\ (c : \sigma, m : \mathrm{Map\ Int}\ (\mathbb{R} \multimap \mathrm{Staged}\, \sigma, \mathbb{R})) :=\\
&\quad \textbf{if}\ m\ \text{is empty}\ \textbf{then}\ c\\
&\qquad\qquad\qquad \textbf{else let}\ i = \text{highest key in}\ m\\
&\qquad\qquad\qquad\quad \textbf{in let}\ (f, a) = \text{lookup}\ i\ \text{in}\ m\\
&\qquad\qquad\qquad\quad \textbf{in let}\ m' = \text{delete}\ i\ \text{from}\ m\\
&\qquad\qquad\qquad\quad \textbf{in}\ \mathrm{SResolve}\ (f\ a +_{\mathrm{Staged}} (c, m'))
\end{aligned}
$$

Utrecht University

Mutable Arrays

- Map operations in SResolve
- Maps to Arrays

$$\mathsf{Staged}\,c = (\mathsf{Map\ Int}\ \mathbb{R}, \mathsf{Map\ Int}\ (\mathbb{R} \multimap (\mathsf{Staged}\,c \longrightarrow \mathsf{Staged}\,c), \mathbb{R}))$$

$$\mathsf{Staged}\,c = (\mathsf{Array}\quad \mathbb{R}, \mathsf{Array}\quad (\mathbb{R} \multimap (\mathsf{Staged}\,c \xrightarrow{R} \mathsf{Staged}\,c), \mathbb{R}))$$

Utrecht University

More Optimizations

- No cotangent collection array

$$(\mathrm{Array}\quad \mathbb{R}, \mathrm{Array}\quad (\mathbb{R} \multimap (\mathrm{Staged}\, c \xrightarrow{R} \mathrm{Staged}\, c), \mathbb{R}))$$

$$\mathrm{Array}\,(\mathbb{R} \multimap (\mathrm{Staged}\, c \xrightarrow{R} \mathrm{Staged}\, c), \mathbb{R})$$

More Optimizations

- No cotangent collection array
- Defunctionalization of backpropagators

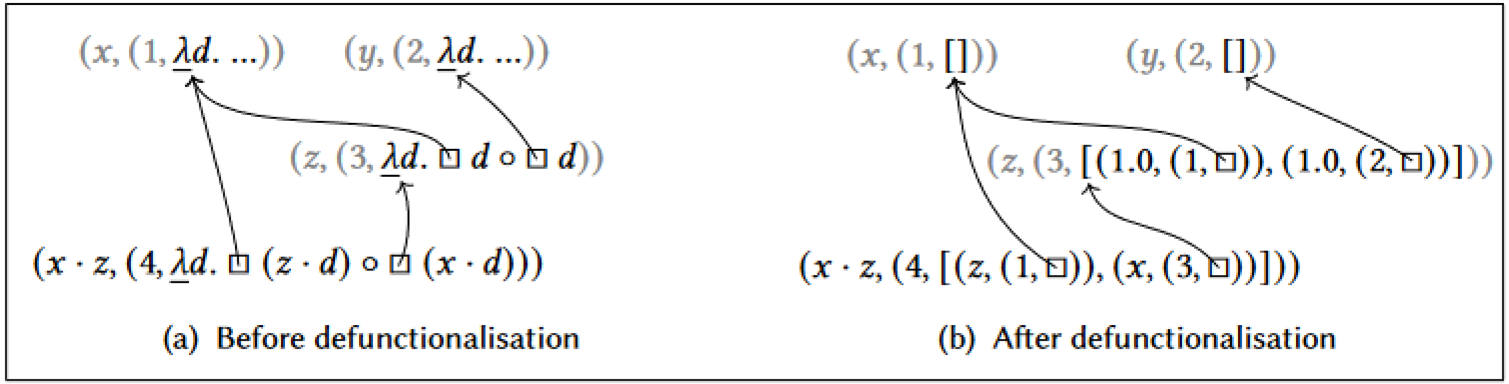
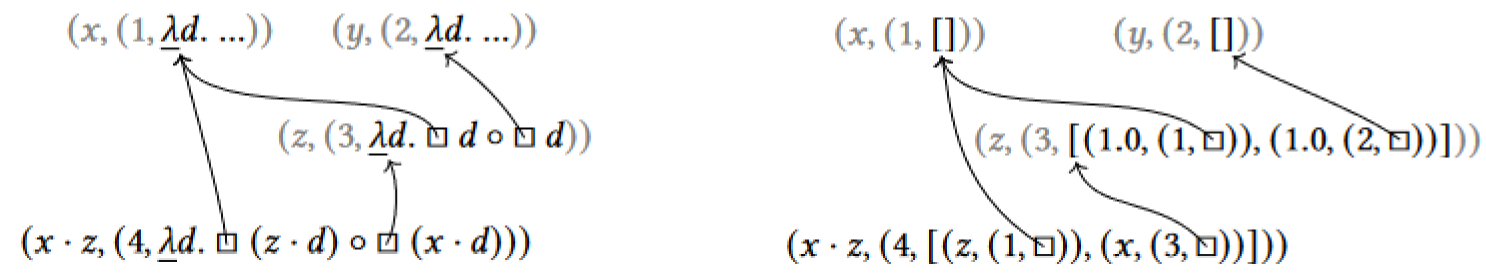


Fig. 10. The sharing structure before and after defunctionalisation. SCall is elided here; in Fig. 10a, the backpropagator calls are depicted as if they are still normal calls. Boxes (□) are the same in-memory value as the value their arrow points to; two boxes pointing to the same value indicates that this value is *shared*: referenced in two places.

More Optimizations

- No cotangent collection array
- Defunctionalization of backpropagators
- Taping



(a) Before defunctionalisation

(b) After defunctionalisation

Fig. 10. The sharing structure before and after defunctionalisation. SCall is elided here; in Fig. 10a, the backpropagator calls are depicted as if they are still normal calls. Boxes (□) are the same in-memory value as the value their arrow points to; two boxes pointing to the same value indicates that this value is *shared*: referenced in two places.

Summary

- Dual-Numbers Reverse AD
- Linear Factoring
- Cayley Transformation
- Efficient Gradient Updates
- Mutable Arrays
- Even more optimizations

Summary

- Dual-Numbers Reverse AD
- Linear Factoring
- Cayley Transformation
- Efficient Gradient Updates
- Mutable Arrays
- Even more optimizations
- Extensions to the source language
- Implementation and Performance

Questions?

Presentation by Simon van Hus