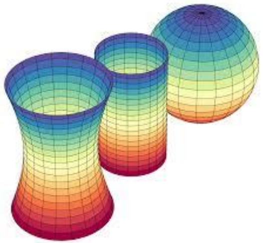
Mathematical foundations of automatic differentiation

$$\begin{tikzcd}
\mathbb{P}(X) \arrow[r, bend left, "\exists_f"] \arrow[r, bend right, "\forall_f"'] & \mathbb{P}(Y) \arrow[l, "f^{-1}"']
\end{tikzcd}$$



A tutorial - part 1

Matthijs Vákár

Utrecht University

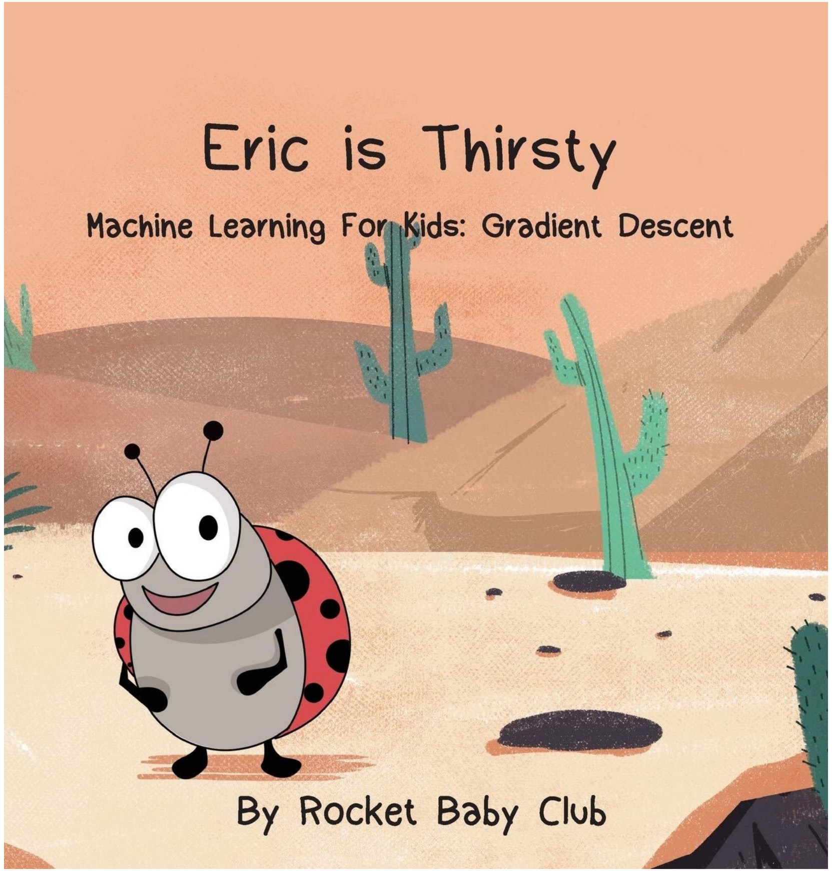
In a nutshell: some motivation

Example: regressions and derivatives

cost(w, b) = some program (* that computes e.g. $\Sigma_i(b + w * x_i - y_i)^2$ *)

Follow the derivative downhill

Best way to calculate derivatives of programs
=
Automatic Differentiation (AD)
(AKA backpropagation)



Eric is Thirsty

Machine Learning For Kids: Gradient Descent

By Rocket Baby Club

AD: many perspectives

Automatic differentiation: a long history

Robin Edwin Wengert
1964 (forward mode)

Seppo Linnainmaa
1976 (reverse mode)

Bart Speelpenning
1980 (reverse mode)

and many many more!

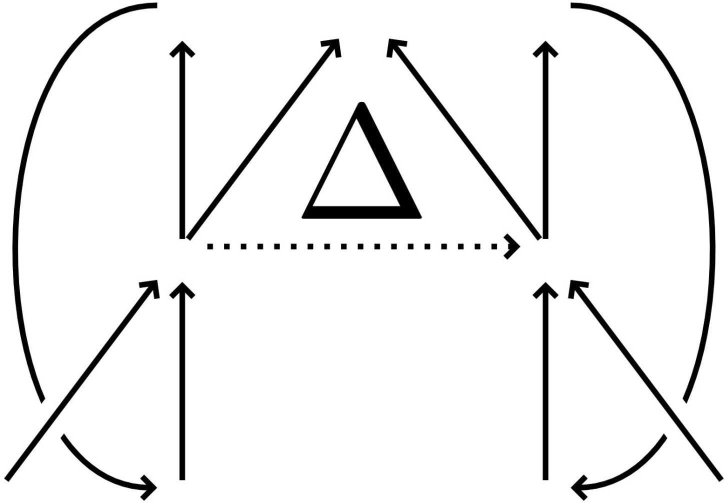
-> scientific computing community

-> machine learning community

-> programming languages community

My perspective in this tutorial:

Combinatory Homomorphic Automatic Differentiation



- Programming languages as freely generated categories

- Universal property ➡ AD definition
semantics
correctness proof } as 3 canonical homomorphic functors

Some of my fantastic collaborators!

supporting GPU
implementation

CHAD for
expressive
languages
+
mathematical
foundations

Fernando Lucatelli Nunes

Tom Smeding

Gabriele Keller

Trevor McDonell

reverse CHAD for
recursion

efficiency and
complexity of
CHAD
+
(GPU)
implementation

Gordon Plotkin

Sam Staton

crucial
initial ideas in
context of dual
numbers
forward AD

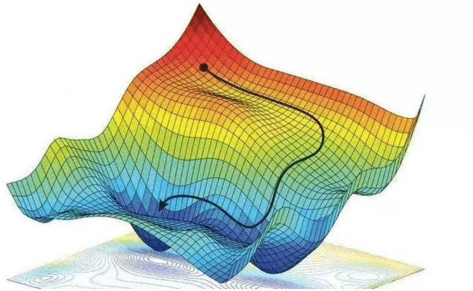
Mathieu Huot

Some topics we'll cover

- In depth: **why care**?
- **Basics of AD**: different modes
- Programming languages as **free categories**
- **Linear** types for **accumulation** effect
- Categorical structure of **Σ-type categories**
- AD as a **homomorphic** functor
- Correctness via categorical **logical relations**
- Deriving **dual numbers** CHAD from regular CHAD
- **Implementation** challenges
- **Generalizing** CHAD beyond AD

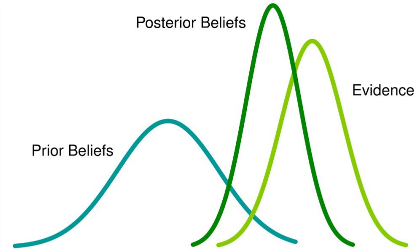
In depth: why compute derivatives?

Why compute derivatives?



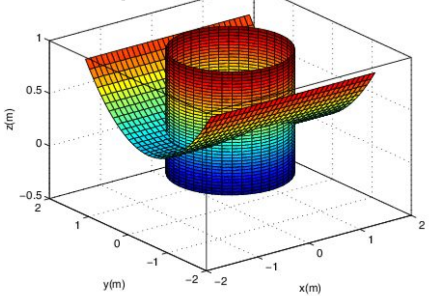
- optimization

  $\mathrm{argmin}_x f(x)$ [gradient descent & its variants]



- Bayesian inference

  $p(\theta | y) = p(\theta, y) / \int p(\theta, y)\, d\theta$ [HMC, ADVI,
  Gibbs with Gradients]



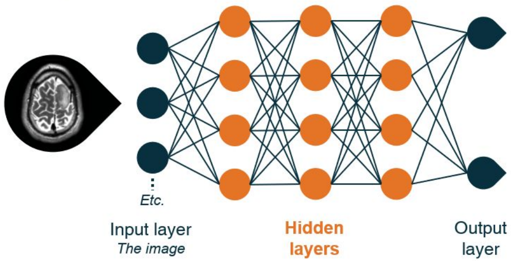
- solving systems of non-linear equations

  $f(x) = 0$ [Newton-Krylov methods]

Why compute derivatives?

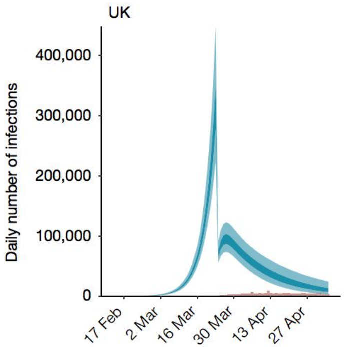
- optimization

  e.g. fitting neural nets, MLE in statistics



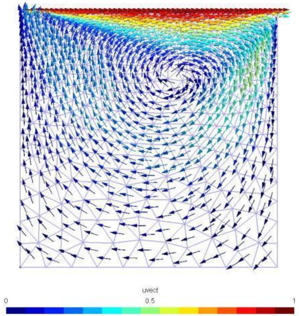
- Bayesian inference

  e.g. applied statistics and probabilistic ML



- solving systems of non-linear equations

  e.g. solve discretizations of PDEs arising in physics



Any more use cases for derivatives?

Computing derivatives?

What is a derivative? Various perspectives

- Geometry: best linear approx. $Df : \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}^m$ to a function $f : \mathbb{R}^n \to \mathbb{R}^m$ (i.e. action of function on tangent vectors)

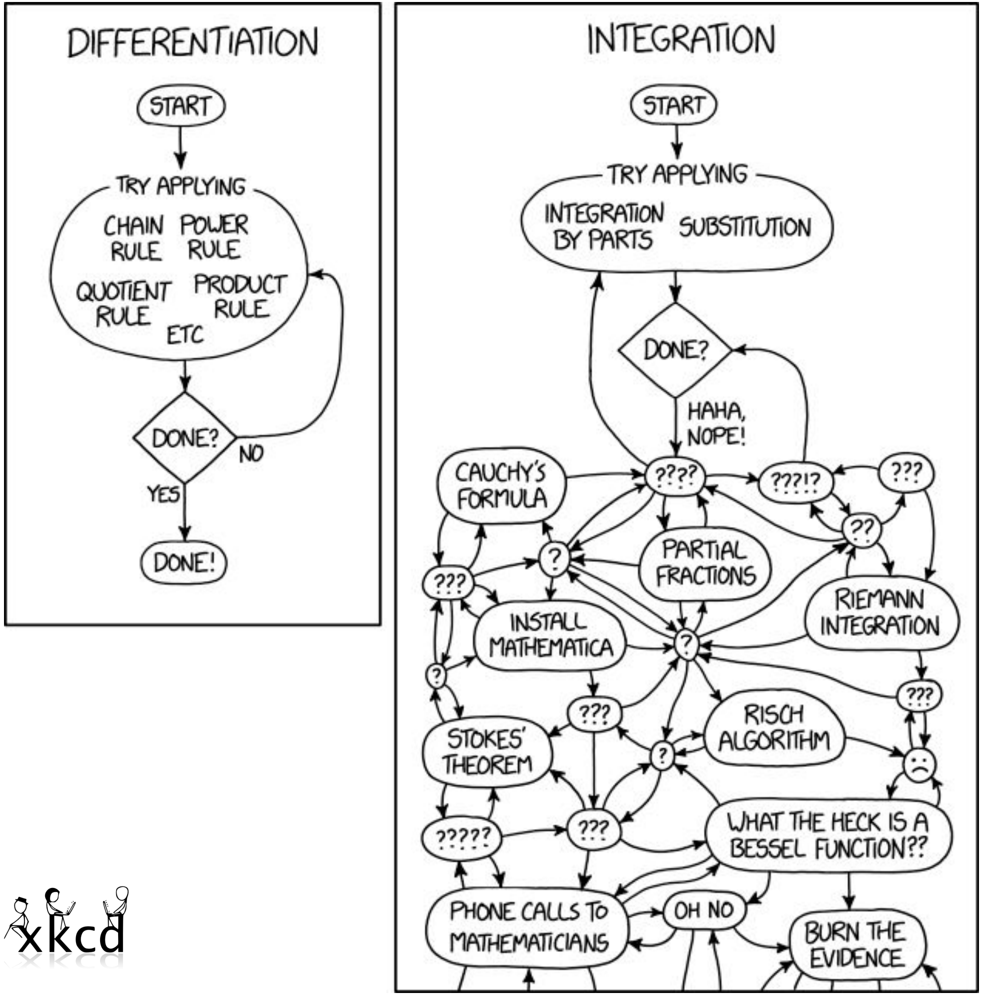
algorithm: ???

- Analysis: limit of finite differences $Df(x)(v) = \lim_{\delta \to 0} \frac{f(x + \delta \cdot v) - f(x)}{\delta}$

algorithm: finite differencing

- Algebra: symbolic transformation governed by certain rules
  - chain rule
  - rules giving derivative for primitive operations (e.g. product rule)

algorithm: automatic differentiation


DIFFERENTIATION
START
TRY APPLYING
CHAIN RULE
POWER RULE
QUOTIENT RULE
PRODUCT RULE
ETC
DONE?
NO
YES
DONE!
INTEGRATION
START
TRY APPLYING
INTEGRATION BY PARTS
SUBSTITUTION
DONE?
HAHA, NOPE!
CAUCHY'S FORMULA
????
???!?
???
??
?
???
PARTIAL FRACTIONS
INSTALL MATHEMATICA
RIEMANN INTEGRATION
?
?
???
???
RISCH ALGORITHM
STOKES' THEOREM
?
?????
???
WHAT THE HECK IS A BESSEL FUNCTION??
PHONE CALLS TO MATHEMATICIANS
OH NO
BURN THE EVIDENCE

xkcd

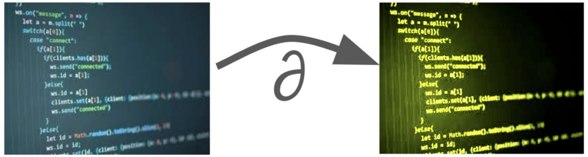
Want to compute derivatives: some desiderata

- low developer cost
- highly efficient in time and space
- parallelism exploiting (preserving)
- correct (proofs, tests)
- numerically stable (floating point arithmetic)
- extensible to new features / modular
- generally applicable

no custom derivative code for each application

reuse existing compiler infrastructures

avoid interpreter overhead



```
ws.on("message", m => {
  let a = m.split(" ")
  switch(a[0]){
    case "connect":
      if(a[1]){
        if(clients.has(a[1])){
          ws.send("connected");
          ws.id = a[1];
        }else{
          ws.id = a[1]
          clients.set(a[1], {client: [illegible]
          ws.send("connected")
        }
      }else{
        let id = Math.random().toString() [illegible]
        ws.id = id;
```

generate purely functional code

c.f.
Wang, Zheng, Decker, Wu, Essertel, Rompf

local, well-typed code transformation

Any more desiderata?

Finite differencing

Finite differencing

$$\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_m}{\partial x_1} & \cdots & \frac{\partial u_m}{\partial x_n} \end{bmatrix}$$

computes column

- differentiation as a higher-order function:

```
diff :: (Vect a, Vect b) =>
        (a -> b) -> a -> a -> b
diff    f           x    v =  (f (x + delta * v) - f x) / delta
                                  where delta = 0.00000001
```

- problems?

- numerical stability
  - adding small number to large number
  - subtracting two numbers that are almost the same

- time inefficiency for high dimensional input
  - $O(n)$ time complexity overhead over $f : \mathbb{R}^n \to \mathbb{R}^m$ for full derivative. Why?
  - same asymptotic space complexity as f (other than storing derivative values)

Automatic differentiation (AD)

Forward mode AD

Elliott
Vákár
Vákár, Smeding
Lucatelli Nunes, Vákár

AD – basic idea – forward mode

$$\left[\begin{matrix}\frac{\partial u_1}{\partial x_1}\\ \vdots\\ \frac{\partial u_m}{\partial x_1}\end{matrix}\quad\boxed{\begin{matrix}\cdots\\ \ddots\\ \cdots\end{matrix}}\quad\begin{matrix}\frac{\partial u_1}{\partial x_n}\\ \vdots\\ \frac{\partial u_m}{\partial x_n}\end{matrix}\right]$$

computes column

- differentiation as a metaprogram:

```
diff :: (Vect a, Vect b) =>
        Code (a -> b) -> Code (a -> a -o b)
diff    (f . g)       =       \x    v -> diff f (g x) (diff g x v)
diff    sin           =       \x    v -> cos x * v
diff    (*)           =       \x    v -> x_1 * v_2 + x_2 * v_1
diff    (+)           =       \x    v -> (+) v
 ⋮                                 ⋮                ⋮
```

- much more numerically stable!
- any problems?
    - chain rule: often need *g* as well as its *Dg*! -> pair *g* with *Dg*
    - common subcomputations in *g* and *Dg* -> share between *g* and *Dg*

AD - basic idea - forward mode

$$\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_m}{\partial x_1} & \cdots & \frac{\partial u_m}{\partial x_n} \end{bmatrix}$$

computes column

- sharing and pairing the primal and tangent computations *g and Dg*

```
diff :: (Vect a, Vect b) =>
        Code (a -> b) -> Code (a ->  b   ×   (a -o b)))

diff    (f . g)       =       \x -> let (g1, g2) = diff g x
                                        (f1, f2) = diff f g1 in
                                    (f1,    \v -> f2 (g2 v))

diff    sin           =       \x -> (sin x, \v -> cos x * v)

diff    (*)           =       \x -> ((*) x, \v -> x_1 * v_2 + x_2 * v_1)

diff    (+)           =       \x -> ((+) x, \v -> (+) v)
 ⋮                                 ⋮                ⋮
```

- exercise: derivative of tuples & let-binding! preserve sharing!

AD – basic idea – forward mode

$$\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_m}{\partial x_1} & \cdots & \frac{\partial u_m}{\partial x_n} \end{bmatrix}$$

computes column

```
diff :: (Vect a, Vect b) =>
        Code (a -> b)    -> Code (a ->  b    ×     (a  -o b)))

diff    x_i              =       \x -> (x_i,      \v  -> v_i)

diff    s_i              =       \x -> let (s1, s2) = diff s x in
                                       (s1_i,     \v -> (s2 v)_i)

diff    (s, t)           =       \x -> let (s1, s2) = diff s x
                                           (t1, t2) = diff t x in
                                       ((s1, t1), \v -> (s2 v, t2 v))

diff    (let x = s in t) =       \x -> let (s1, s2) = diff s x
                                           (t1, t2) = diff t (x, s1) in
                                       (t1,       \v -> t2 (v, s2 v))
```

we will improve on this later

- complexity:
    - $O(n)$ time complexity overhead over $f : \mathbb{R}^n \to \mathbb{R}^m$ for full derivative. Why?
    - space complexity overhead over *f* proportional to all intermediates of *f*. Why?
    - generates code of size $O(size(f))$. Why?

Forward AD example

Original program

```
x : real ⊦ t : real × real × real

let y = 2 * x
    z = x * y
    w = cos z
    v = (y , z ,w) in
    v
```

Forward AD transformed program

```
x : real ⊦ Dt : real × real × real ×
                (real -o real × real × real)

let y = 2 * x
    z = x * y
    w = cos z
    v = (y , z , w) in
    (v, \x' ->
              let y' = 2 * x'
                  z' = x' * y + x * y'
                  w' = -sin z * z'
                  v' = (y', z', w') in
                  v')
```

primals

tangents

Reverse mode AD

Elliott
Abadi, Wei, Plotkin, Vytiniotis, Belov
Vákár
Vákár, Smeding
Lucatelli Nunes, Vákár

AD – recall – forward mode

$$\left[\begin{matrix} \frac{\partial u_1}{\partial x_1} \\ \vdots \\ \frac{\partial u_m}{\partial x_1} \end{matrix}\ \boxed{\begin{matrix} \cdots \\ \ddots \\ \cdots \end{matrix}}\ \begin{matrix} \frac{\partial u_1}{\partial x_n} \\ \vdots \\ \frac{\partial u_m}{\partial x_n} \end{matrix}\right]$$

computes column

- compute derivative $Df : \mathbb{R}^n \to \mathbb{R}^n \multimap \mathbb{R}^m$ for $f : \mathbb{R}^n \to \mathbb{R}^m$

```
diff :: (Vect a, Vect b) =>
        Code (a -> b) -> Code (a ->  b   ×  (a -o b)))

diff    (f . g)       =       \x -> let (g1, g2) = diff g x
                                        (f1, f2) = diff f g1 in
                                    (f1,     \v -> f2 (g2 v))

diff    sin           =       \x -> (sin x, \v -> cos x * v)

diff    (*)           =       \x -> ((*) x, \v -> x_1 * v_2 + x_2 * v_1)

diff    (+)           =       \x -> ((+) x, \v -> (+) v)
  ⋮                                     ⋮                 ⋮
```

AD – basic idea – reverse mode

$$\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_m}{\partial x_1} & \cdots & \frac{\partial u_m}{\partial x_n} \end{bmatrix}$$

computes row

- compute $\overbrace{\text{derivative}}^{\text{transposed}}$ $D^t f : \mathbb{R}^n \to \mathbb{R}^m \multimap \mathbb{R}^n$ for $f : \mathbb{R}^n \to \mathbb{R}^m$

```
diff :: (Vect a, Vect b) =>
        Code (a -> b) -> Code (a ->  b    ×  (b -o a)))

diff    (f . g)       =       \x -> let (g1, g2) = diff g x
                                        (f1, f2) = diff f g1 in
                                    (f1,    \v -> g2 (f2 v))

diff    sin           =       \x -> (sin x, \v -> cos x * v)

diff    (*)           =       \x -> ((*) x, \v -> (x_1 * v, x_2 * v))

diff    (+)           =       \x -> ((+) x, \v -> (v, v))
 ⋮                                  ⋮                 ⋮
```

AD – basic idea – reverse mode

$$\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_m}{\partial x_1} & \cdots & \frac{\partial u_m}{\partial x_n} \end{bmatrix}$$

computes row

```
diff :: (Vect a, Vect b) =>
        Code (a -> b)    -> Code (a ->  b    ×     (b  -o a)))

diff    x_i              =       \x -> (x_i,       \v   -> v~i)

diff    s_i              =       \x -> let (s1, s2) = diff s x in
                                       (s1_i,      \v -> s2 (v~i))

diff    (s, t)           =       \x -> let (s1, s2) = diff s x
                                           (t1, t2) = diff t x in
                                       ((s1, t1), \v -> s2 v_1 + t2 v_2))

diff    (let x = s in t) =       \x -> let (s1, s2) = diff s x
                                           (t1, t2) = diff t (x, s1) in
                                       (t1,        \v -> let (t21, t22) = t2 v in
                                                         t21 + s2 t22              )
                    i-1
    where v~i = (0,...,0,v,0,...,0)
```

AD - basic idea - reverse mode

$$\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_m}{\partial x_1} & \cdots & \frac{\partial u_m}{\partial x_n} \end{bmatrix}$$

computes row

more about this later

- complexity (assuming a sparse vector representation for cotangents):
  - *O*(*m*) time complexity overhead over $f : \mathbb{R}^n \to \mathbb{R}^m$ for full derivative. Why?
  - space complexity overhead over *f* proportional to all intermediates of *f*. Why?
  - generates code of size *O*(*size*(*f*)). Why?

Reverse AD example

Original program

```
x : real × real × real × real
        ⊦ t : real

let y  = x1 * x4 + 2 * x2
    z  = y * x3
    w  = z + x4
    u1 = sin w
    u2 = cos w
    v  = u1 + u2 in
    v
```

Reverse AD transformed program

```
x : real × real × real × real
  ⊦ t : real × (real -o real × real × real ×
real)

let y  = x1 * x4 + 2 * x2
    z  = y * x3
    w  = z + x4
    u1 = sin w
    u2 = cos w
    v = u1 + u2 in
    (v , \v' ->
           let u2' = v'
               u1' = v'
               w' = cos w * u1' - sin w * u2'
               z' = w'
               y' = z' * x3
               x1' = y' * x4
               x2' = 2 * y'
               x3' = y * z'
               x4' = x1 * y' + w' in
               (x1' , x2' , x3' , x4'))
```

primals

cotangents

duplication -> addition

Dual numbers forward mode AD

Kmett
Shaikhha, Fitzgibbon, Vytiniotis, Peyton Jones
Huot, Staton, Vákár

AD – basic idea – fwd dual numbers

$$\left[\begin{matrix}\frac{\partial u_1}{\partial x_1} \\ \vdots \\ \frac{\partial u_m}{\partial x_1}\end{matrix}\quad \boxed{\begin{matrix}\cdots \\ \ddots \\ \cdots\end{matrix}}\quad \begin{matrix}\frac{\partial u_1}{\partial x_n} \\ \vdots \\ \frac{\partial u_m}{\partial x_n}\end{matrix}\right]$$

computes column

- compute derivative

```
diff :: (Vect a, Vect b) =>
        Code (a -> b) -> Code ((a × a) -> (b × b))

diff    (f . g)       =       diff f . diff g

diff    sin           =       \(x, x') -> (sin x, x' * cos x)

diff    (*)           =       \(x, x') -> ((*) x, x_1 * x'_2 + x_2 * x'_1)

diff    (+)           =       \(x, x') -> ((+) x, (+) x')
 ⋮                                  ⋮                 ⋮
```

AD – basic idea – fwd dual numbers

$$\left[\begin{matrix} \frac{\partial u_1}{\partial x_1} \\ \vdots \\ \frac{\partial u_m}{\partial x_1} \end{matrix} \;\; \boxed{\begin{matrix} \cdots \\ \ddots \\ \cdots \end{matrix}} \;\; \begin{matrix} \frac{\partial u_1}{\partial x_n} \\ \vdots \\ \frac{\partial u_m}{\partial x_n} \end{matrix}\right]$$

computes column

```
diff :: (Vect a, Vect b) =>
        Code (a -> b) -> Code ((a × a) -> (b × b))

diff    x_i             =    \(x, x') -> (x_i, x'_i)

diff    s_i             =    \(x, x') -> let (s1, s2) = diff s (x, x') in
                                         (s1_i, s2_i)

diff    (s, t)          =    \(x, x') -> let (s1, s2) = diff s (x, x')
                                             (t1, t2) = diff t (x, x') in
                                         ((s1, t1),(s2, t2))

diff    (let x = s in t) =   \(x, x') -> let (s1, s2) = diff s (x, x') in
                                             diff t ((x, s1), (x', s2))
```

- complexity:
    - $O(n)$ time complexity overhead over $f : \mathbb{R}^n \to \mathbb{R}^m$ for full derivative. Why?
    - $O(1)$ space complexity overhead over f. Why?
    - generates code of size $O(size(f))$. Why?

Dual numbers forward AD example

Original program

```
x : real ⊦ t : real ×
               real ×
               real


let y = 2 * x
    z = x * y
    w = cos z
    v = (y , z ,w) in
    v
```

Dual numbers forward AD transformed program

```
(x, x') : real × real ⊦ Dt : (real × real)
×
                              (real × real)
×
                              (real × real)


let (y, y') = (2 * x, 2 * x')
    (z, z') = (x * y, x' * y + x * y')
    (w, w') = (cos z, -sin z * z')
    (v, v') = ((y , z ,w), (y', z', w')) in
    (v, v')
```

mixed primals and tangents

Dual numbers reverse mode AD

Kmett
Abadi, Plotkin?
Mak, Ong?
Brunel, Mazza, Pagani
Huot, Staton, Vákár
Mazza, Pagani
Krawiec, Peyton Jones, Krishnaswami, Ellis, Eisenberg, Fitzgibbon

AD – basic idea – rev dual numbers

$$\begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_m}{\partial x_1} & \cdots & \frac{\partial u_m}{\partial x_n} \end{bmatrix}$$

computes row

- $\overbrace{\text{compute}}^{\text{transposed}}$ derivative

```
diff :: (Vect a, Vect b) =>
        Code (a -> b) -> Code ((a × (a -o c)) -> (b × (b -o c)))

diff    (f . g)        =       diff f . diff g

diff    sin            =       \(x, x') -> (sin x, \z -> x' (z * cos x))

diff    (*)            =       \(x, x') -> ((*) x, \z -> x'(x_1 * z, x_2 * z))

diff    (+)            =       \(x, x') -> ((+) x, \z -> x'(z, z))
 ⋮                                    ⋮                 ⋮
```

Michele Pagani's talk!!!

- complexity:
  - $O(m)$ time complexity overhead over $f : \mathbb{R}^n \to \mathbb{R}^m$ for full derivative.
  - space complexity overhead over *f* proportional to all intermediates of *f*.
  - generates code of size *O*(*size*(*f*)). Why?

but only with custom operational semantics for linear functions!

Dual numbers reverse AD example

Original program

```
x : real × real × real × real
        ⊦ t : real


let y  = x1 * x4 + 2 * x2
    z  = y * x3
    w  = z + x4
    u1 = sin w
    u2 = cos w
    v  = u1 + u2 in
    v
```

aargh! duplicate computation
from w' onwards, for both
contributions!

Brunel, Mazza, Pagani
solution: linear factoring rule
`wk w' + wk w'' ->  wk (w' + w'')`
i.e. custom interpreter

Dual numbers reverse AD transformed program

```
x : real × real × real × real
  ⊦ t : real × (real -o
                real × real × real × real)

let y = x1 * x4 + 2 * x2
    z = y * x3
    w = z + x4
    u1 = sin w
    u2 = cos w
    v = u1 + u2 in
(v, \v' -> let u1' = v'
           let u2' = v' in
           (let w' = cos w * u1'
            let z' = w'
            let y' = z' * x3 in
            (y' * x4, 2 * y', y * z',  x1 * y' + w'))
          + (let w' = -sin w * u2'
            let z' = w'
            let y' = z' * x3 in
            (y' * x4, 2 * y', y * z',  x1 * y' + w')))
```

primals

cotangents