https://www.cl.cam.ac.uk/~nk480/higher-order-ad.pdf

PROVABLY CORRECT, ASYMPTOTICALLY EFFICIENT, HIGHER-ORDER, REVERSE-MODE AUTOMATIC DIFFERENTIATION

ACM
POPL
2022

Faustyna Krawiec | University of Cambridge
Simon Peyton Jones | Epic Games/University of Cambridge*
Neel Krishnaswami | University of Cambridge
Tom Ellis | Groq*
Richard Eisenberg | Tweag
Andrew Fitzgibbon | Graphcore*

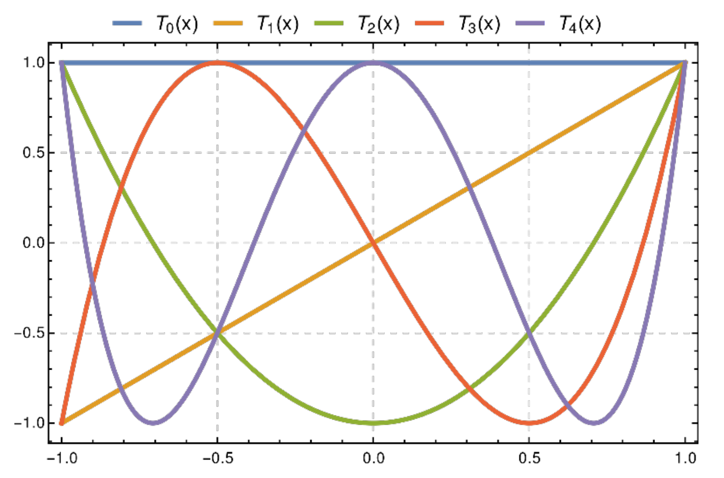
*Work done while at Microsoft Research

Why are we here?

- Before deep learning

$$f(x) \approx \sum_i w_i \phi_i(x)$$

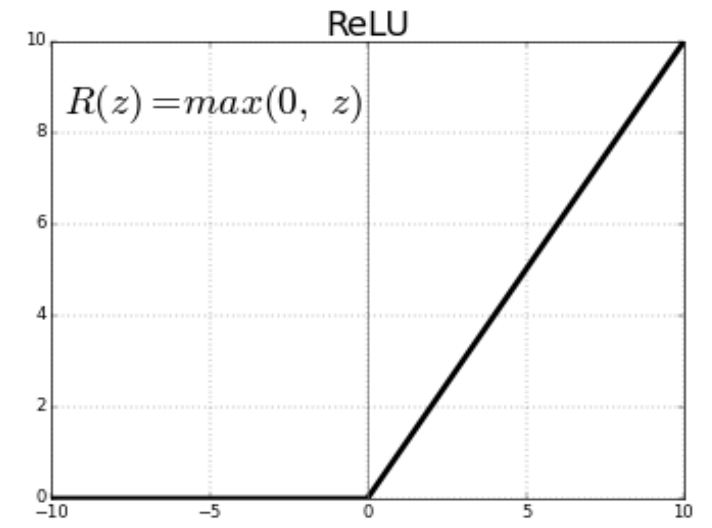
smooth functions, closed-form solutions, grist to analysis



- After deep learning

$$f(x) \approx f_L \circ f_{L-1} \circ \ldots \circ f_1(x) f_i(a) = \sum_j w_{ij} \max(a_j,\ 0)$$

deep recursion, nonsmooth, grist to gradient descent



Automatic differentiation

Given: computer code for

$$f \;::\; \mathbb{R}^N \to \mathbb{R}^M$$

Produce: code that computes the Jacobian

$$J_f :: \mathbb{R}^N \to \mathbb{R}^{M \times N}$$

Or variations thereof:

$f' :: \left(\mathbb{R}^N,\ \mathbb{R}^N\right) \to \mathbb{R}^M$ $\quad f'(x, dx) = J_f \cdot dx$ $\quad$ Forward

$f^{\backprime} :: \left(\mathbb{R}^N, \mathbb{R}^M\right) \to \mathbb{R}^N$ $\quad f^{\backprime}\left(x, df\right) = J_f^{\top} \cdot df$ $\quad$ Reverse

And for the important case of $M = 1$

$\nabla f :: \mathbb{R}^N \to \mathbb{R}^N$ $\quad \nabla f(x) = f^{\backprime}(x, 1)$ $\quad$ Gradient

Automatic differentiation (arbitrary datatypes)

Given: computer code for

$$f \;::\; S \to T$$

S and *T* essentially containers of “reals + other stuff”

Produce: code that computes the Jacobian

$$J_f :: S \to \left(\mathrm{d}S \multimap \mathrm{d}T\right)$$

$\mathrm{d}S$ the tangent space of S

$A \multimap B$ is the type of linear maps from A to B

Equipped with “apply” $\odot$ and “transpose” $\cdot^{\top}$

Or variations thereof:

$$f' :: \left(S,\ \mathrm{d}S\right) \to \mathrm{d}T \qquad f'(x, dx) = J_f \odot dx$$

Forward

$$f^{\grave{}} :: \left(S, \mathrm{d}T\right) \to \mathrm{d}S \qquad f^{\grave{}}\left(x, df\right) = J_f^{\top} \odot df$$

Reverse

And for the important case of $M = 1$, the **gradient**

$$\nabla f :: S \to \mathrm{d}S \qquad \nabla f(x) = f^{\grave{}}(x, 1)$$

Gradient

Automatic differentiation

Given: computer code for

$$f \ :: \mathbb{R}^N \to \mathbb{R}^M$$

Produce: code that computes the Jacobian

$$J_f :: \mathbb{R}^N \to \mathbb{R}^{M \times N}$$

Or variations thereof:

$f' :: \left(\mathbb{R}^N,\ \mathbb{R}^N\right) \to \mathbb{R}^M$	$f'(x, dx) = J_f \cdot dx$	Forward
$f^{`} :: \left(\mathbb{R}^N, \mathbb{R}^M\right) \to \mathbb{R}^N$	$f^{`}\left(x, df\right) = J_f^\top \cdot df$	Reverse

And for the important case of $M = 1$

$\nabla f :: \mathbb{R}^N \to \mathbb{R}^N$	$\nabla f(x) = f^{`}(x, 1)$	Gradient

Automatic differentiation

Given: computer code for

$$f \ :: \mathbb{R}^N \to \mathbb{R}^M$$

Produce: code that computes the Jacobian

$$J_f :: \mathbb{R}^N \to \mathbb{R}^{M \times N}$$

Or variations thereof:

$$f' :: \left(\mathbb{R}^N,\ \mathrm{d}\mathbb{R}^N\right) \to \mathrm{d}\mathbb{R}^M \qquad f'(x, dx) = J_f \cdot dx$$

Forward

$$f^{\grave{}} :: \left(\mathbb{R}^N, \mathrm{d}\mathbb{R}^M\right) \to \mathrm{d}\mathbb{R}^N \qquad f^{\grave{}}\left(x, df\right) = J_f^\top \cdot df$$

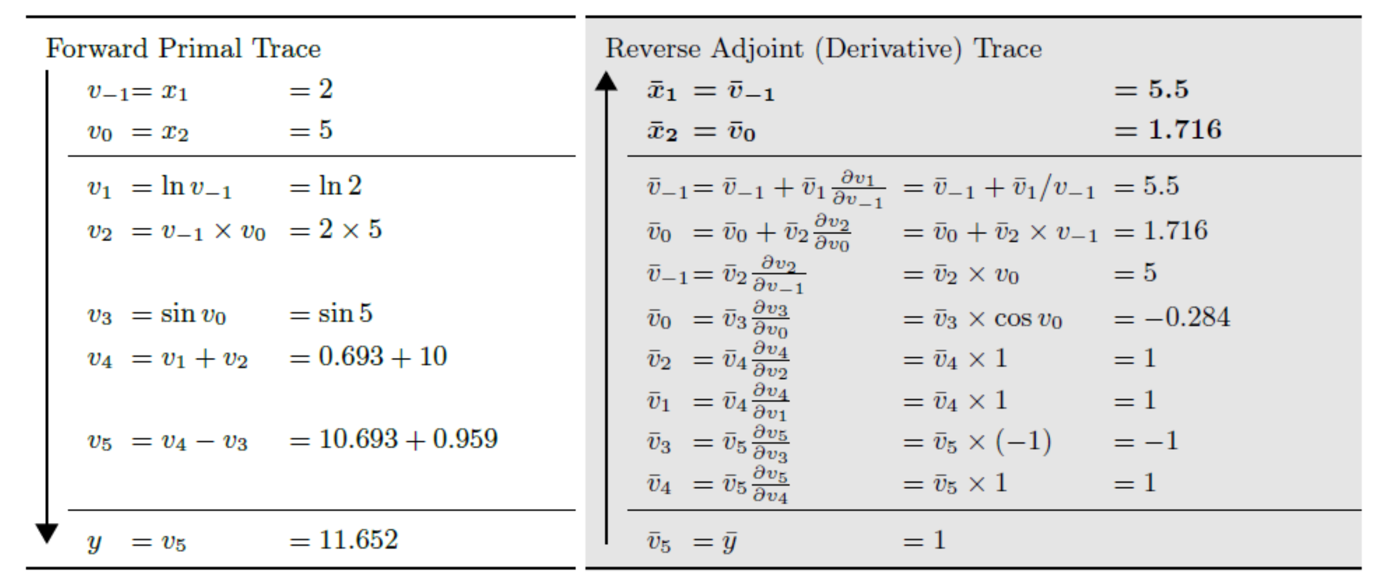
Reverse

And for the important case of $M = 1$

$$\nabla f :: \mathbb{R}^N \to \mathrm{d}\mathbb{R}^N \qquad \nabla f(x) = f^{\grave{}}(x, 1)$$

Gradient

Problem



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

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

The (huge, hot)
AD literature

- Has many well-written papers and remarkable implementations
- Implementations tend to be operational and stateful: graph construction and mutation, “tapes”, primal traces (aka Wengert lists), derivative traces, perturbation confusion, call/cc, etc.   Accompanying papers (when they exist) tend to be dominated by examples
- Principled theory papers tend to lack implementations, and/or are asymptotically slow
- Is often indirect: you write a program that constructs a graph, that is then run/differentiated

Goals

- Simple, principled
- Works for reverse derivatives
- Works for higher order programs
- Works for functions of type $S \to T$, not just $\mathbb{R}^a \to \mathbb{R}^b$
- Asymptotically fast
- Provably correct

Dream AD

- Source-to-source
- Optimally efficient for small and large programs
- Supports all language features

```
def f(x: float) -> float:
  y = x * sin x
  return y + x*10

def f′(x: float) -> float:
  (s,c) = sincos x
  return x * c + s + 10
```

Dual numbers

The simplest way to do AD

Not very good for reverse AD

But even if you know this, watch keenly...

Forward AD using dual numbers

- Make Float a pair $(\mathbb{R},\ \mathrm{d}\mathbb{R})$
- Define operations using simple chain rule

$$(a, da)\ +\ (b, db)\ =\ (a\ +\ b,\ da\ +\ db)$$

$$(a, da) * (b, db) = (a * b,\ a * db + da * b)$$

$$\sin(a, da)\ =\ \big(\sin(a),\ \ da\ *\ \cos(a)\big)$$

$$\mathrm{atan2}((a, da), (b, db)) = \left(\mathrm{atan2}(a, b), \frac{b * da - a * db}{a^2 + b^2}\right)$$

- And discard the first part of the result

```
class Float:
  primal: float
  tangent: float

def sin (a: Float) -> Float:
  return Float (
       sin(a.primal),
       a.tangent*cos(a.primal)
  )

def f(x: Float) -> Float:
  y = x * sin x
  return y + x*10

def f’(x: float) -> float:
  xdual = (x, 1.0)
  return snd(f(xdual))
```

Forward AD using dual numbers

- Functions $f :: \mathbb{R}^N \rightarrow \mathbb{R}^M$
- Become $f' :: \left(\mathbb{R}, \mathrm{d}\mathbb{R}\right)^N \rightarrow \left(\mathbb{R}, \mathrm{d}\mathbb{R}\right)^M$
- Easily transformed to

$$f' :: \left(\mathbb{R}^N, \mathrm{d}\mathbb{R}^N\right) \rightarrow \mathrm{d}\mathbb{R}^M$$

- A reasonable way to implement forward derivative, jvp, …
- … but notoriously poor for reverse derivatives, and, most importantly, for **gradients**

```
class Float:
  primal: float
  tangent: float

def sin (a: Float) -> Float:
  return Float (
        sin(a.primal),
        a.tangent*cos(a.primal)
  )

def f(x: Float) -> Float:
  y = x * sin x
  return y + x*10

def f’(x: float) -> float:
  xdual = (x, 1.0)
  return f(xdual)₂
```

(e)$_2$ means second element of tuple (e), (e)[1] in Python

Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \begin{bmatrix} f'\left(x,\ [1, \ldots, 0]\right) \\ \vdots \\ f'\left(x,\ [0, \ldots, 1]\right) \end{bmatrix}$$

- Hurts if $N = 10^9$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def ∇f(x : float[3]) -> float[3]:
  return [
    f([(x[0], 1), (x[1], 0), x[2], 0)])_2
    f([(x[0], 0), (x[1], 1), x[2], 0)])_2
    f([(x[0], 0), (x[1], 0), x[2], 1)])_2
  ]
```

$(e)_2$ means second element of tuple (e)

And yes… some day we may learn to use fewer probes [Baydin et al], but for now we need gradients

[Aside: can be OK for Jacobians]

- Essentially, forward or reverse derivatives compute a column/row of J.

If you want all of J and

- it’s strongly portrait or landscape,
- not sparse

you’re probably fine.

Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \begin{bmatrix} f'\left(x,\ [1,\ldots,0]\right) \\ \vdots \\ f'\left(x,\ [0,\ldots,1]\right) \end{bmatrix}$$

- Hurts if $N = 10^8$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def ∇f(x : float[3]) -> float[3]:
  return [
    f([(x[0], 1), (x[1], 0), x[2], 0)])_2
    f([(x[0], 0), (x[1], 1), x[2], 0)])_2
    f([(x[0], 0), (x[1], 0), x[2], 1)])_2
  ]
```

And yes… we may learn to use fewer probes
[Baydin et al]

Fix #1: tupling

Forward AD using dual numbers

- Make Float a pair $(\mathbb{R},\ \mathrm{d}\mathbb{R})$

- Define operations using simple chain rule

$$(a, da)\ +\ (b, db)\ =\ (a\ +\ b,\ da\ +\ db)$$

$$(a, da) * (b, db) = (a * b,\ a * db + da * b)$$

$$\sin(a, da)\ =\ \big(\sin(a),\ da\ *\ \cos(a)\big)$$

$$\operatorname{atan2}((a, da), (b, db)) = \left(\operatorname{atan2}(a, b), \frac{b * da - a * db}{a^2 + b^2}\right)$$

- And discard the first part of the result

```
class Float:
  primal: float
  tangent: float

def sin (a: Float) -> Float:
  return Float (
       sin(a.primal),
       a.tangent*cos(a.primal)
  )

def f(x: Float) -> Float:
  y = x * sin x
  return y + x*10

def f′(x: float) -> float:
  xdual = (x, 1.0)
  return snd(f(xdual))
```

Forward AD using dual numbers

- Make Float a pair $(\mathbb{R},\ \mathrm{d}\mathbb{R})$
- Define operations using simple chain rule

$+\!: da + db$

$*: a * db + da * b$

$\sin\!: da * \cos(a)$

$\mathrm{atan2}\!: \frac{b}{a^2 + b^2} * da - \frac{a}{a^2 + b^2} * db$

```
class Float:
  primal: float
  tangent: float

def sin (a: Float) -> Float:
  return Float (
        sin(a.primal),
        a.tangent*cos(a.primal)
  )
```

Forward AD using DualVectors

- Make Float a pair $(\mathbb{R},\ \mathrm{d}S)$
- Define operations using simple chain rule

$$+ : \mathrm{dAdd}(da, db)$$

$$* : \mathrm{dAdd}\big(\mathrm{dScale}(a, db), \mathrm{dScale}(b, da)\big)$$

$$\sin : \mathrm{dScale}\big(\cos(a), da\big)$$

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec

class Float:
  primal: float
  tangent: DualVec

def sin (a: Float) -> Float:
  return Float (
       sin(a.primal),
       scale(cos(a.primal), a.tangent)
  )
```

$$\mathrm{atan2} : \mathrm{dAdd}\left(\mathrm{dScale}\left(\frac{b}{a^2+b^2},\ da\right),\ \mathrm{dScale}\left(-\frac{a}{a^2+b^2}, db\right)\right)$$

And now the full gradient in one call…

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2\left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix}\right)$$

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \begin{bmatrix} f'\left(x,\ [1,\ldots,0]\right) \\ \vdots \\ f'\left(x,\ [0,\ldots,1]\right) \end{bmatrix}$$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def ∇f(x : float[3]) -> float[3]:
  return [
    f([(x[0], 1), (x[1], 0), x[2], 0)])_2
    f([(x[0], 0), (x[1], 1), x[2], 0)])_2
    f([(x[0], 0), (x[1], 0), x[2], 1)])_2  ]
```

$$f_2(x) = f(x)[1] = \mathrm{snd}\left(f(x)\right)$$

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2\left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix}\right)$$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def ∇f(x : float[3]) -> float[3]:
  return [
    f([(x[0], 1), (x[1], 0), x[2], 0)])_2
    f([(x[0], 0), (x[1], 1), x[2], 0)])_2
    f([(x[0], 0), (x[1], 0), x[2], 1)])_2  ]
```

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \left[f_2\left(\begin{bmatrix} x_0, 1 \\ \vdots \\ x_N, 0 \end{bmatrix}\right), \ldots, f_2\left(\begin{bmatrix} x_0, 0 \\ \vdots \\ x_N, \; 1 \end{bmatrix}\right) \right]$$

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2\left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix}\right)$$

- In order to get $\nabla f(x)$, make 1 call:

$$\nabla f(x) = f_2\left(\begin{bmatrix} x_0, [1, \ldots, 0] \\ \vdots \\ x_N, [0, \ldots, 1] \end{bmatrix}\right)$$

- Hooray! $1 \ll N$! But still obviously $O\left(N^2\right)$...

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def ∇f(x : float[3]) -> float[3]:
  return
             [1]         [0]         [0]
    f((x[0], [0]), (x[1], [1]), (x[2], [0])])2
             [0]         [0]         [1]
```

[Not real Python syntax ☺]

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2\left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix}\right)$$

- In order to get $\nabla f(x)$, make 1 call:

$$\nabla f(x) = f_2\left(\begin{bmatrix} x_0, [1, \ldots, 0] \\ \vdots \\ x_N, [0, \ldots, 1] \end{bmatrix}\right)$$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def                            ]:

                                  [0]
                          [2],    [0]])])2
                                  [1]
```

So use sparse matrices… problem solved?

- Hooray! $1 \ll N$! But still obviously $O\left(N^2\right)$…

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2\left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix}\right)$$

- In order to get $\nabla f(x)$, make 1 call:

$$\nabla f(x) = f_2\left(\begin{bmatrix} x_0, \text{sparseOneHot}(1,N) \\ \vdots \\ x_N, \text{sparseOneHot}(N, N) \end{bmatrix}\right)$$

- Hooray! $1 \ll N$! But still obviously $O\left(N^2\right)$...

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def                          ]:
                                    [0]
                          [2],  [0])])₂
                                    [1]
```

So use sparse matrices... problem solved?

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

Not quite... they start sparse, but end up full*.

But... bear with us while we introduce a funny kind of sparse vector...

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def                                ]:

                                           [0]
                               [2],        [0])])₂
                                           [1]
```

So use sparse matrices... problem solved?

- In

$$\nabla f(x) = f_2\left(\begin{bmatrix} x_0, \text{sparseOneHot}(1,N) \\ \vdots \\ x_N, \text{sparseOneHot}(N, N) \end{bmatrix}\right)$$

- Hooray! $1 \ll N$! But still obviously $O\left(N^2\right)$...

* and if there's one thing worse than a $10^8 \times 10^8$ dense matrix, it's a $10^8 \times 10^8$ sparse matrix which is mostly full.

Fix #2: Symbolic sparse vectors

Record constructors, rather than actually constructing

Forward AD using DualVectors

- Make Float a pair $(\mathbb{R},\ \mathrm{d}S)$
- Define operations using simple chain rule

$$+ : \mathrm{dAdd}(da, db)$$

$$* : \mathrm{dAdd}\big(\mathrm{dScale}(a, db), \mathrm{dScale}(b, da)\big)$$

$$\sin : \mathrm{dScale}\big(\cos(a), da\big)$$

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec
  dDot: (DualVec, DualVec) -> float

class Float:
  primal: float
  tangent: DualVec

def sin (a: Float) -> Float:
  return Float (
        sin(a.primal),
        scale(cos(a.primal), a.tangent)
  )
```

$$\mathrm{atan2} : \mathrm{dAdd}\left(\mathrm{dScale}\left(\frac{b}{a^2 + b^2},\ da\right),\ \mathrm{dScale}\left(-\frac{a}{a^2 + b^2}, db\right)\right)$$

Forward AD using DualVectors

- Make Float a pair $(\mathbb{R},\ \mathrm{d}S)$
- Define operations using simple chain rule

$$+: \mathrm{dAdd}(da, db)$$

$$*: \mathrm{dAdd}\big(\mathrm{dScale}(a, db), \mathrm{dScale}(b, da)\big)$$

$$\sin: \mathrm{dScale}\big(\cos(a), da\big)$$

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec
  dOneHot: (i:int, n:int) -> DualVec
  dDot: (DualVec, DualVec) -> float

class Float:
  primal: float
  tangent: DualVec

def sin (a: Float) -> Float:
  return Float (
         sin(a.primal),
         scale(cos(a.primal), a.tangent)
  )
```

$$\mathrm{atan2}: \mathrm{dAdd}\left(\mathrm{dScale}\left(\frac{b}{a^2+b^2},\ da\right),\ \mathrm{dScale}\left(-\frac{a}{a^2+b^2}, db\right)\right)$$

Lazy DualVecs

- In forward mode, start with dx vector in $f'(x, dx)$ and DualVec interfaces just operate directly
- In Lazy mode, DualVec instances are just constructors, so after running the program, we get not a vector but a tree dv
- Need a function $eval(dv)$ to turn it into a dense Vec
- All ops constant time

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec
  dOneHot: (i:int, n:int) -> DualVec
  dDot: (DualVec, DualVec) -> float

class Scale(DualVec):
  s: float
  dv: DualVec

class Add(DualVec):
  da: DualVec
  db: DualVec

class OneHot(DualVec):
  i:int
  n:int
```

- This is just a symbolic sparseVector representation.
- How might you represent a sparse vector?

  Type SparseVec = List[int,float]

- `SparseVec([2,1.1],[17,2.2]) == Add(Scale(1.1,OneHot(2)), Scale(2.2, OneHot(17))`

```
interface DualVec:                                 Storage
  dScale: (float, DualVec) -> DualVec              4 bytes
  dAdd: (DualVec, DualVec) -> DualVec              0 bytes
  dZero: DualVec                                   0 bytes
  dOneHot: (i:int, n:int) -> DualVec               8 bytes
  dDot: (DualVec, DualVec) -> float
```

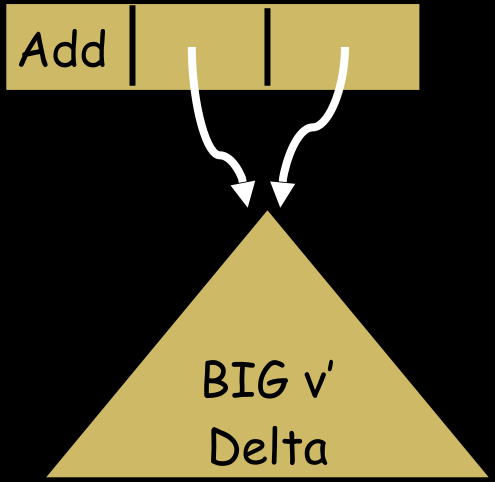
- Essentially equally efficient
- But now, get lazy…

Fix #3: sharing

Job done? Not so fast...

```
def f(x):
  y = <very-big-expr>  # Returns a big LazyDV
  return y + y
```

- We will call f @(Dual Delta)
- So y::Dual Delta = $D\ v\ v'$ where v': $Delta$ may be VERY BIG
- Then (y+y) = $D\ (v+v)\ (Add\ v'v')$
  NB: v' is shared
- BUT **ALAS** eval can't see the sharing
  and so will traverse BIG Delta twice



Catastrophic loss of sharing

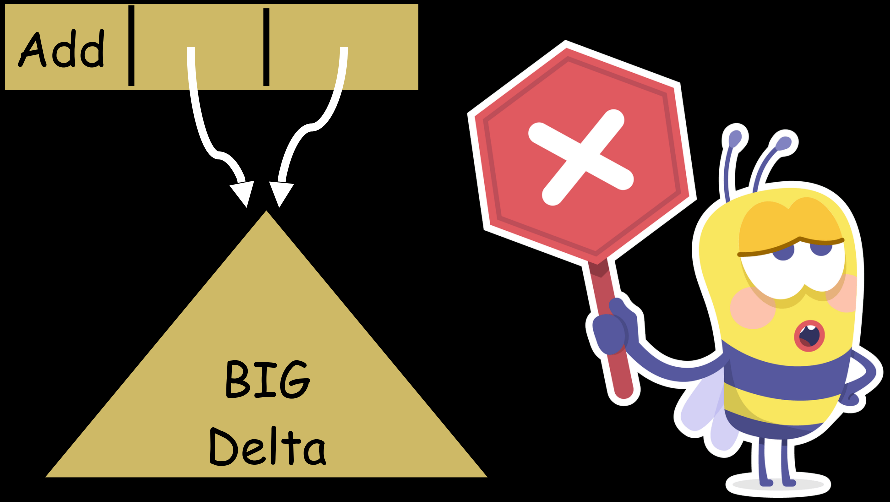
- BUT ALAS eval can’t see the sharing and so will traverse BIG Delta twice
- This can be *asymptotically* bad

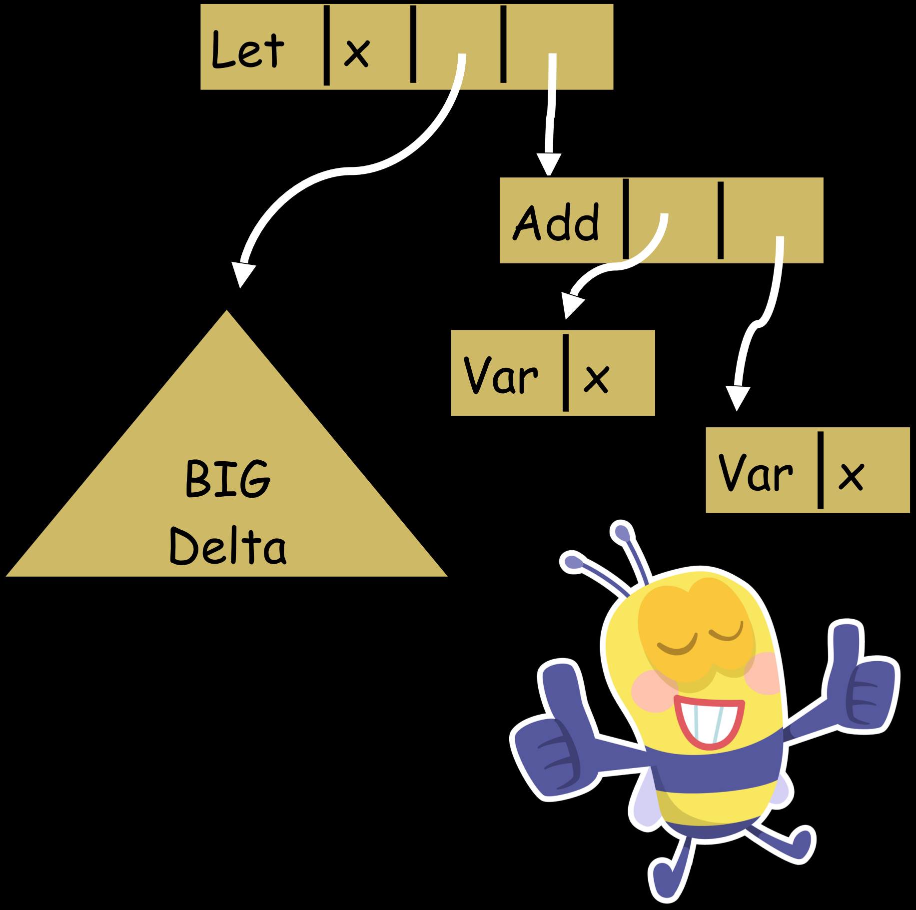
```
f :: Num a => Vec a -> a
f x = let y1 = <rhs>
          y2 = y1+y1
          y3 = y2+y2
          …
          y100 = y99+y99
      in y100
```

- A linear sized graph unravels to an *exponentially* larger tree

The fix: make sharing explicit

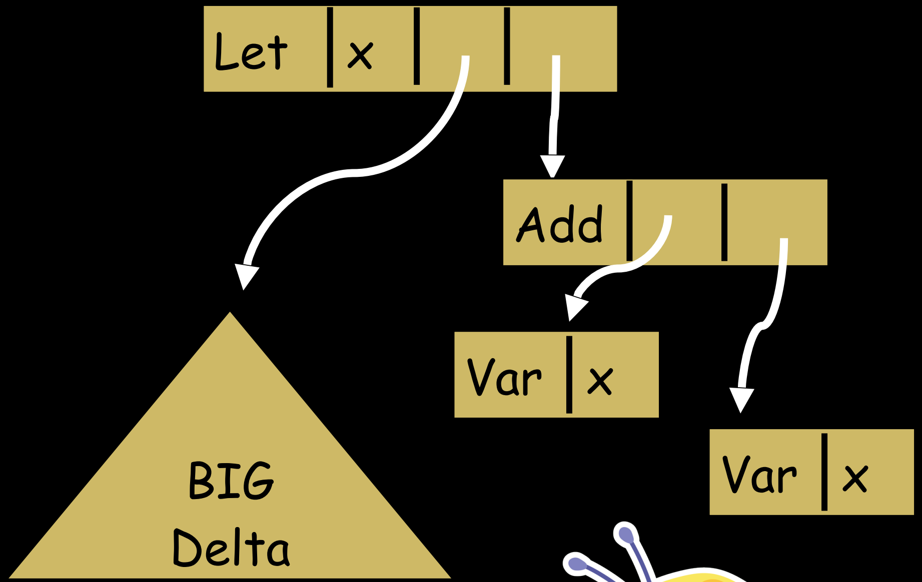
```
data Delta = Zero
           | OneHot Int
           | Scale Float Delta
           | Add Delta Delta
           | Var DeltaId
           | Let DeltaId Delta Delta
type DeltaId = Int
eval :: (Int,Int) -> Delta -> Vec Float'
```





The fix: make sharing explicit

```
data Delta = Zero
           | OneHot Int
           | Scale Float Delta
           | Add Delta Delta
           | Var DeltaId
           | Let DeltaId Delta Delta
type DeltaId = Int
eval :: (Int,Int) -> Delta -> Vec Float'
```

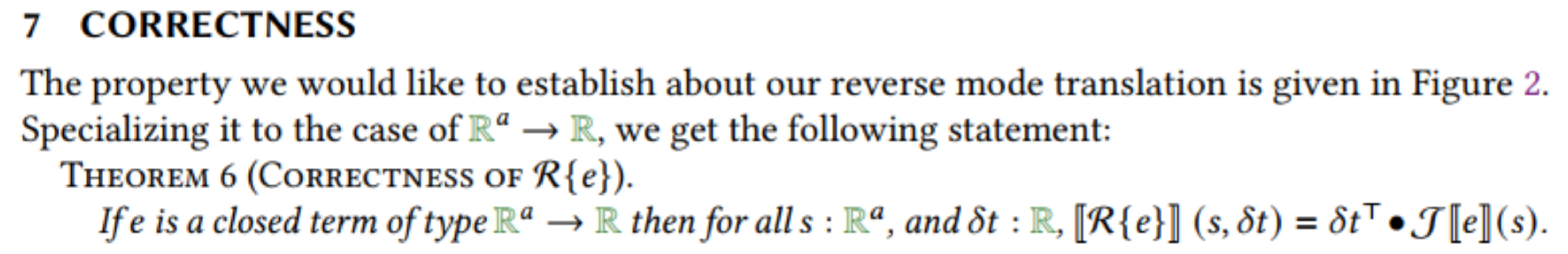


- And eval is now efficient
- Evaluates the RHS of a let just once!
- Can still use update-in-place
- Side note: evaluate Let “backwards” [see paper]

Does it work?

…i.e. does it correctly compute derivatives?

Yes: we have a proof (see the paper)



7 CORRECTNESS

The property we would like to establish about our reverse mode translation is given in Figure 2. Specializing it to the case of $\mathbb{R}^a \to \mathbb{R}$, we get the following statement:

Theorem 6 (Correctness of $\mathcal{R}\{e\}$).

If e is a closed term of type $\mathbb{R}^a \to \mathbb{R}$ then for all $s : \mathbb{R}^a$, and $\delta t : \mathbb{R}$, $[\![\mathcal{R}\{e\}]\!]\ (s, \delta t) = \delta t^\top \bullet \mathcal{J}[\![e]\!](s)$.

Does it work fast?

- Every Float turns into a (D Float Delta)
- Each primop allocates a constant amount of Delta stuff – and nothing else does.
- eval runs in O(size Delta)

Bottom line: runtime is only a constant factor worse than the original program – and that too is easy to prove

```
module Program (R : Real) = struct
  open R    -- access real from the module R
            -- along with arithmetic operations

  type vec3 = {x : real; y : real; z : real}

  type quaternion =
    {x : real; y : real; z : real; w : real}

  let q_to_vec (q : quaternion) : vec3 =
    {x = q.x; y = q.y; z = q.z}

  let dot (p : vec3) (q : vec3) : real =
    p.x × q.x + p.y × q.y + p.z × q.z

      -- Vector addition
  let (++) (p : vec3) (q : vec3) : vec3 =
    {x = p.x + q.x; y = p.y + q.y; z = p.z + q.z}

  let scale k (v : vec3) : vec3 =
    {x = k × v.x; y = k × v.y; z = k × v.z}

  let cross (a : vec3) (b : vec3) : vec3 =
    {x = a.y × b.z − a.z × b.y;
     y = a.z × b.x − a.x × b.z;
     z = a.x × b.y − a.y × b.x}

  let norm (x : vec3) : real = sqrt (dot x x)

  let rotate_vec_by_quat (v : vec3)
                         (q : quaternion) : vec3 =
    let u = q_to_vec q in
    let s = q.w in
    scale (from_float 2.0 × dot u v) u
    ++
    scale (s × s − dot u u) v
    ++
    scale (from_float 2.0 × s) (cross u v)
end
```

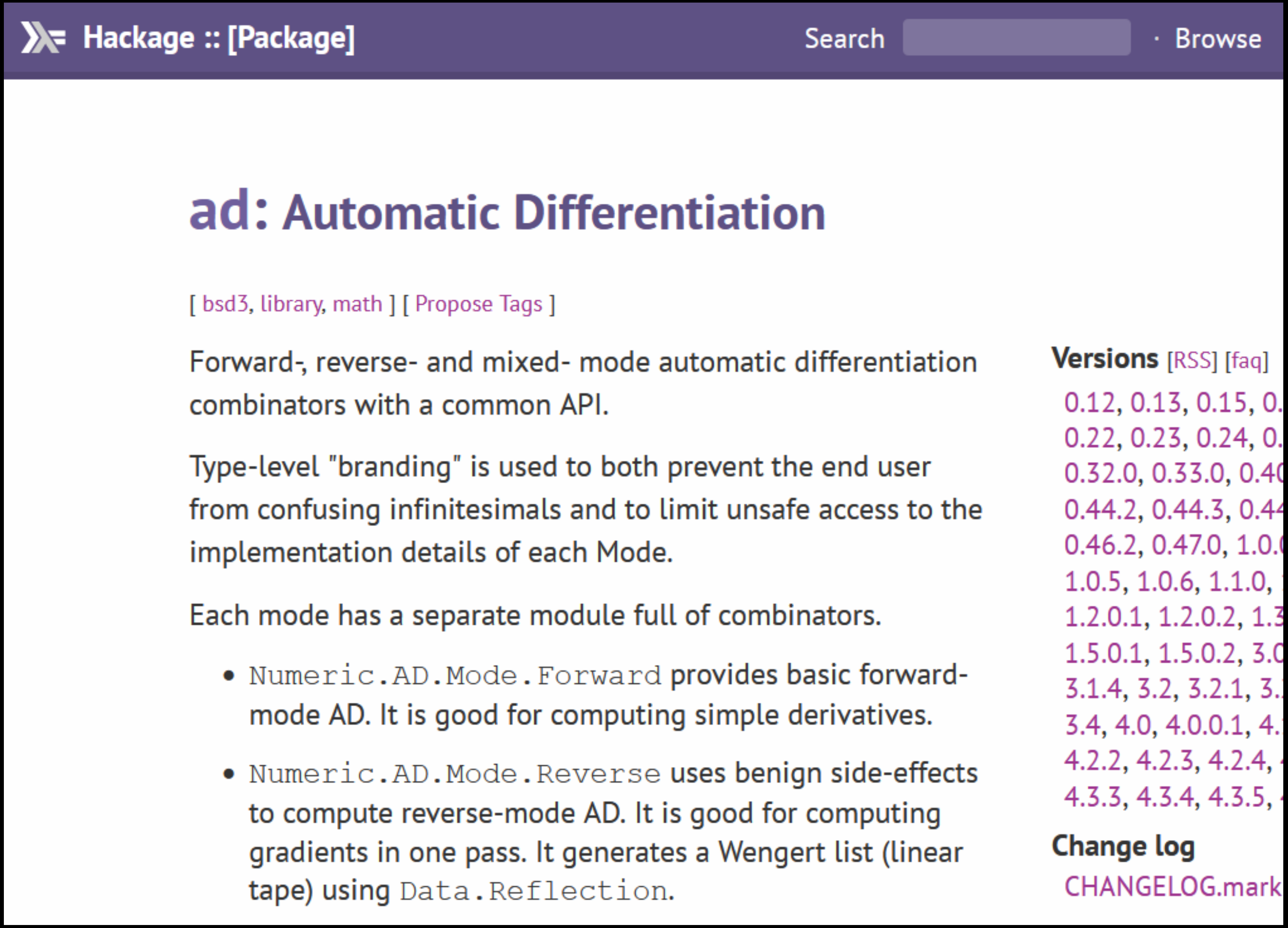
```
  -- Result of runDelta 8 (𝒟⃖{e} s0), where
  -- e = λq v. (rotate_vec_by_quat v q).x
  -- s0 = (q0; v0)
  -- q0 = {(1.1; Var qx); (2.2; Var qy); (3.3; Var qz); (4.4; Var qw)}
  -- v0 = {(5.5; Var vx); (6.6; Var vy); (7.7; Var vz)}

  -- We informally use a Let/in notation for the constructor Let, and
  -- use variable names instead of numbers, so xN stands for variable N+8

Let x1  = Add (Scale 5.5 (Var qy)) (Scale 2.2 (Var vx)) in
Let x2  = Add (Scale 6.6 (Var qx)) (Scale 1.1 (Var vy)) in
Let x3  = Add (Var x2) (Scale (−1.0) (Var x1)) in
Let x4  = Add (Scale 7.7 (Var qx)) (Scale 1.1 (Var vz)) in
Let x5  = Add (Scale 5.5 (Var qz)) (Scale 3.3 (Var vx)) in
Let x6  = Add (Var x5) (Scale (−1.0) (Var x4)) in
Let x7  = Add (Scale 6.6 (Var qz)) (Scale 3.3 (Var vy)) in
Let x8  = Add (Scale 7.7 (Var qy)) (Scale 2.2 (Var vz)) in
Let x9  = Add (Var x8) (Scale (−1.0) (Var x7)) in
Let x10 = Zero in
Let x11 = Add (Scale 4.4 (Var x10)) (Scale 2 (Var qw)) in
Let x12 = Add (Scale (−4.84) (Var x11)) (Scale 8.8 (Var x3)) in
Let x13 = Add (Scale 9.68 (Var x11)) (Scale 8.8 (Var x6)) in
Let x14 = Add (Scale (−4.84) (Var x11)) (Scale 8.8 (Var x9)) in
Let x15 = Add (Scale 3.3 (Var qz)) (Scale 3.3 (Var qz)) in
Let x16 = Add (Scale 2.2 (Var qy)) (Scale 2.2 (Var qy)) in
Let x17 = Add (Scale 1.1 (Var qx)) (Scale 1.1 (Var qx)) in
Let x18 = Add (Var x17) (Var x16) in
Let x19 = Add (Var x18) (Var x15) in
Let x20 = Add (Scale 4.4 (Var qw)) (Scale 4.4 (Var qw)) in
Let x21 = Add (Var x20) (Scale (−1.0) (Var x19)) in
Let x22 = Add (Scale 7.7 (Var x21)) (Scale 2.42 (Var vz)) in
Let x23 = Add (Scale 6.6 (Var x21)) (Scale 2.42 (Var vy)) in
Let x24 = Add (Scale 5.5 (Var x21)) (Scale 2.42 (Var vx)) in
Let x25 = Add (Scale 7.7 (Var qz)) (Scale 3.3 (Var vz)) in
Let x26 = Add (Scale 6.6 (Var qy)) (Scale 2.2 (Var vy)) in
Let x27 = Add (Scale 5.5 (Var qx)) (Scale 1.1 (Var vx)) in
Let x28 = Add (Var x27) (Var x26) in
Let x29 = Add (Var x28) (Var x25) in
Let x30 = Zero in
Let x31 = Add (Scale 45.98 (Var x30)) (Scale 2 (Var x29)) in
Let x32 = Add (Scale 3.3 (Var x31)) (Scale 91.96 (Var qz)) in
Let x33 = Add (Scale 2.2 (Var x31)) (Scale 91.96 (Var qy)) in
Let x34 = Add (Scale 1.1 (Var x31)) (Scale 91.96 (Var qx)) in
Let x35 = Add (Var x32) (Var x22) in
Let x36 = Add (Var x33) (Var x23) in
Let x37 = Add (Var x34) (Var x24) in
Let x38 = Add (Var x35) (Var x12) in
Let x39 = Add (Var x36) (Var x13) in
Let x40 = Add (Var x37) (Var x14) in
Var x40
```

This is the Delta we get

Is it “new”? No...



ad: Automatic Differentiation

[bsd3, library, math] [Propose Tags]

Forward-, reverse- and mixed- mode automatic differentiation combinators with a common API.

Type-level "branding" is used to both prevent the end user from confusing infinitesimals and to limit unsafe access to the implementation details of each Mode.

Each mode has a separate module full of combinators.

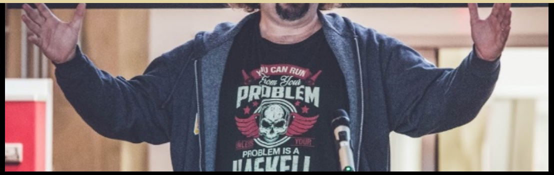
- `Numeric.AD.Mode.Forward` provides basic forward-mode AD. It is good for computing simple derivatives.
- `Numeric.AD.Mode.Reverse` uses benign side-effects to compute reverse-mode AD. It is good for computing gradients in one pass. It generates a Wengert list (linear tape) using `Data.Reflection`.

Versions [RSS] [faq]

0.12, 0.13, 0.15, 0.
0.22, 0.23, 0.24, 0.
0.32.0, 0.33.0, 0.4
0.44.2, 0.44.3, 0.4
0.46.2, 0.47.0, 1.0.
1.0.5, 1.0.6, 1.1.0,
1.2.0.1, 1.2.0.2, 1.3
1.5.0.1, 1.5.0.2, 3.0
3.1.4, 3.2, 3.2.1, 3.
3.4, 4.0, 4.0.0.1, 4.
4.2.2, 4.2.3, 4.2.4,
4.3.3, 4.3.4, 4.3.5,

Change log

CHANGELOG.mark



Edward Kmett

Barak Pearlmutter

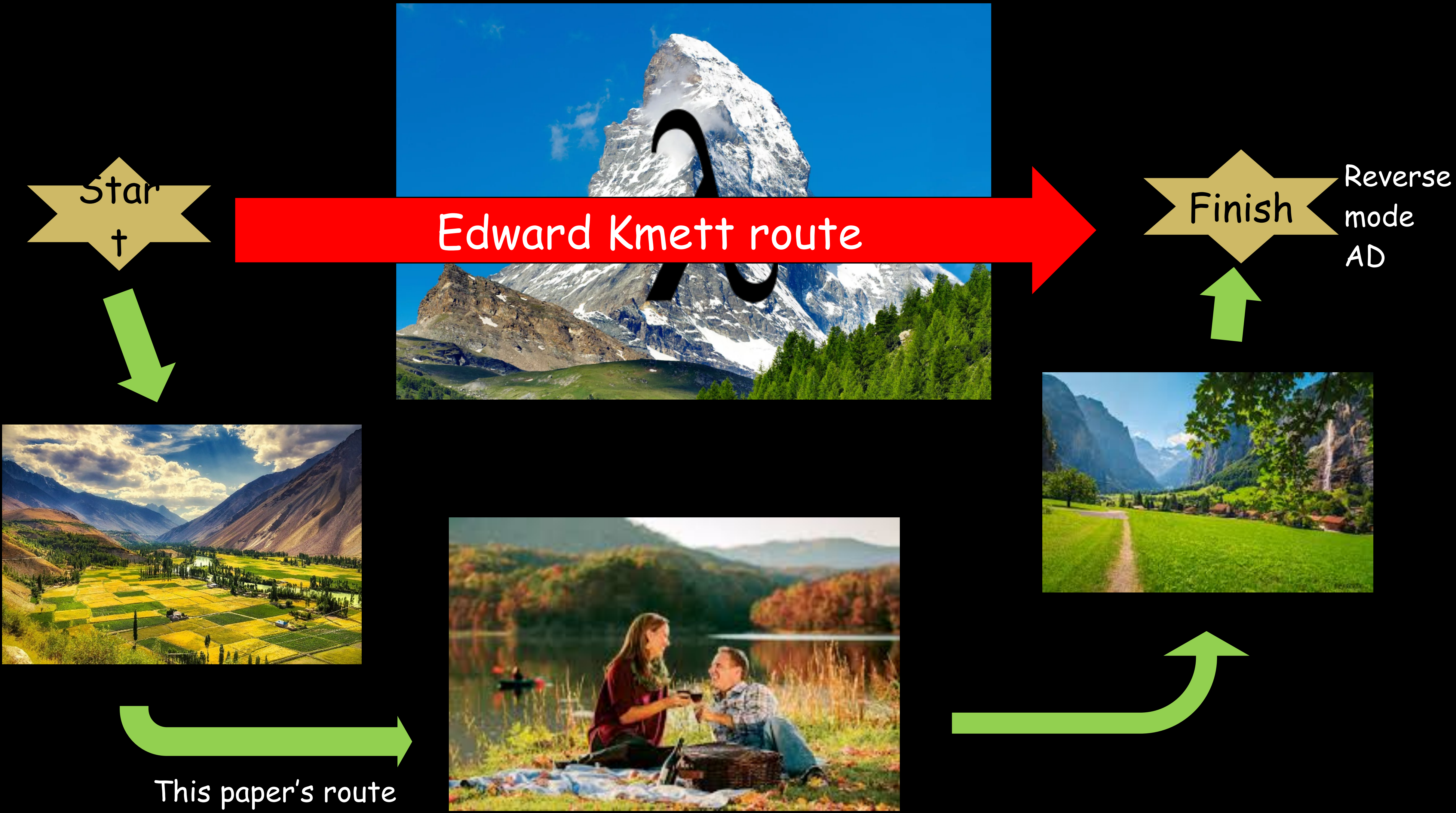
Jeffrey Siskind

Start

Finish

Reverse mode AD

This paper's route



Start

Edward Kmett route

Finish

Reverse mode AD

This paper's route

Conclusions

Take away ideas

- A series of simple steps got us
  - From a simple, “obviously correct” forward mode solution
  - To a subtle, but still correct, reverse mode solution
  - Using the dual-number idea all the way
- Simple enough to have a correctness proof

Plenty to do

- More general argument and result types
  - e.g $\left(\text{Array of } \mathbb{R}^3, \mathbb{R}^{3\times 4}\right) \to \mathbb{R}^3$
- Nested arrays
- Recursion and recursive types
- Higher order derivatives
- Fusion and checkpointing
- ~~A decent implementation~~