Reverse-Mode Automatic Differentiation

Cosmin E. Oancea in collaboration with
Robert Schenck and Troels Henriksen
`cosmin.oancea@diku.dk`

Department of Computer Science (DIKU)
University of Copenhagen

December 2020 PFP Lecture Slides

Material for This Lecture

The material presented in this lecture was gathered from:

- “Automatic Differentiation in Machine Learning: a Survey”, A. G. Baydin, B. A. Pearlmutter, A. A. Radul and J. M. Siskind, *Journal of Machine Learning Research*, Vol. 18, pages: 1−43, 2018;
- High-School Math Curiculum (wikipedia helps to brush up);
- Thoughts by R. Schenck, T. Henriksen and C. E. Oancea, on how to apply reverse-mode automatic differentiation to programs written in a data-parallel language.

Math: Simple Differentiation Rules $h : \mathbb{R} \to \mathbb{R}$

Basic Rules:

sin: $\frac{\partial sin\ x}{\partial x} = cos\ x$

cos: $\frac{\partial cos\ x}{\partial x} = -sin\ x$

pow: $\frac{\partial x^r}{\partial x} = r \cdot x^{r-1},\ \ \forall r \in \mathbb{R},\ r \neq 0$

log: $\frac{\partial log_c\ x}{\partial x} = \frac{1}{x \cdot ln\ c}$ where $ln\ x = log_e\ x, e = 2.718281828459...$ Euler's number.

- ...

How do we combine this rules?

Math: Simple Differentiation Rules $h : \mathbb{R} \to \mathbb{R}$

Basic Rules:

sin: $\frac{\partial sin\ x}{\partial x} = cos\ x$

cos: $\frac{\partial cos\ x}{\partial x} = -sin\ x$

pow: $\frac{\partial x^r}{\partial x} = r \cdot x^{r-1}, \ \forall r \in \mathbb{R}, \ r \neq 0$

log: $\frac{\partial log_c\ x}{\partial x} = \frac{1}{x \cdot ln\ c}$ where $ln\ x = log_e\ x, e = 2.718281828459\ldots$ Euler's number.

- ...

How do we combine this rules?

Linear: If $h(x) = a \cdot f(x) \ + \ b \cdot g(x)$ then $\frac{\partial h}{\partial x} = a \cdot \frac{\partial f}{\partial x} \ + \ b \cdot \frac{\partial g}{\partial x}$

Product: If $h(x) = f(x) \cdot g(x)$ then $\frac{\partial h}{\partial x} = \frac{\partial f}{\partial x} \cdot g \ + \ f \cdot \frac{\partial g}{\partial x}$

Quotient: If $h(x) = \frac{f(x)}{g(x)}$ then $\frac{\partial h}{\partial x} = \frac{\frac{\partial f}{\partial x} \cdot g \ + \ \frac{\partial g}{\partial x} \cdot f}{g^2}$

GenPower: If $h(x) = f(x)^{g(x)}$ then $\frac{\partial h}{\partial x} = f^g \cdot (\frac{\partial f}{\partial x} \cdot \frac{g}{f} \ + \ \frac{\partial g}{\partial x} \cdot ln\ f)$

GenLog: If $h(x) = ln(f(x))$ then $\frac{\partial h}{\partial x} = \frac{\frac{\partial f}{\partial x}}{f}$, where f is positive.

I have not mentioned the most important rule (?)

Math: Chain Rule

If $h(x) = (f \circ g)(x) = f(g(x))$ then $\frac{\partial h(x)}{\partial x} = \frac{\partial f(z)|_{z=g(x)}}{\partial z} \cdot \frac{\partial g(x)}{\partial x}$

Note: the differential in a point x is a function that might be specific to the value of x.

Math: Chain Rule

If $h(x) = (f \circ g)(x) = f(g(x))$ then $\frac{\partial h(x)}{\partial x} = \frac{\partial f(z)|_{z=g(x)}}{\partial z} \cdot \frac{\partial g(x)}{\partial x}$

Note: the differential in a point x is a function that might be specific to the value of x.

1^{st} Generalization: partial derivatives of $f : \mathbb{R}^n \to \mathbb{R}$:

Intuition: $f : \mathbb{R}^2 \to \mathbb{R},\ f(y,x) = x^2 + xy + y^2$
Fix $y = a$, define $f_a : \mathbb{R} \to \mathbb{R},\ f_a(x) = f(a,x) = x^2 + ax + a^2$
$\frac{\partial f_a(x)}{\partial x} = 2x + a$, i.e., a different function for each instance of y.

The i^{th} partial derivative of f in point $a \in \mathbb{R}^n$ is defined as:
$\frac{\partial f(a_1,\ldots,a_n)}{\partial x_i}(x) = lim_{h\to 0}\frac{f(a_1,\ldots,a_{i-1},x+h,a_{i+1},\ldots,a_n)-f(a_1,\ldots,a_{i-1},x,a_{i+1},\ldots,a_n)}{h}$

Math: Chain Rule

If $h(x) = (f \circ g)(x) = f(g(x))$ then $\frac{\partial h(x)}{\partial x} = \frac{\partial f(z)|_{z=g(x)}}{\partial z} \cdot \frac{\partial g(x)}{\partial x}$

Note: the differential in a point x is a function that might be specific to the value of x.

1st Generalization: partial derivatives of $f : \mathbb{R}^n \to \mathbb{R}$:

Intuition: $f : \mathbb{R}^2 \to \mathbb{R},\ f(y,x) = x^2 + xy + y^2$
Fix $y = a$, define $f_a : \mathbb{R} \to \mathbb{R},\ f_a(x) = f(a,x) = x^2 + ax + a^2$
$\frac{\partial f_a(x)}{\partial x} = 2x + a$, i.e., a different function for each instance of y.

The i^{th} partial derivative of f in point $a \in \mathbb{R}^n$ is defined as:
$\frac{\partial f(a_1,\ldots,a_n)}{\partial x_i}(x) = lim_{h\to 0}\frac{f(a_1,\ldots,a_{i-1},x+h,a_{i+1},\ldots,a_n) - f(a_1,\ldots,a_{i-1},x,a_{i+1},\ldots,a_n)}{h}$

If all partial derivatives exist in a neighborhood of $a \in \mathbb{R}^n$ and are continuous there, then f is totally differentiable in that neighborhood and the total derivative is continuous.

Math: Chain Rule for Multi-Variate Functions

Consider $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, and $f(x) = (f_1(x), \ldots, f_m(x))$

$$J_f(a) = \left[\frac{\partial f(a)}{\partial x_1}(a_1) \ldots \frac{\partial f(a)}{\partial x_n}(a_n)\right] = \begin{bmatrix} \frac{\partial f_1(a)}{\partial x_1}(a_1) & \ldots & \frac{\partial f_1(a)}{\partial x_n}(a_n) \\ & \ldots & \\ \frac{\partial f_m(a)}{\partial x_1}(a_1) & \ldots & \frac{\partial f_m(a)}{\partial x_n}(a_n) \end{bmatrix}$$

$$\frac{\partial f(a)}{\partial x}(y) = J_f(a) \cdot y$$

Math: Chain Rule for Multi-Variate Functions

Consider $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, and $f(x) = (f_1(x), \ldots, f_m(x))$

$$J_f(a) = \left[\frac{\partial f(a)}{\partial x_1}(a_1) \ldots \frac{\partial f(a)}{\partial x_n}(a_n) \right] = \begin{bmatrix} \frac{\partial f_1(a)}{\partial x_1}(a_1) & \ldots & \frac{\partial f_1(a)}{\partial x_n}(a_n) \\ & \ldots & \\ \frac{\partial f_m(a)}{\partial x_1}(a_1) & \ldots & \frac{\partial f_m(a)}{\partial x_n}(a_n) \end{bmatrix}$$

$$\frac{\partial f(a)}{\partial x}(y) = J_f(a) \cdot y$$

- f needs not be differentiable for its Jacobian to be defined, i.e., only its 1^{st}-order partial derivatives have to exist;
- If f is differentiable at $a \in \mathbb{R}^n$, then its differential is $J_f(a)$, i.e., $J_f(a)$ is the best linear approximation of f near point a: $f(x) - f(a) = J_f(a) \cdot (x - a) \; + \; o(||x - a||)$ as $x \rightarrow a$

Chain Rule for $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $g : \mathbb{R}^m \rightarrow \mathbb{R}^k$ and $p \in \mathbb{R}^n$ is:

$$J_{g \circ f}(p) = J_g(f(p)) \cdot J_f(p)$$

Chain Rule: Intuition on Differentiating a Program

Consider $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, \forall i = 1 \dots m$, and "program" P defined as:

$P : \mathbb{R}^{n_0} \to \mathbb{R}^{n_m} = f_m \circ f_{m-1} \circ \dots \circ f_2 \circ f_1$

Applying the Chain Rule for some $x \in \mathbb{R}^{n_0}$ results in:

$$J_P(x) = J_{f_m}(f_{m-1}(\dots f_1(x))) \cdot \dots \cdot J_{f_2}(f_1(x)) \cdot J_{f_1}(x)$$

Should compute $J_P(x)$, from right-to-left or from left-to-right?

Chain Rule: Intuition on Differentiating a Program

Consider $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, \forall i = 1 \ldots m$, and "program" P defined as:

$P : \mathbb{R}^{n_0} \to \mathbb{R}^{n_m} = f_m \circ f_{m-1} \circ \ldots \circ f_2 \circ f_1$

Applying the Chain Rule for some $x \in \mathbb{R}^{n_0}$ results in:

$$J_P(x) = J_{f_m}(f_{m-1}(\ldots f_1(x))) \cdot \ldots \cdot J_{f_2}(f_1(x)) \cdot J_{f_1}(x)$$

Should compute $J_P(x)$, from right-to-left or from left-to-right?

$\leftarrow$ Forward mode (from right to left)

- when the length of the input is less than or comparable to the length of the output, i.e., $n_0 \leq n_m$ or $n_0 \sim n_m$;
- the jacobian J_{f_i} is computed in the same time as $f_i(x)$;
- Example: consider $n_0 = 1$ and $n_i = n, \forall 0 < i \leq m$.
  - $J_{f_0}(x) \in \mathbb{M}^{n\times 1}$, but $J_{f_i}(x) \in \mathbb{M}^{n\times n}$, $i > 0$.
  - $O(J_P(x))$ = $O(m \times n^2)$ instead of $O(m \times n^3)$ if you do it the other way around.

Chain Rule: Intuition on Differentiating a Program

Consider $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, \forall i = 1 \dots m$, and "program" P defined as:

$P : \mathbb{R}^{n_0} \to \mathbb{R}^{n_m} = f_m \circ f_{m-1} \circ \dots \circ f_2 \circ f_1$

Applying the Chain Rule for some $x \in \mathbb{R}^{n_0}$ results in:

$$J_P(x) = J_{f_m}(f_{m-1}(\dots f_1(x))) \cdot \dots \cdot J_{f_2}(f_1(x)) \cdot J_{f_1}(x)$$

Should compute $J_P(x)$, from right-to-left or from left-to-right?

$\to$ Reverse mode (from left to right)

- when the length of the output is significantly less than the length of the input, i.e., $n_0 >> n_m$;
- typically the common case for AI;
- $f_{m-1}(\dots f_1(x))$ needs to be computed before we start computing the Jacobians, hence we need to save on "tape" the values of intermediate variables.
- Example: consider $n_m = 1$ and $n_i = n, \forall 0 \leq i < m$.
  - $O(J_P(x))$ = $O(m \times n^2)$ instead of $O(m \times n^3)$ if you apply the forward mode.

What is Automatic Differentiation (AD)?

What is not AD:

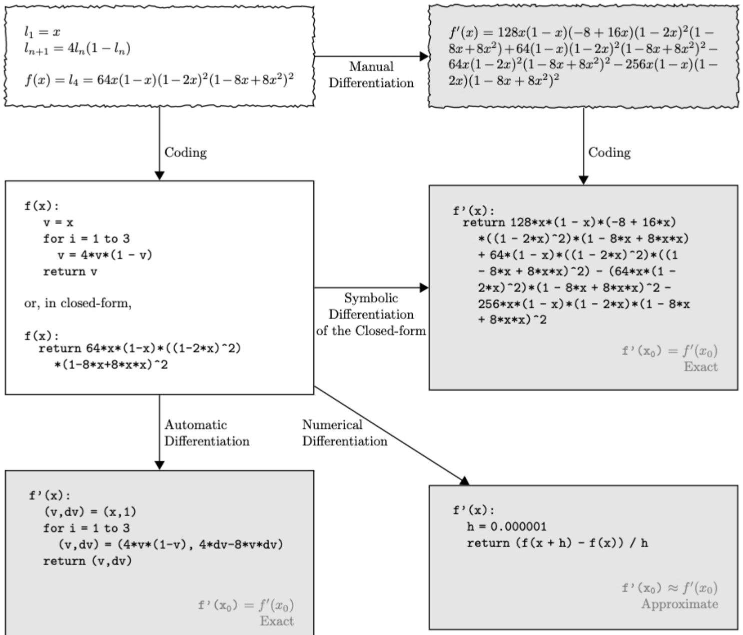
- manually working out derivatives and coding them: time consuming and error prone;
- numerical differentiation using finite-difference approximation:
  - may be highly innacurate due to round-off and truncation error;
  - it scales poorly to gradients (where gradients w.r.t. million of parameters are needed)
- symbolic differentiation in compute-algebra systems such as Mathematica, Maxima, Maple:
  - solves the problems above, but
  - may suffer ”expression swell” + cryptic and complex expressions;
  - requires models to be defined as closed-formed expressions (formula), severely limiting control flow, expressivity (and the application of compiler-like optimizations).

What is Automatic Differentiation (AD)?

“Automatic (or algorithmic) differentiation performs a non-standard interpretation of a given program by replacing the domain of variables to incorporate derivative values and redifinging the semantics of operators to propagate derivatives per the chain rule of differential calculus.” [Baydin et. al. 2018].

AD can be applied to regular code with minimal change, allowing branching, loops, and even recursion.

What is AD? [Baydin et. al. 2018]


Manual Differentiation
Coding
Coding
Symbolic Differentiation of the Closed-form
Automatic Differentiation
Numerical Differentiation

$$l_1 = x$$
$$l_{n+1} = 4l_n(1-l_n)$$
$$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

$$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$$

```
f(x):
  v = x
  for i = 1 to 3
    v = 4*v*(1 - v)
  return v
```

or, in closed-form,

```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

```
f'(x):
  return 128*x*(1 - x)*(-8 + 16*x)
    *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
    + 64*(1 - x)*((1 - 2*x)^2)*((1
    - 8*x + 8*x*x)^2) - (64*x*(1 -
    2*x)^2)*(1 - 8*x + 8*x*x)^2 -
    256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
    + 8*x*x)^2
```

`f'(x₀)` $= f'(x_0)$
Exact

```
f'(x):
  (v,dv) = (x,1)
  for i = 1 to 3
    (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
  return (v,dv)
```

`f'(x₀)` $= f'(x_0)$
Exact

```
f'(x):
  h = 0.000001
  return (f(x + h) - f(x)) / h
```

`f'(x₀)` $\approx f'(x_0)$
Approximate

Running Example and Notation

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Running example: $y = f(x_1, x_2) = ln(x_1) + x_1 \cdot x_2 - sin(x_2)$

Will use the notation used by Griewank and Walther (2008), where a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is constructed from variables v_i:

- $v_{i-n} = x_i,\ i = 1, \ldots, n$ are the input variables;
- $v_i,\ i = 1, \ldots, l$ are the intermediate variables;
- $y_{m-i} = v_{l-i},\ i = m-1, \ldots, 0$ are the output variables.

AD is blind w.r.t. any operation, including control flow statements, which do not directly alter numeric values.

Forward Mode: Main Intution

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Running example: $y = f(x_1, x_2) = ln(x_1) + x_1 \cdot x_2 - sin(x_2)$

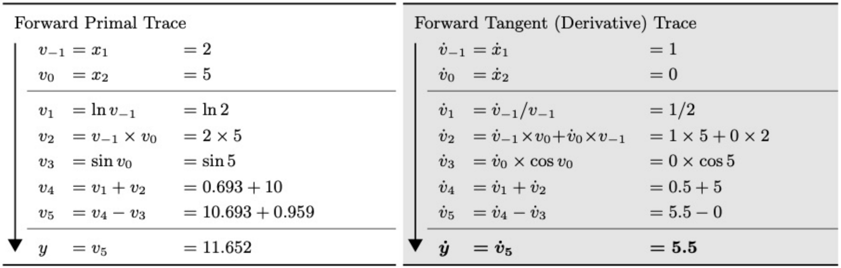
- For computing derivative w.r.t. x_1, we associate with each intermediate value v_i a derivative $\dot{v_i} = \frac{\partial v_i}{\partial x_1}$
- applying the chain rule to each elementary operation in the formal primal trace (i.e., original program) results in the corresponding tangent (derivative) trace on the right.
- evaluating the primals v_i in lockstep with their tangents $\dot{v_i}$ gives the required derivative in final variable $\dot{v_5} = \frac{\partial y}{\partial x_1}$

Forward Mode: Applied to Basic-Block Code

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Running example: $y = f(x_1, x_2) = ln(x_1) + x_1 \cdot x_2 - sin(x_2)$

Table 2: Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$ and setting $\dot{x}_1 = 1$ to compute $\frac{\partial y}{\partial x_1}$. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.



Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\boldsymbol{\dot{y}}$	$\boldsymbol{= \dot{v}_5}$	$\mathbf{= 5.5}$

Forward Mode: Generalization to Multiple Dimensions

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Assume $f : \mathbb{R}^n \to \mathbb{R}^m$ with input $x_{1 \dots n}$ and output $y_{1 \dots m}$:

- requires n runs of the code,
- in which run i initializes $\dot{x} = e_i$ (the i-th unit vector), $x = a$
- and computes one column of the Jacobian matrix $J_f(a)$:

$$\dot{y}_j = \frac{\partial y_j}{\partial x_i}|_{x=a},\ j = 1, \dots, m$$

Essentially, a `map` operation over the n unit vectors, where the unnamed function $\lambda \dot{x}_i \to \dots$ implements the forward mode generically for each value of $\dot{x}_i = e_i$.

Forward Mode: Dual Numbers

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Forward mode seen as evaluating a function by dual numbers: $v + \dot{v}\epsilon$, where $v, \dot{v} \in \mathbb{R}$ and $\epsilon \neq 0$, but $\epsilon^2 = 0$:

- $(v + \dot{v}\epsilon) \ + \ (u + \dot{u}\epsilon) \ = \ (v + u) \ + \ (\dot{v} + \dot{u})\epsilon$
- $(v + \dot{v}\epsilon) \ \cdot \ (u + \dot{u}\epsilon) \ = \ (vu) \ + \ (v\dot{u} + \dot{v}u)\epsilon$
- Seeting up a regime such as $f(v + \dot{v}\epsilon) \ = \ f(v) \ + \ f'(v)\dot{v}\epsilon$ makes the chain rule work as expected: $f(g(v + \dot{v}\epsilon)) \ = \ f(g(v)) \ + \ f'(g(v))g'(v)\dot{v}\epsilon$

$\frac{\partial f(x)}{\partial x}|_{x=v} \ = \ \texttt{espsilon-coeff(dual-version(f)}(v + 1\epsilon))$

Reverse Mode: Main Intution

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

Propagates derivatives backward from a given output: achieved by complementing each intermediate variable v_i with an adjoint $\bar{v}_i = \frac{\partial y_j}{v_i}$, representing the sensitivity of output y_j to changes in v_i.

Phase 1: original code run forward, populating intermediate vars v_i

Phase 2: derivatives are calculated by propagating adjoints in reverse, from the outputs to the inputs.

- computes a row of the Jacobian at a time!

If one output y, then start with $\bar{y} = \frac{\partial y}{\partial y} = 1$.
If m outputs, then start a computation for each output with $\bar{y} = e_i,\ i = 1 \ldots m$, where e_i is the i-th unit vector as before!

Reverse Mode: Applied to Basic-Block Code

[Baydin et. al., Automatic Differentiation in Machine Learning: a Survey, 2018]

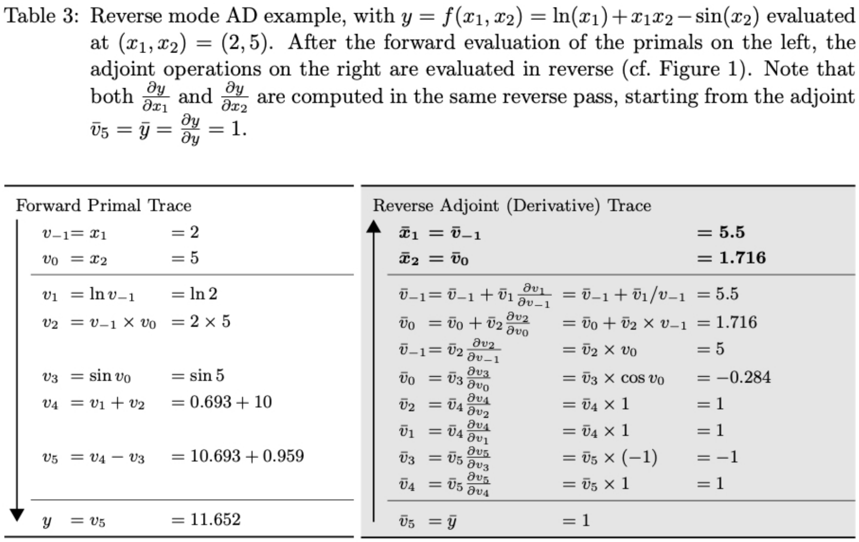


Table 3: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Reverse Adjoint (Derivative) Trace			
$\boldsymbol{\bar{x}_1}$	$\boldsymbol{= \bar{v}_{-1}}$		$\boldsymbol{= 5.5}$
$\boldsymbol{\bar{x}_2}$	$\boldsymbol{= \bar{v}_0}$		$\boldsymbol{= 1.716}$
$\bar{v}_{-1}$	$= \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1}$	$= \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0$	$= \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1$	$= \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4$	$= \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5$	$= \bar{y}$	$= 1$	

Reverse Mode: Key Difficulties

1. tape: is hidden by means of closures, co-rutines, etc., which allows easy code generation, but results in difficult-to-optimize code;
2. low-level code with arbitrary in-place updates is difficult to optimize;

Reverse Mode for a Data-Parallel Language: Key Ideas

1. make the tape part of the translated program rather than a separate abstraction $\Rightarrow$ want to generate "scietific-like" code!

   Key: classic time-space tradeoff based on redundant computation and checkpointing.

2. use the richer semantics of second-order parallel operators to obtain optimized rules

Reverse: parallel constructs will be translated to parallel constructs, but there will be surprises: simplest construct is the most difficult to translate;

- for this presentation we dissalow in-place updates (and scatter).

Time-Space Tradeoff: Redundant Exec + Checkpointing

- Insight from basic-block code: all intermediate variables of the original code are defined in the same scope as the differentiated code, hence directly available;
- Generalization: The codegen of the differentiating code for each new scope:
  - replay the original code of the new scope;
  - recursively codegen and append the differentiating code;
  - "the tape" becomes part of the program and subject to optimizations, such as copy/constant propagation, common-subexpression optimizations, etc.
  - Loops require checkpointing:
    - need to save the (variant) input for each iteration of the loop;
    - then re-execute the original-iteration code and append to it the differentiated-code for an iteration;
    - loop strip-mining implements the space-time tradeoff.
  - Parallel constructs do not require checkpointing, why?
- Performance argument: redundant scalar ops is cheaper than saving results on tape and reading from global memory!
- Compute overhead: $k\times$, where k is the max scopes' depth.

Codegen Demo: Redundant Exec + Checkpointing

Assume the program below, where $f : [m]f32 \to [m]f32$:

```
let b =
  loop (a: [m]f32) for i < n do
    let a' = f a in a'
let r : f32 = g b
in  r

-- original code @level 0 augmented with checkpointing
let as = replicate n (replicate m 0.0)
let (as, b) =
  loop (as: [n][m]f32, a :[m]f32) for i < n do
    let as[i] = a
    let a' = f a
    in  (as, a')
let r : f32 = g b
-- differentiatiated code @level 0
... next slide
```

Codegen Demo: Redundant Exec + Checkpointing

Assume the program below, where $f : [m]f32 \rightarrow [m]f32$:

```
let b =
  loop (a: [m]f32) for i < n do
    let a' = f a in a'
let r : f32 = g b -- assume g inlined
in  r

-- ... continuation from previous slide
-- differentiatiated code @level 0
let r̄ = 1
let b̄ = replicate m 0.0
let b̄ = ḡ r̄ b̄ b -- in-place accumulations to b̄; assume ḡ inlined
let ā = loop (ā:[m]f32) =(b̄: [m]f32) for i = n-1..0 do
    -- redundant computation of original code @level 1
    let a = as[i]
    let a' = f a -- assume f inlined
    -- append differentiating code
    let ā = f̄ ā a a' -- in-place accumulation to ā, assume f̄ inlined
    in  ā
```

- Memory overhead of checkpointing $as : [n][m]f32$;
- Original code redundantly executed twice.

Demo: Time-Space Tradeoff

Assume the program below, where $f : [m]f32 \rightarrow [m]f32$:

```
let b =
  loop (a: [m]f32) for i < n² do
    let a' = f a in a'
let r : f32 = g b
in  r
```

The memory overhead for differentiating would be $O(m \times n^2)$

However if we apply loop stripmining we get:

```
let b =
  loop (a1: [m]f32) = (a) for j < n do
    loop (a2: [m]f32) = (a1) for k < n do
      let i = j*n + k
      let a' = f a2 in a'
let r : f32 = g b in  r
```

Differentiating this code should have:

- memory overhead $2 \times m \times n$ (need to expand a1 and a2);
- redundant-computation factor $3\times$

Morale: a bit of redundant computation reduces the space expansion from polynomial to linear.

Demo: Time-Space Tradeoff

```
-- ... continuation from previous slide
-- differentiatiated code @level 0
let r̄ = 1
let b̄ = replicate m 0.0
let b̄ = ḡ r̄ b̄ b -- in-place accumulations to b̄
let ā = loop (ā1 : [m]f32) = (b̄: [m]f32) for j = n-1..0 do
    -- redundant computation of original code @level 1
    let a1 = a1s[i]
    let (a2s, a') = loop (a2s, a2)=(replicate n (replicate m 0.0), a1)
                      for k < n do
                        let a2s[k] = a2
                        let a' = f a2 in (a2s, a')
    -- differentiating code at level 1
    let ā1 = loop (ā2 : [m]f32) = (ā1: [m]f32) for k = n-1..0 do
                        -- redundant original code @level 2
                        let a2 = a2s[k]
                        let a' = f a2
                        -- differentiating code @level 2
                        let ā2 = f̄ ā2 a2 a'
                        in  ā2
    in ā1
```

Note: `f` is called twice in the diff. code @level 0 + once in the orig.

Differentiating Sum

y = reduce (+) 0 $[a_0, a_1, \ldots, a_{n-1}]$ = $a_0 + a_1 + \ldots + a_{n-1}$

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

Differentiating Sum

y = reduce (+) 0 $[a_0, a_1, \ldots, a_{n-1}]$ = $a_0 + a_1 + \ldots + a_{n-1}$

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

We have the original stmt $y = a_i + k_i$, where $k_i = y - a_i$ is the sum of the other elements. This results in the differentiating (adjoint) code:

$$\bar{a}_i \mathrel{+}= \frac{\partial(a_i + k_i)}{\partial a_i} \cdot \bar{y}$$

Which is simplified to:

$$\bar{a}_i \mathrel{+}= \bar{y}$$

It follows that the adjoint code for summing up an array `as` is:

$\bar{as}$ = map (+ $\bar{y}$) $\bar{as}$

Differentiating Non-Null Product

`y = reduce (·) 0` $[a_0, a_1, \ldots, a_{n-1}]$ `=` $a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1}$

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

Differentiating Non-Null Product

```
y = reduce (·) 0 [$a_0, a_1, \ldots, a_{n-1}$] = $a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1}$
```

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

We have the original stmt $y = a_i \cdot k_i$, where $k_i = y / a_i$ is the product of the other elements, assumed non-zero. This results in the differentiating (adjoint) code:

$$\bar{a}_i {+}= \frac{\partial(a_i \cdot k_i)}{\partial a_i} \cdot \bar{y}$$

Which is simplified to:

$$\bar{a}_i {+}= (y/a_i) \cdot \bar{y}$$

It follows that the adjoint code for taking the product of `as` is:

```
$\bar{as}$ = map2 (+) $\bar{as}$ <| map (\$a_i$ -> (y / $a_i$) * $\bar{y}$) as
```

Differentiating Reduce with Min/Max

??? Think on it.

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \rightarrow \alpha \rightarrow \alpha$ an arbitrary associative operator, e.g., can be 2×2 matrix multiplication. Recall:

$$\texttt{y = reduce} \odot \texttt{e}_{\odot} \; [a_0, a_1, \ldots, a_{n-1}] \; \texttt{=} \; a_0 \odot a_1 \odot \ldots \odot a_{n-1}$$

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator, e.g., can be 2×2 matrix multiplication. Recall:

$$\texttt{y = reduce } \odot \texttt{ e}_\odot \texttt{ [}a_0, a_1, \ldots, a_{n-1}\texttt{]} = a_0 \odot a_1 \odot \ldots \odot a_{n-1}$$

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

$$y = (a_0 \odot \ldots \odot a_{i-1}) \odot a_i \odot (a_{i+1} \odot \ldots \odot a_{n-1})$$

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator, e.g., can be 2×2 matrix multiplication. Recall:

```
y = reduce ⊙ e⊙ [a_0, a_1, ..., a_{n-1}] = a_0 ⊙ a_1 ⊙ ... ⊙ a_{n-1}
```

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

$$y = (a_0 \odot \ldots \odot a_{i-1}) \odot a_i \odot (a_{i+1} \odot \ldots \odot a_{n-1})$$

Computing $\bar{a}_i$ requires to:

(1) compute $\forall i, l_i = a_0 \odot \ldots \odot a_{i-1}$ and $r_i = a_{i+1} \odot \ldots \odot a_{n-1}$

(2) assume the function $f\ l_i\ a_i\ r_i = l_i \odot a_i \odot r_i$;
recursively reverse-differentiate the statement $y = f\ l_i\ a_i\ r_i$,
where l_i and r_i are assumed constants;
the resulting code is denoted by $\bar{a}_i = \bar{f}\ \bar{y}\ l_i\ a_i\ r_i$;

(3) do this for every $a_i, i = 0 \ldots n-1$.

How do we implement steps (1) and (3)?

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator

`y = reduce` $\odot$ `e`$_\odot$ $[a_0, a_1, \ldots, a_{n-1}]$ = $a_0 \odot a_1 \odot \ldots \odot a_{n-1}$

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

$$y = (a_0 \odot \ldots \odot a_{i-1}) \odot a_i \odot (a_{i+1} \odot \ldots \odot a_{n-1})$$

How do we implement step (1)? Assume $as = [a_0, a_1, \ldots, a_{n-1}]$

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \rightarrow \alpha \rightarrow \alpha$ an arbitrary associative operator

```
y = reduce ⊙ e⊙ [a_0, a_1, ..., a_{n-1}] = a_0 ⊙ a_1 ⊙ ... ⊙ a_{n-1}
```

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

$$y = (a_0 \odot \ldots \odot a_{i-1}) \odot a_i \odot (a_{i+1} \odot \ldots \odot a_{n-1})$$

How do we implement step (1)? Assume $as = [a_0, a_1, \ldots, a_{n-1}]$

```
ls = scan^exc ⊙ e⊙ as
rs = reverse <| scan^exc ⊙' e⊙ (reverse as)

where x ⊙' y = y ⊙ x
```

How do we implement step (3)?

Differentiating an Arbitrary Reduce Operation

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator

```
y = reduce ⊙ e⊙ [a_0, a_1, ..., a_{n-1}] = a_0 ⊙ a_1 ⊙ ... ⊙ a_{n-1}
```

So the question is, if we are on the reverse mode and have already computed $\bar{y}$, then what should be the contribution of $\bar{y}$ to some array element a_i?

$$y = (a_0 \odot \ldots \odot a_{i-1}) \odot a_i \odot (a_{i+1} \odot \ldots \odot a_{n-1})$$

How do we implement step (1)? Assume $as = [a_0, a_1, \ldots, a_{n-1}]$

```
ls = scan^exc ⊙ e⊙ as
rs = reverse <| scan^exc ⊙' e⊙ (reverse as)

where x ⊙' y = y ⊙ x
```

How do we implement step (3)?

```
ās = map2 (+̄) ās <| map3 (f̄ ȳ) ls as rs
```

Essentially a reduce is implemented with two scans and a map!

Differentiating Arbitrary Scans

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator
Input array $as = [a_0, a_1, \ldots, a_{n-1}]$

`ys = scan` $\odot$ `e`$_\odot$ `as =` $[a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$

So if we have already computed $\bar{ys}$, then what should be the contribution of $\bar{ys}$ to some array element a_i?

Differentiating Arbitrary Scans

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator
Input array $as = [a_0, a_1, \ldots, a_{n-1}]$

```
ys = scan ⊙ e⊙ as = [a0, a0 ⊙ a1, ..., a0 ⊙ ... ⊙ an−1]
```

$ys = scan \odot e_\odot\ as = [a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}]$

So if we have already computed $\bar{ys}$, then what should be the contribution of $\bar{ys}$ to some array element a_i?

Actually, for reasoning we need to write scan as a loop.

```
let ys[0] = as[0] in
loop (ys) = (replicate n dummy) for i = 1..n−1 do
    let ys[i] = ys[i−1] ⊙ as[i]
    in  ys
```

Differentiating the loop implementation using human knowledge:

Differentiating Arbitrary Scans

$\odot : \alpha \to \alpha \to \alpha$ an arbitrary associative operator
Input array $as = [a_0, a_1, \ldots, a_{n-1}]$

```
ys = scan ⊙ e⊙ as = [$a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}$]
```

So if we have already computed $\bar{ys}$, then what should be the contribution of $\bar{ys}$ to some array element a_i?

Actually, for reasoning we need to write scan as a loop.

```
let ys[0] = as[0] in
loop (ys) = (replicate n dummy) for i = 1..n−1 do
    let ys[i] = ys[i−1] ⊙ as[i]
    in  ys
```

Differentiating the loop implementation using human knowledge:

```
let ($\bar{ys}, \bar{as}$) =
  loop ($\bar{ys}, \bar{as}$) for i = n−1..1 do
    let $\bar{ys}$[i−1] $\bar{+}$= $\frac{\partial ys[i-1] \odot as[i]}{\partial ys[i-1]} \bar{\cdot} \bar{ys}[i]$
    let $\bar{as}$[i] $\bar{+}$= $\frac{\partial ys[i-1] \odot as[i]}{\partial as[i]} \bar{\cdot} \bar{ys}[i]$
    in  ($\bar{ys}$, $\bar{as}$)
let $\bar{as}$[0] $\bar{+}$= $\bar{ys}$[0]
```

Scratch

```
let $(\bar{ys}, \bar{as})$ =
  loop $(\bar{ys}, \bar{as})$ for i = $n-1..1$ do
    let $\bar{ys}[i-1] \mathbin{\bar{+}}= \frac{\partial ys[i-1] \odot as[i]}{\partial ys[i-1]} \bar{\cdot} \bar{ys}[i]$
    let $\bar{as}[i] \mathbin{\bar{+}}= \frac{\partial ys[i-1] \odot as[i]}{\partial as[i]} \bar{\cdot} \bar{ys}[i]$
    in $(\bar{ys}, \bar{as})$
let $\bar{as}[0] \mathbin{\bar{+}}= \bar{ys}[0]$
```

Differentiating Arbitrary Scans

```
ys = scan ⊙ e⊙ as = [$a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}$]

let ($\bar{ys}, \bar{as}$) = -- adjoint code
  loop ($\bar{ys}, \bar{as}$) for i = n-1..1 do
     let $\bar{ys}$[i-1] $\bar{+}$= $\frac{\partial ys[i-1] \odot as[i]}{\partial ys[i-1]} \bar{\cdot} \bar{ys}[i]$
     let $\bar{as}$[i] $\bar{+}$= $\frac{\partial ys[i-1] \odot as[i]}{\partial as[i]} \bar{\cdot} \bar{ys}[i]$
     in ($\bar{ys}$, $\bar{as}$)
let $\bar{as}$[0] $\bar{+}$= $\bar{ys}$[0]
```

It is safe to distributed the outer loop:

```
let ($\bar{ys}$) = loop ($\bar{ys}$) for i = n-1..1 do
     let $\bar{ys}$[i-1] $\bar{+}$= $\frac{\partial ys[i-1] \odot as[i]}{\partial ys[i-1]} \bar{\cdot} \bar{ys}[i]$ in $\bar{ys}$
let ($\bar{as}$) = loop ($\bar{as}$) for i = n-1..0 do -- this is a map!
     let el = if i==0 then $\bar{ys}$[0] else $\frac{\partial ys[i-1] \odot as[i]}{\partial as[i]} \bar{\cdot} \bar{ys}[i]$
     let $\bar{as}$[i] $\bar{+}$= el in $\bar{as}$
```

We have denoted with $\bar{+}$ and $\bar{\cdot}$ the elementwise application of addition and multiplication to elements of tuples or even arrays.

Differentiating Arbitrary Scans

```
ys = scan ⊙ e⊙ as = [$a_0, a_0 \odot a_1, \ldots, a_0 \odot \ldots \odot a_{n-1}$]

let ($\bar{ys}$) = loop ($\bar{ys}$) for i = n−1..1 do
    let $\bar{ys}$[i−1] $\bar{+}$= $\frac{\partial ys[i-1] \odot as[i]}{\partial ys[i-1]} \bar{\cdot} \bar{ys}[i]$ in $\bar{ys}$
let ($\bar{as}$) = loop ($\bar{as}$) for i = n−1..0 do -- this is a map!
    let el = if i==0 then $\bar{ys}$[0] else $\frac{\partial ys[i-1] \odot as[i]}{\partial as[i]} \bar{\cdot} \bar{ys}[i]$
    let $\bar{as}$[i] $\bar{+}$= el in $\bar{as}$
```

Denoting by $c_{n-1} = \bar{1}$ and $c_{i-1} = \frac{\partial ys_{i-1} \odot as_i}{\partial ys_{i-1}}$, the first loop is a linear recurrence of form:

$\bar{ys}_{i-1} = \bar{ys}^0_{i-1} \; + \; c_{i-1} \cdot \bar{ys}_i, \quad i = n-1 \ldots 1, \; and \; \bar{ys}_{n-1} = \bar{ys}^0_{n-1}$

where $\bar{ys}^0$ is the value before the loops is entered.

But a recurrence such as $x_0 = v_0, x_i = a_i + b_i \cdot x_{i-1}$, we know it can be written as a scan with linear-function composition as operator.

Differentiating Arbitrary Scans

Let us summarize:

(1) compute the c_i with a map

(2) perform a scan with linear function composition on the reversed array obtained from zipping the c_is and the $\bar{ys}^0$. then another map operation computes the updated $\bar{ys}$.

(3) a map operation is then necessary to compute the updated $\bar{as}$, i.e., the second loop.

Differentiating Arbitrary Scans

Zoom on step (1): compute the array c (with a map):

$c_{n-1} = \bar{1}$ and $c_i = \frac{\partial ys_i \odot as_{i+1}}{\partial ys_i}$, *for* $i = 0..n-2$

Zoom on step (2): scan with linear function composition

$d_{n-1} = \bar{0},\ d_i = \bar{ys}^0_i,\ i = n-2..0$

```
lin_o (a1,b1) (a2,b2) = (a2 + b2*a1, b1*b2)

$\bar{ys}$ = scan lin_o (0,1) (zip (reverse d) (reverse c))
     |> map (\ $d_i$ $c_i$ -> $d_i$ + $c_i$*$\bar{ys}^0$[n-1])
     |> reverse
```

Zoom on step (3) computing $\bar{as}$

```
$\bar{as}$ =
  map3(\ i $a_i$ $\bar{a}_i$ ->
          let el = if i==0 then $\bar{ys}$[0] else $\frac{\partial ys[i-1]\odot a_i}{\partial a_i} \cdot \bar{ys}[i]$
          in  $\bar{a}_i$ + el
      ) (iota n) as $\bar{as}$
```

Or something like that, maybe test it part of last weekly :-))

Differentiating Map Simple Case

Original program:

```
let xs = map f as
```

If the mapped function has no free variables then it is trivial to differentiate a map; the translation is simply another map:

```
let $\bar{a}s$ =
  map3(\ a $\bar{a}^0$ $\bar{x}$ $->$
        let x = f a
        let $\bar{a}$ = $\bar{f}\,a\,\bar{x}$
        in  $\bar{a}^0$ $\bar{+}$ $\bar{a}$
      ) as $\bar{a}s$ $\bar{x}s$
```

Differentiating Map General Case

The general case requires understanding of imperative analysis of loop parallelism, namely the generalized reduction. Consider a "crazy" example:

```
let ys = map (\ k ind n ->
                loop (x) = (1.0) for i = 2..n do
                  if   p1(k,ind,i) then x * a[ind+i]
                  elif p2(k,ind,i) then x + b[2*ind+i]
                  elif p3(k,ind,i) then x + sin c[3*ind-i]
                  else                  x * d[4*ind-2*i]
             ) keys inds counts
```

Map: writes once per iteration, but has no restrictions on what it reads!

Differentiating Map General Case

The general case requires understanding of imperative analysis of loop parallelism, namely the generalized reduction. Consider a “crazy” example:

```
let ys = map (\ k ind n ->
                 loop (x) = (1.0) for i = 2..n do
                   if   p1(k,ind,i) then x * a[ind+i]
                   elif p2(k,ind,i) then x + b[2*ind+i]
                   elif p3(k,ind,i) then x + sin c[3*ind-i]
                   else                  x * d[4*ind-2*i]
             ) keys inds counts
```

Map: writes once per iteration, but has no restrictions on what it reads!

The adjoint code would have to update each of the elements read by the map. This cannot be represented as a map in general! In the example, we need to update a statically-unknown number of elements from statcically-unknown arrays.

Differentiating Map General Case

The adjoint of the map can be written as a loop:

```
for q = 0..m-1 do
  x = 1; k = keys[q]; ind = inds[q]; n = counts[q];
  xs[n]; -- original code (redundant)
  for i = 2..n do
    xs[i] = x;
    if   p1(k,ind,i) { x = x * a[ind+i]; }
    elif p2(k,ind,i) { x = x + b[2*ind+i]; }
    elif p3(k,ind,i) { x = x + sin(c[3*ind-i]); }
    else             { x = x * d[4*ind-2*i]; }
  -- adjoint code
  $\bar{x}$ = $\bar{ys}$[i]
  for i = n..2 do
    if   p1(k,ind,i) { $\bar{a}$[ind+i] += xs[i] * $\bar{x}$;
                       $\bar{x}$ *= a[ind+i];  }
    elif p2(k,ind,i) { $\bar{b}$[2*ind+i] += $\bar{x}$; }
    elif p3(k,ind,i) { $\bar{c}$[3*ind-i] += cos(c[3*ind-i]) * $\bar{x}$; }
    else             { $\bar{d}$[4*ind-2*i] += xs[i] * $\bar{x}$;
                       $\bar{x}$ *= d[4*ind-2*i];  }
```

This is a generalized reduction, i.e., the outer loop can be executed in parallel if the += updates to arrays $\bar{a}, \bar{b}, \bar{c}, \bar{d}$ are executed attomically.

Generalized Reduction

If all the statements in which a variable `x` appears are of the form `x[index]` $\oplus$`=` *exp*, where `x` does not appear in exp and $\oplus$ is associative and commutative, then the cross-iteration RAWs on `x` can be resolved, for example by executing the update atomically.

A loop nest which whose inter-iterations dependences are all due to such reductions, is called a generalized reduction. Most of the properties that hold on parallel loops also hold on generalized reductions, for example, such a loop can be safely distributed across its statements, and can be interchanged inwards.

The adjoint of a `map` can always be translated to a generalized reduction!

Optimizing Generalized Reductions

One important optimization of generalized reductions is to eliminate the atomic update!

In some cases this can be achieved by performoing loop/map distribution and interchange, that would allow to rewrite the innermost loop as a reduction.

```
for i = 0..n-1
  for j = 0 ..n-1
    for k = 0 .. n-1
      a[i,k] += b[k,j] * c[i,j]
      d[j,k] += b[k,i] * c[i,j]
```

Can be reorganized (on next slide)

Optimizing Generalized Reductions

Can be reorganized such that the index of the atomic update is invariant to the innermost loop. In such case we can re-write it as a reduction:

```
for i = 0..n-1
  for k = 0 .. n-1
    acc = 0.0;
    for j = 0 ..n-1
      acc += b[k,j] * c[i,j]
    a[i,k] += acc;
```

```
map (\ c_row a_row ->
    map (\ b_row a_el ->
          let acc = map2 (*) b_row c_row
                    |> reduce (+) 0
          in  acc + a_el
        ) b a_row
    ) c a
```

```
for j = 0 ..n-1
  for k = 0 .. n-1
    acc = 0.0
    for i = 0 .. n-1
      acc += b[k,i] * c[i,j]
    d[j,k] += acc
```

```
map (\ c_col d_row ->
    map (\ b_row d_el ->
          let acc = map2 (*) b_row c_col
                    |> reduce (+) 0
          in  acc + d_el
        ) b d_row
    ) (transpose c) d
```

This is matrix-multiplication-like code that can be optimized with register and block tiling.