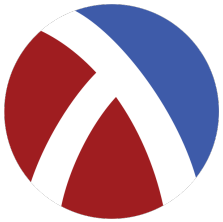
A functional tour of automatic differentiation with Racket

Oliver Strickson

2020-02-14

Kraków



lambda
DAλS

Oliver Strickson
Research Software Engineer
Research Engineering Group

The Alan Turing Institute

Photo credit: https://commons.wikimedia.org/wiki/User:Patche99z

Overview

- Differentiation
- Automatic differentiation algorithm
- Implementation by program tracing
- Implementation by program transformation
- Local program transformation: Dual numbers
- Resources

Differentiation

Differentiation

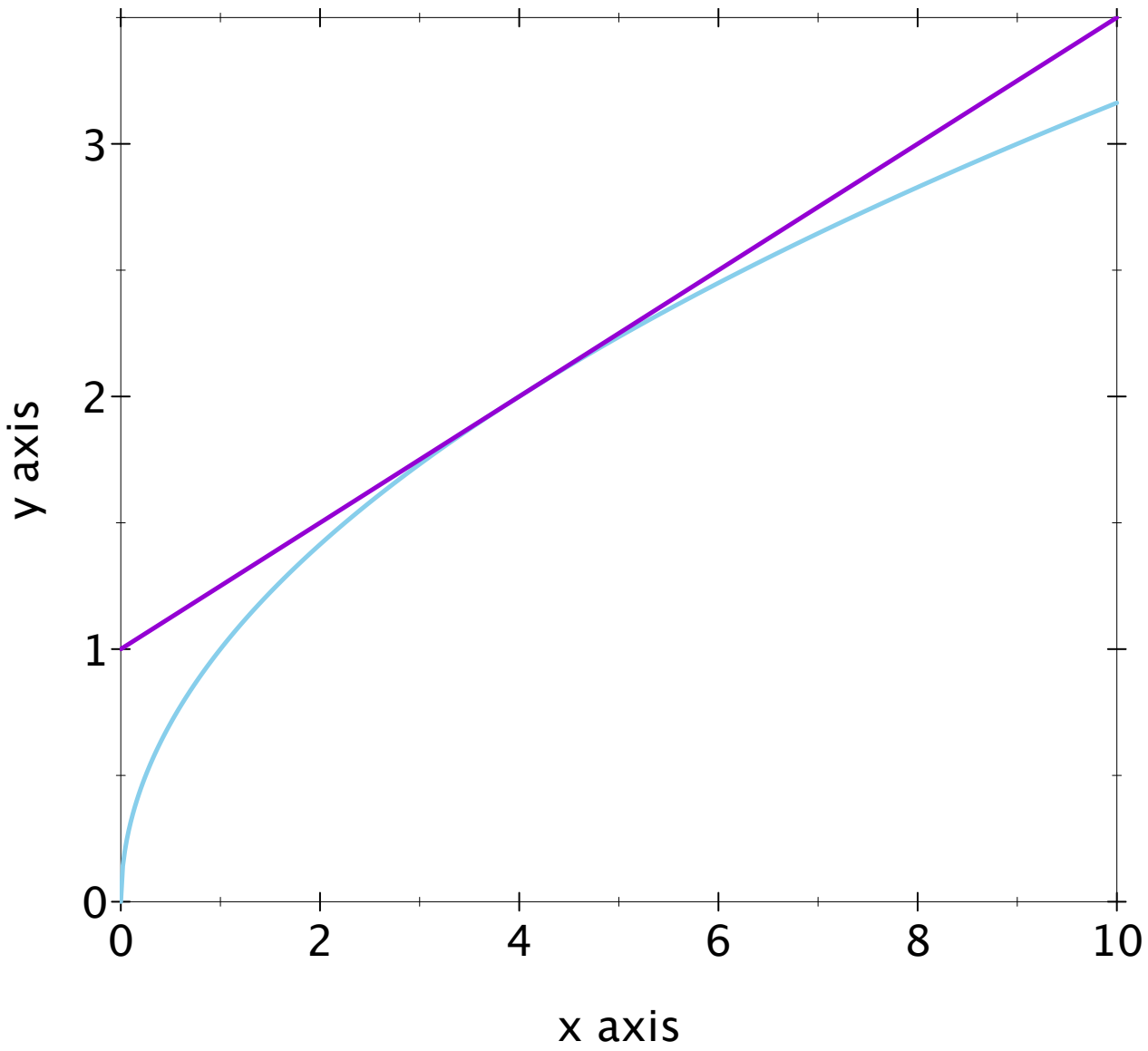
The best linear approximation to a function about a point (if it exists)

Differentiation

The best linear approximation to a function about a point (if it exists)

Function f or `f`

Derivative Df or `(D f)`


y axis
x axis
0
1
2
3
0
2
4
6
8
10

Differentiation

function $f(x)$

find a with

$$f(x) - f(x_0) \approx a\,(x - x_0)$$

Differentiation

function $f(x)$

find a with

$$f(x) - f(x_0) \approx a\,(x - x_0)$$

$$f(x) - f(x_0) = a\,(x - x_0) + o(x - x_0)$$

Differentiation

function $f(x)$

find a with

$$f(x) - f(x_0) \approx a\,(x - x_0)$$

$$f(x) - f(x_0) = a\,(x - x_0) + o(x - x_0)$$

$$f(x) - f(x_0) = Df(x_0)\,(x - x_0) + o(x - x_0)$$

Differentiation

function $f(x, y)$

find a, b with

$$f(x, y) - f(x_0, y_0) \approx a\,(x - x_0) + b\,(y - y_0)$$

Differentiation

function $f(x, y)$

find a, b with

$$f(x, y) - f(x_0, y_0) \approx a\,(x - x_0) + b\,(y - y_0)$$

$$f(x, y) - f(x_0, y_0) \approx D_0 f(x_0, y_0)\,(x - x_0) + D_1 f(x_0, y_0)\,(y - y_0)$$

Differentiation

function $f(x, y)$

find a, b with

$$f(x, y) - f(x_0, y_0) \approx a\,(x - x_0) + b\,(y - y_0)$$

$$f(x, y) - f(x_0, y_0) \approx D_0 f(x_0, y_0)\,(x - x_0) + D_1 f(x_0, y_0)\,(y - y_0)$$

Partial derivative $D_i f$ or `(partial i f)`

Differentiation

function $f(x, y)$

find a, b with

$$f(x, y) - f(x_0, y_0) \approx a\,(x - x_0) + b\,(y - y_0)$$

$$f(x, y) - f(x_0, y_0) \approx D_0 f(x_0, y_0)\,(x - x_0) + D_1 f(x_0, y_0)\,(y - y_0)$$

Partial derivative $D_i f$ or **`(partial i f)`**

$$Df(x, y) = (D_0 f(x, y), D_1 f(x, y))$$

```
(define ((partial i f) . xs)
  (case f
    ; ...
    [(exp)        (case i
                    [(0)    (exp (first xs))]
                    [else   (err)])]
    ; ...
    [else (err)]))
```

```
(define ((partial i f) . xs)
  (case f
    ; ...
    [(*)          (case i
                    [(0)    (list-ref xs 1)]
                    [(1)    (list-ref xs 0)]
                    [else   (err)])]
    ; ...
    [else (err)]))
```

Composition

Composition

```
([D (compose g f)] x)
= (* ([D g] (f x))
     ([D f] x))
```

Composition

$$f(x, y) = g(u(x, y), v(x, y))$$

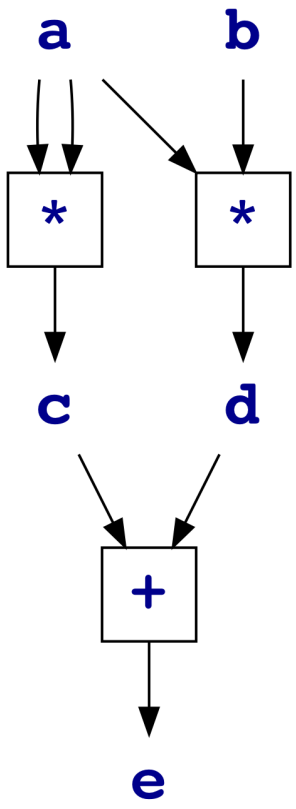
$$\begin{aligned} Df(x, y) = & \\ & D_0 g(u(x, y), v(x, y))\, Du(x, y) \\ + & D_1 g(u(x, y), v(x, y))\, Dv(x, y) \end{aligned}$$

Arithmetic expressions

```
(define (f a b)
  (+ (* a a) (* a b)))
```

Arithmetic expressions

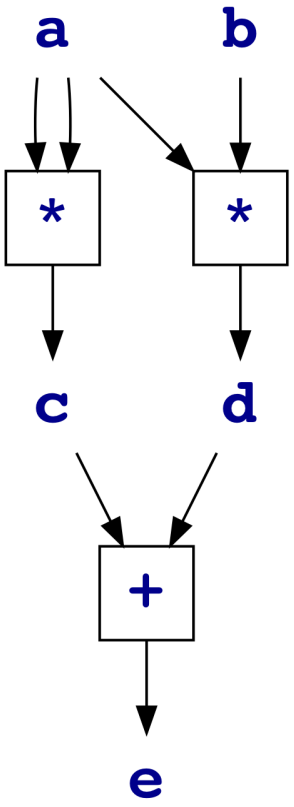
```
(define (f a b)
  (+ (* a a) (* a b)))
```




```
c ← (* a a)
d ← (* a b)
e ← (+ c d)
```

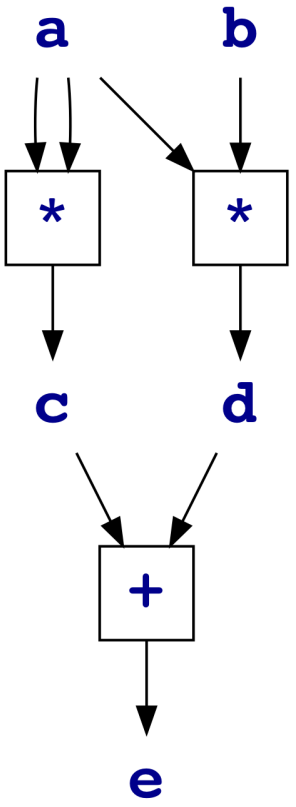
Automatic differentiation

Compute $\mathrm{D}f(a, b)$



Automatic differentiation

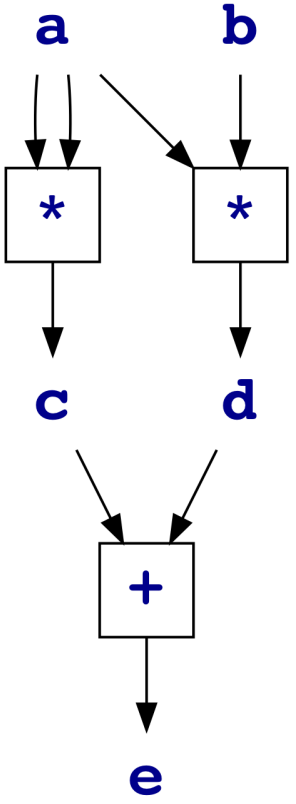
Compute $Df(a, b)$



$$\frac{da}{dr} = 1$$
$$\frac{db}{dr} = 0$$

Automatic differentiation

Compute $Df(a, b)$



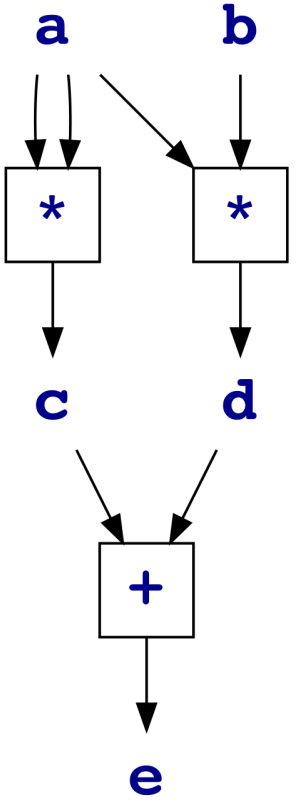
$$\frac{da}{dr} = 1$$

$$\frac{db}{dr} = 0$$

$$\frac{dc}{dr} = D_0(*)(a, a)\frac{da}{dr} + D_1(*)(a, a)\frac{da}{dr}$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{da}{dr} = 1$$

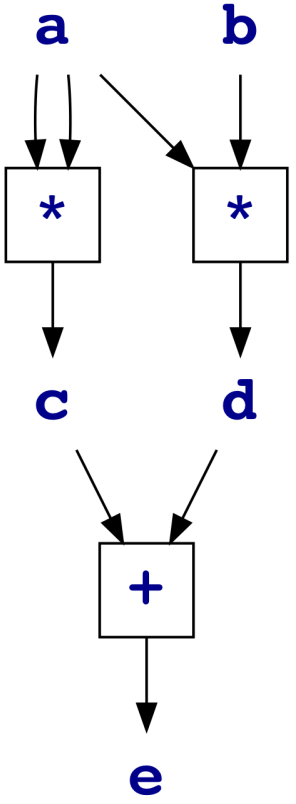
$$\frac{db}{dr} = 0$$

$$\frac{dc}{dr} = D_0(*)(a, a)\frac{da}{dr} + D_1(*)(a, a)\frac{da}{dr}$$

$$\frac{dd}{dr} = D_0(*)(a, b)\frac{da}{dr} + D_1(*)(a, b)\frac{db}{dr}$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{da}{dr} = 1$$

$$\frac{db}{dr} = 0$$

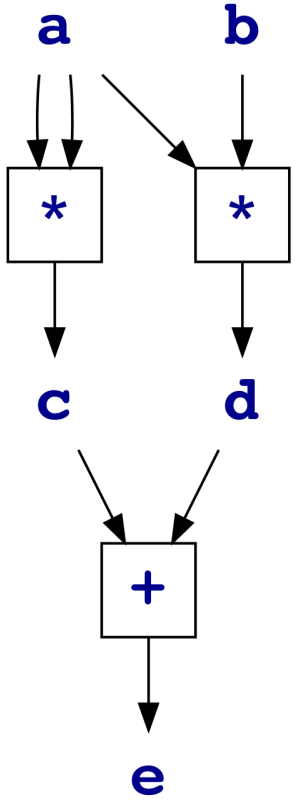
$$\frac{dc}{dr} = D_0(*)(a, a)\frac{da}{dr} + D_1(*)(a, a)\frac{da}{dr}$$

$$\frac{dd}{dr} = D_0(*)(a, b)\frac{da}{dr} + D_1(*)(a, b)\frac{db}{dr}$$

$$\frac{de}{dr} = D_0(+)(c, d)\frac{dc}{dr} + D_1(+)(c, d)\frac{dd}{dr}$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{da}{dr} = 1$$

$$\frac{db}{dr} = 0$$

$$\frac{dc}{dr} = D_0(*)(a, a)\frac{da}{dr} + D_1(*)(a, a)\frac{da}{dr}$$

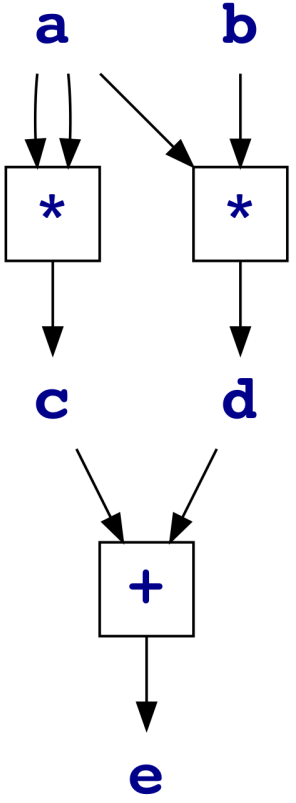
$$\frac{dd}{dr} = D_0(*)(a, b)\frac{da}{dr} + D_1(*)(a, b)\frac{db}{dr}$$

$$\frac{de}{dr} = D_0(+)(c, d)\frac{dc}{dr} + D_1(+)(c, d)\frac{dd}{dr}$$

$$D_0 f(a, b) = \frac{de}{dr}$$

Automatic differentiation

Compute $Df(a, b)$



$$\boxed{\begin{aligned}\frac{da}{dr} &= 1\\ \frac{db}{dr} &= 0\end{aligned}}$$

$$\frac{dc}{dr} = D_0(*)(a, a)\frac{da}{dr} + D_1(*)(a, a)\frac{da}{dr}$$

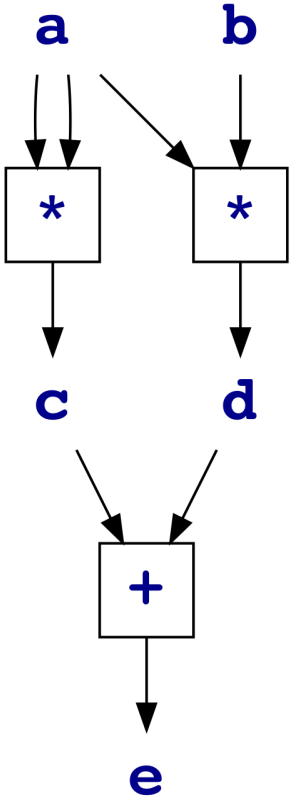
$$\frac{dd}{dr} = D_0(*)(a, b)\frac{da}{dr} + D_1(*)(a, b)\frac{db}{dr}$$

$$\frac{de}{dr} = D_0(+)(c, d)\frac{dc}{dr} + D_1(+)(c, d)\frac{dd}{dr}$$

$$D_0 f(a, b) = \frac{de}{dr}$$

Automatic differentiation

Compute $Df(a, b)$



$$\boxed{\begin{aligned}\frac{da}{dr} &= 0\\ \frac{db}{dr} &= 1\end{aligned}}$$

$$\frac{dc}{dr} = D_0(*)(a, a)\frac{da}{dr} + D_1(*)(a, a)\frac{da}{dr}$$

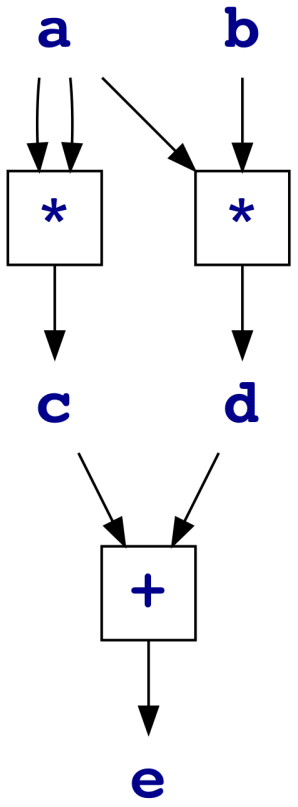
$$\frac{dd}{dr} = D_0(*)(a, b)\frac{da}{dr} + D_1(*)(a, b)\frac{db}{dr}$$

$$\frac{de}{dr} = D_0(+)(c, d)\frac{dc}{dr} + D_1(+)(c, d)\frac{dd}{dr}$$

$$D_1 f(a, b) = \frac{de}{dr}$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{da}{dr} = 0$$

$$\frac{db}{dr} = 1$$

$$\frac{dc}{dr} = D_0(*)(a, a)\frac{da}{dr} + D_1(*)(a, a)\frac{da}{dr}$$

$$\frac{dd}{dr} = D_0(*)(a, b)\frac{da}{dr} + D_1(*)(a, b)\frac{db}{dr}$$

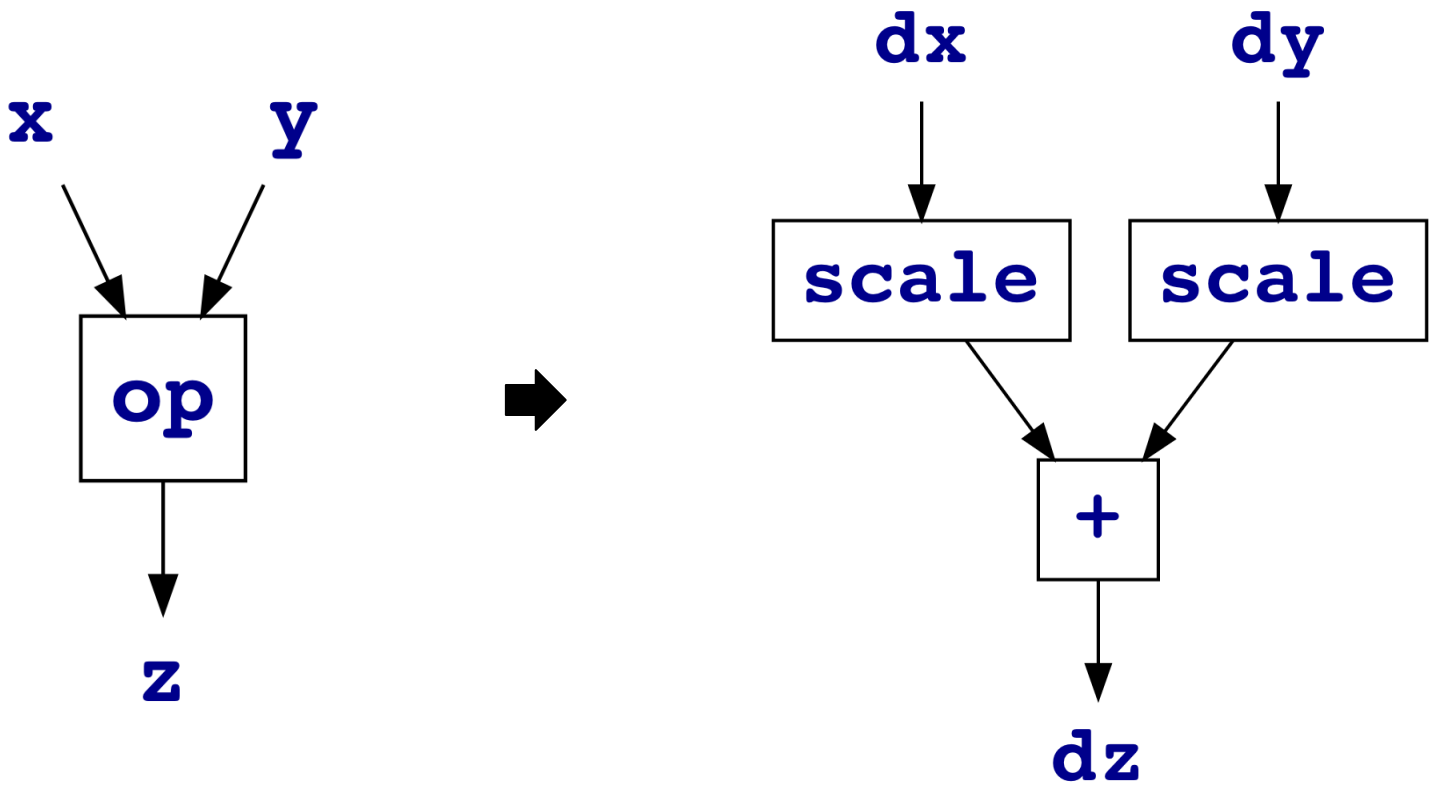
$$\frac{de}{dr} = D_0(+)(c, d)\frac{dc}{dr} + D_1(+)(c, d)\frac{dd}{dr}$$

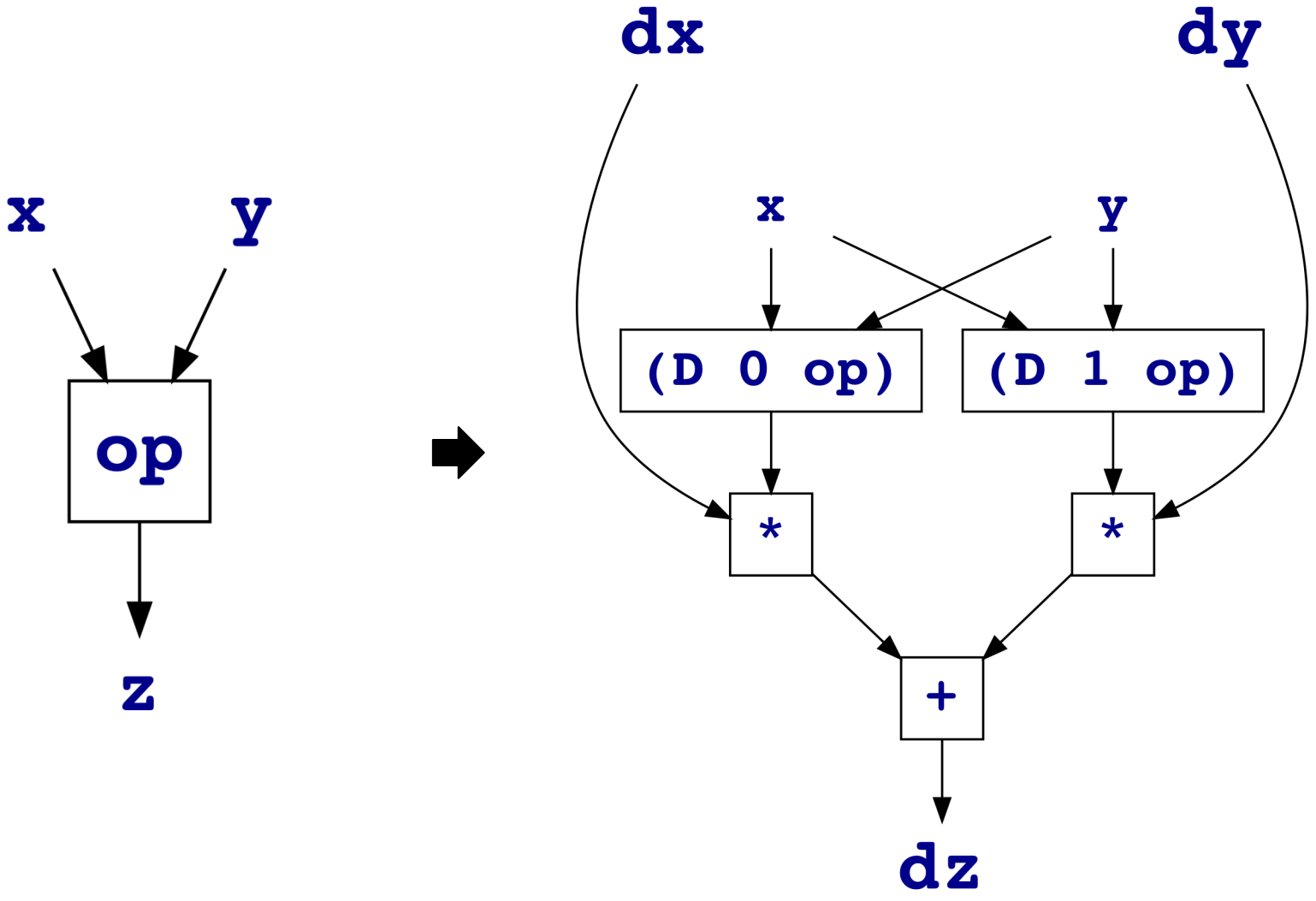
$$D_1 f(a, b) = \frac{de}{dr}$$

Forward mode

Often write **dx** instead of $\frac{dx}{dr}$

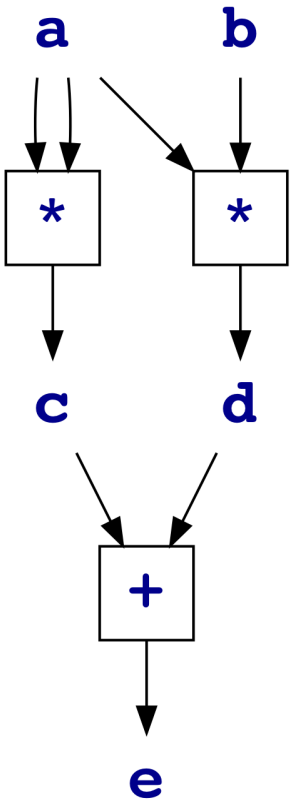
Known as *perturbation variables*


x
y
op
z
dx
dy
scale
scale
+
dz


x
y
op
z
dx
dy
x
y
(D 0 op)
(D 1 op)
*
*
+
dz

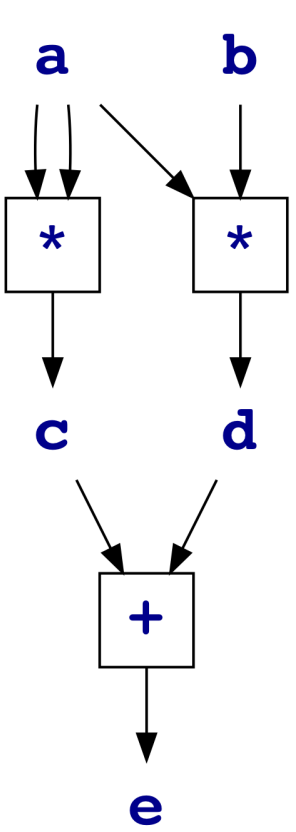
Automatic differentiation

Compute $\mathrm{D}\mathrm{f}(\mathrm{a}, \mathrm{b})$



Automatic differentiation

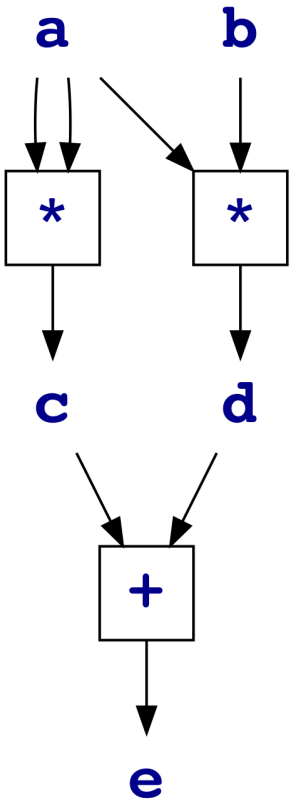
Compute $Df(a, b)$



$$\frac{ds}{de} = 1$$

Automatic differentiation

Compute $Df(a, b)$

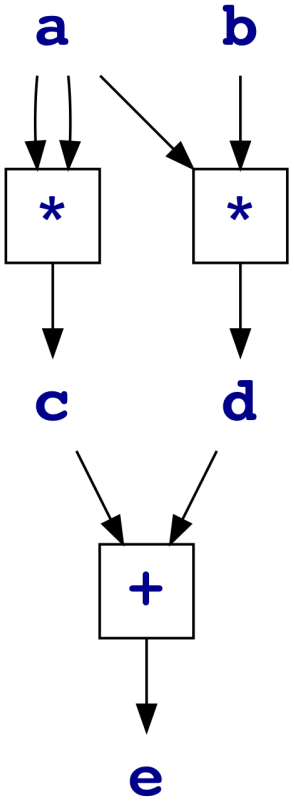


$$\frac{ds}{de} = 1$$

$$\frac{ds}{dd} = D_1(+)(c, d)\frac{ds}{de}$$

Automatic differentiation

Compute $Df(a, b)$



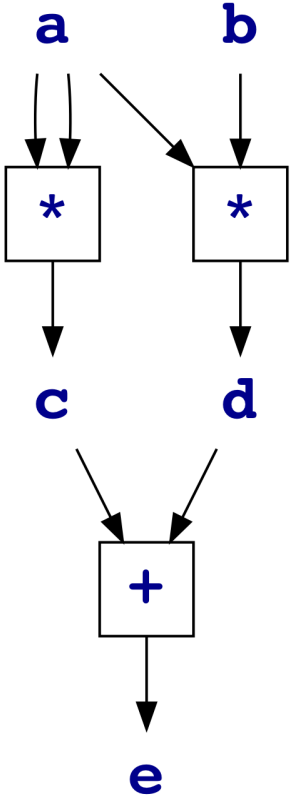
$$\frac{ds}{de} = 1$$

$$\frac{ds}{dd} = D_1(+)(c, d)\frac{ds}{de}$$

$$\frac{ds}{dc} = D_0(+)(c, d)\frac{ds}{de}$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{ds}{de} = 1$$

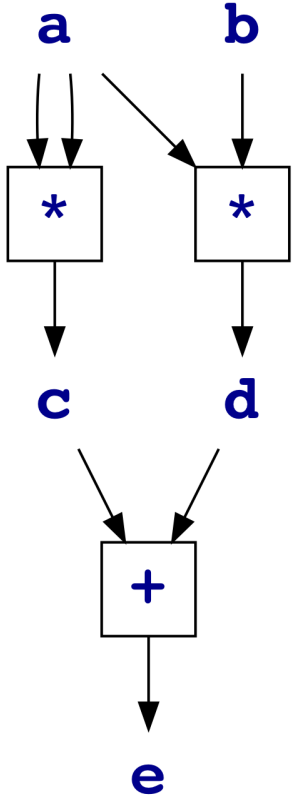
$$\frac{ds}{dd} = D_1(+)(c, d)\frac{ds}{de}$$

$$\frac{ds}{dc} = D_0(+)(c, d)\frac{ds}{de}$$

$$\frac{ds}{db} = D_1(*)(a, b)\frac{ds}{dd}$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{ds}{de} = 1$$

$$\frac{ds}{dd} = D_1(+)(c, d)\frac{ds}{de}$$

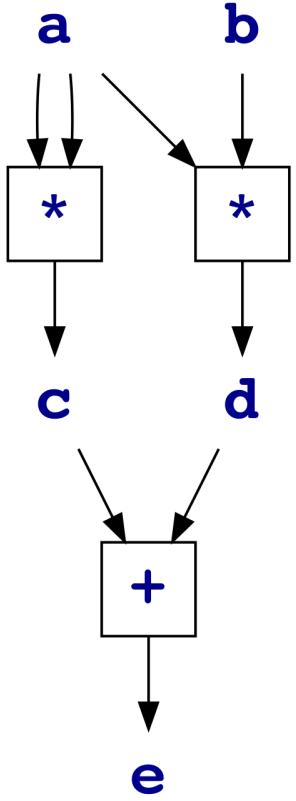
$$\frac{ds}{dc} = D_0(+)(c, d)\frac{ds}{de}$$

$$\frac{ds}{db} = D_1(*)(a, b)\frac{ds}{dd}$$

$$\frac{ds}{da} = D_0(*)(a,a)\frac{ds}{dc} + D_1(*)(a, a)\frac{ds}{dc} + D_0(*)(a, b)\frac{ds}{dd}$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{ds}{de} = 1$$

$$\frac{ds}{dd} = D_1(+)(c, d)\frac{ds}{de}$$

$$\frac{ds}{dc} = D_0(+)(c, d)\frac{ds}{de}$$

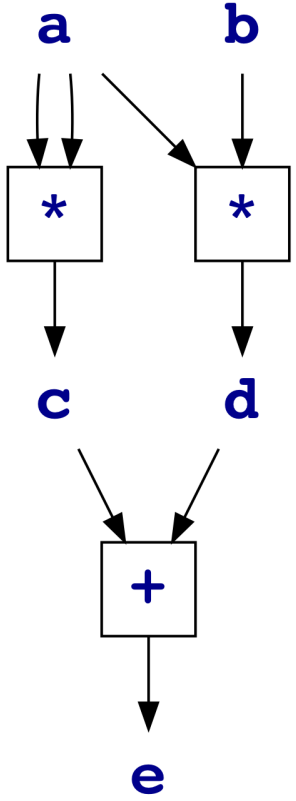
$$\frac{ds}{db} = D_1(*)(a, b)\frac{ds}{dd}$$

$$\frac{ds}{da} = D_0(*)(a,a)\frac{ds}{dc} + D_1(*)(a, a)\frac{ds}{dc} + D_0(*)(a, b)\frac{ds}{dd}$$

$$Df(a, b) = \left(\frac{ds}{da}, \frac{ds}{db}\right)$$

Automatic differentiation

Compute $Df(a, b)$



$$\frac{ds}{de} = 1$$

$$\frac{ds}{dd} = D_1(+)(c, d)\frac{ds}{de}$$

$$\frac{ds}{dc} = D_0(+)(c, d)\frac{ds}{de}$$

$$\frac{ds}{db} = D_1(*)(a, b)\frac{ds}{dd}$$

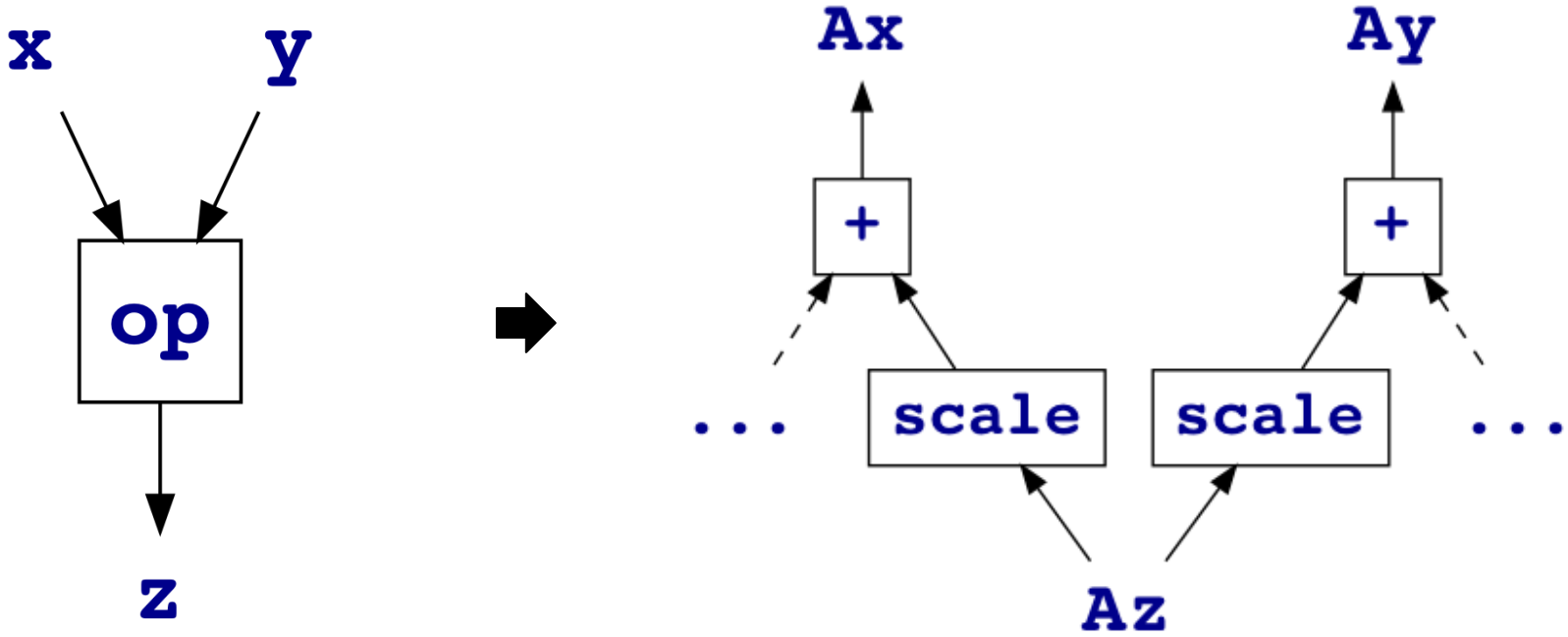
$$\frac{ds}{da} = D_0(*)(a,a)\frac{ds}{dc} + D_1(*)(a, a)\frac{ds}{dc} + D_0(*)(a, b)\frac{ds}{dd}$$

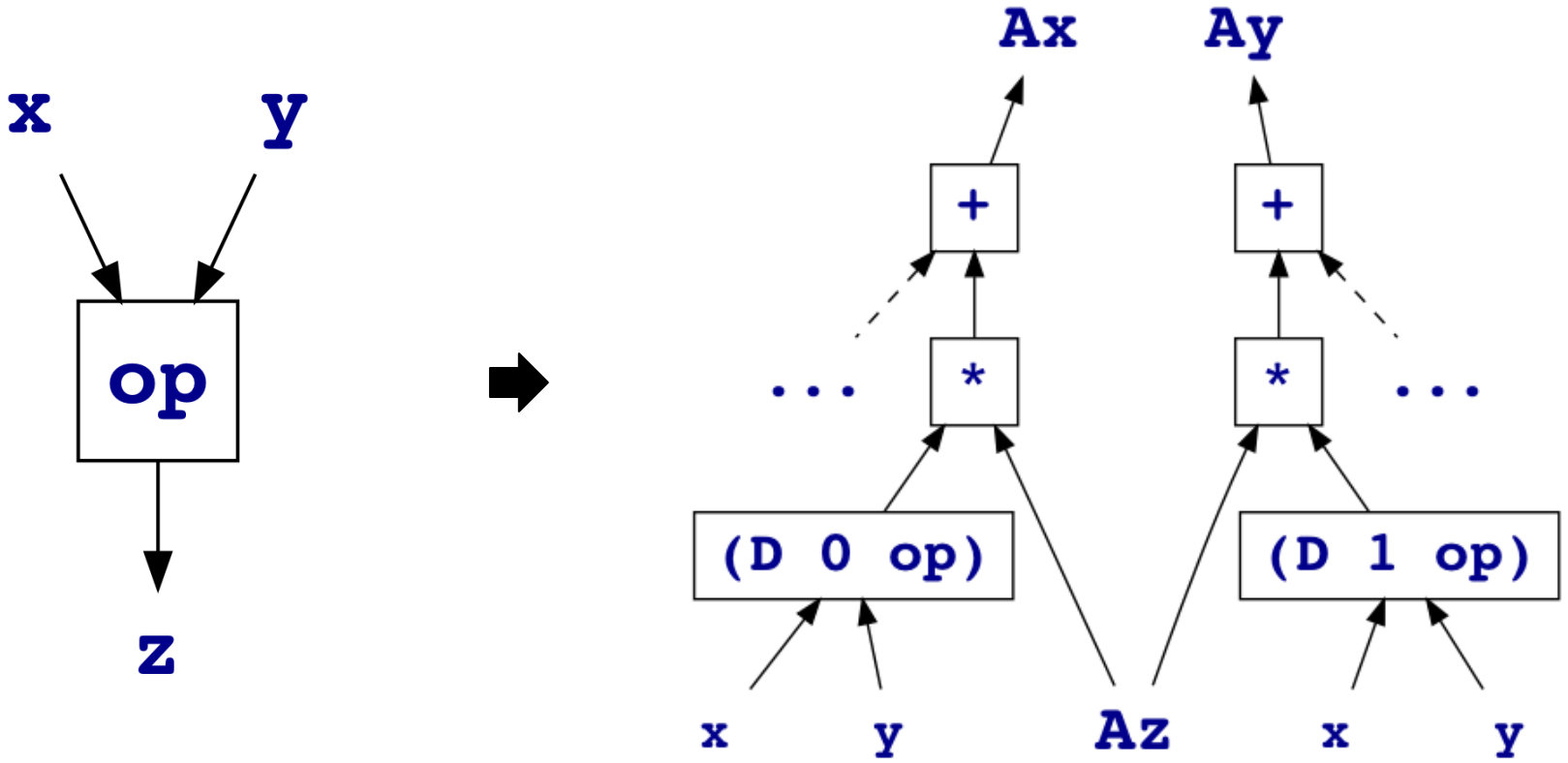
$$Df(a, b) = \left(\frac{ds}{da}, \frac{ds}{db}\right)$$

Reverse mode

Often write **Ax** instead of $\frac{ds}{dx}$

Known as *sensitivity variables* or *adjoints*


x
y
op
z
Ax
Ay
+
+
scale
scale
...
...
Az


x
y
op
z
Ax
Ay
+
+
...
*
*
...
(D 0 op)
(D 1 op)
x
y
Az
x
y

Idea: every value returned by a program was determined from a particular (dynamic) computational graph.

Differentiate *that*

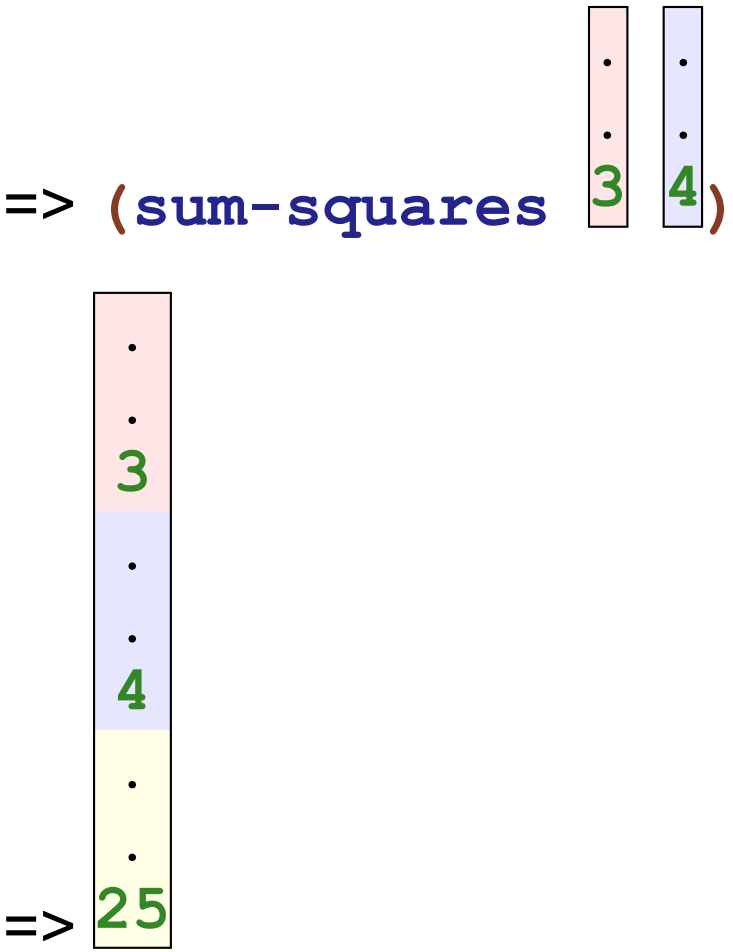
Tracing program execution

We want to make a particular type of trace, which is:

- flat
- topologically sorted
- contains only *primitive operations*

```
  (sum-squares x y)
=> (sum-squares 3 4)
=> 25
```

```
(sum-squares x y)
```



```
x
```

=> 3, as

%1	(constant 3)	3

```
y
```

=> 4, as

%2	(constant 4)	4

```
(sum-squares x y)
```

=> 25, as

%1	(constant 3)	3
%2	(constant 4)	4
%3	(app * %1 %1)	9
%4	(app * %2 %2)	16
%5	(app + %3 %4)	25

Let’s make a little language that does this...

What is a language?

Functions

Other special forms (**`if`**, **`λ`**, **`define`**, **`require`**, ...)

Evaluation model

Literal data

Syntax

assignments

```
(struct assignment (id expr val)
  #:transparent
  #:guard (struct-guard/c symbol? expr? any/c))
```

assignments

```
(struct assignment (id expr val)
  #:transparent
  #:guard (struct-guard/c symbol? expr? any/c))

                    assignment?
                    assignment-id
                    assignment-expr
                    assignment-val
```

assignments

```
(struct assignment (id expr val)
  #:transparent
  #:guard (struct-guard/c symbol? expr? any/c))

                  assignment?
                  assignment-id
                  assignment-expr
                  assignment-val

    (define (expr? e)
      (match e
        [(list 'constant _) #t]
        [(list 'app (? symbol? _) ..1) #t]
        [_ #f]))
```

trace

```
(struct trace (assignments))
```

trace

```
(struct trace (assignments))

   (trace-add tr assgn)
   (trace-append trs ...)
```

trace

```
(struct trace (assignments))

(trace-add tr assgn)
(trace-append trs ...)
```

top of a trace is the most recent assignment

```
(top tr)
```

trace

```
(struct trace (assignments))

(trace-add tr assgn)
(trace-append trs ...)
```

top of a trace is the most recent assignment

```
(top tr)

(top-val tr)
(top-id tr)
(top-expr tr)
```

trace-lang functions

```
(define (+& a b)
  (trace-add
   (trace-append a b)
   (make-assignment
    #:expr (list 'app '+ (top-id a) (top-id b))
    #:val  (+ (top-val a) (top-val b)))))
```

trace-lang functions

```
(define (*& a b)
  (trace-add
   (trace-append a b)
   (make-assignment
    #:expr (list 'app '* (top-id a) (top-id b))
    #:val  (* (top-val a) (top-val b)))))
```

trace-lang functions

```
(define (exp& x)
  (trace-add
   x
   (make-assignment
    #:expr (list 'app 'exp (top-id x))
    #:val  (exp (top-val x)))))
```

```
(define (f a ...)
  (trace-add
   (trace-append a ...)
   (make-assignment
    #:expr (list 'app f-name (top-id a) ...)
    #:val  (let ([a (top-val a)] ...)
             body ...))))
```

```
#'(define (f a ...)
    (trace-add
     (trace-append a ...)
     (make-assignment
      #:expr (list 'app f-name (top-id a) ...)
      #:val  (let ([a (top-val a)] ...)
               body ...))))
```

```
(define-syntax (define-traced-primitive stx)
  (syntax-case stx ()
    [(_ (f a ...) f-name
        body ...)
     #'(define (f a ...)
         (trace-add
          (trace-append a ...)
          (make-assignment
           #:expr (list 'app f-name (top-id a) ...)
           #:val  (let ([a (top-val a)] ...)
                    body ...))))]))
```

```
(define-syntax (define-traced-primitive stx)
  (syntax-case stx ()
    [(_ (f a ...) f-name
        body ...)
     #'(define (f a ...)
         (trace-add
          (trace-append a ...)
          (make-assignment
           #:expr (list 'app f-name (top-id a) ...)
           #:val  (let ([a (top-val a)] ...)
                    body ...)))))]))
```

trace-lang functions

```
(define-traced-primitive (+& a b) '+
  (+ a b))
(define-traced-primitive (*& a b) '*
  (* a b))
; ...
(define-traced-primitive (<& a b) '<
  (< a b))
; ...
(define-traced-primitive (cons& a b) 'cons
  (cons a b))
; ...
```

```
#lang racket
; ...
(provide (rename-out [+& +]
                     [*& *]
                     [exp& exp]
                     ...))
; ...
```

rename-out

- Useful for modifying behaviour of an existing language
- Can refer to the original binding in the defining module
- External interface has the new binding

Interposition points

```
(+ 1 2)
=> (#%app + 1 2)
=> (#%app + (#%datum . 1) (#%datum . 2))
```

Interposition points

`#%app`

`#%datum`

`#%module-begin`

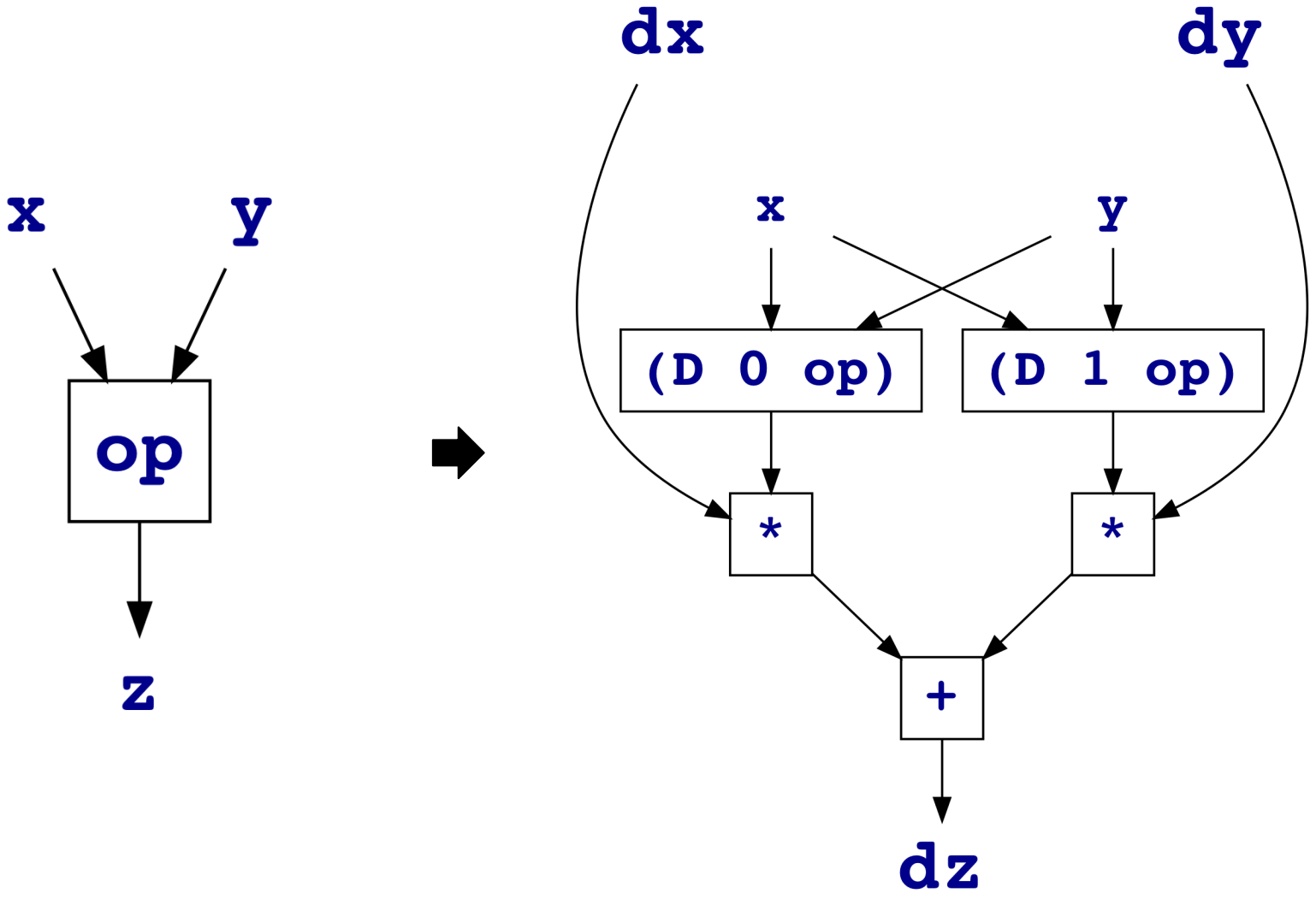
`#%top`

`#%top-interaction`

```
(#%datum . 1)
=> (make-trace (make-assignment #:val 1))
=> %1 | (constant 1) | 1
```

try it!

Recap: Forward-mode AD



Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x           (top-id (list-ref xs i))]
        [indep-ids   (map top-id xs)]
        [result      (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))}))))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
       (let ([dz (D/f-prim-op z x indep-ids
                              tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(for/fold ([sum 0]
           [prod 1])
          ([x (range 1 6)])
  (values (+ x sum)
          (* x prod)))
=>
15
120
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))})))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          {values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))}))))
    (trace-prune Dresult)))
```

Forward-mode AD

```
(define ((D/f i f) . xs)
  (let ([x          (top-id (list-ref xs i))]
        [indep-ids  (map top-id xs)]
        [result     (apply f xs)])
    (define-values (Dresult _)
      (for/fold ([tr result]
                 [deriv-dict (hash)])
                ([z (reverse (trace-items result))])
        (let ([dz (D/f-prim-op z x indep-ids
                               tr deriv-dict)])
          (values
           (trace-append dz tr)
           (hash-set deriv-dict
                     (id z) (top-id dz))))))
    (trace-prune Dresult)))
```

```
; D/f-prim-op: assignment? symbol? (Listof symbol?)
;   trace? (HashTable symbol? symbol?) -> trace?
(define (D/f-prim-op z x-symb indep-ids
                     tr deriv-dict)

  ; I : symbol? -> trace?
  (define (I s) (trace-get s tr))

  ; d : symbol? -> trace?
  (define (d s) (I (hash-ref deriv-dict s)))

  (cond
    ; ...
    ))
```

```
; ...
(cond
  [(eq? (id z) x-symb) (datum . 1.0)]
  [(memq (id z) indep-ids) (datum . 0.0)]
  [else
   (match (expr z)
     ; ...
     )])
```

```
; ...
(match (expr z)
  [(list 'constant '())   (datum . null)]
  [(list 'constant c)     (datum . 0.0)]
  ; ...
  )
```

```
; ...
(match (expr z)
  ; ...
  [(list 'app op xs ...)
   (let ([xs& (map I xs)])
     (for/fold ([acc (datum . 0.0)])
               ([x xs]
                [i (in-naturals)])
       (define D_i_op (apply partial i op xs&))
       (+& (*& D_i_op (d x)) acc)))])
```

```
; ...
(match (expr z)
  ; ...
  [(list 'app 'cons x y) (cons& (d x) (d y))]
  [(list 'app 'car ls)   (car&  (d ls))]
  [(list 'app 'cdr ls)   (cdr&  (d ls))]
  ; ...
  )
```

```
(cons 'a 'b) => (a . b)
(cons 'a null) => (a)
(cons 'a (cons 'b null)) => (a b)
(list 'a 'b) => (a b)
```

```
(cons 'a 'b) => (a . b)
(car (cons 'a 'b)) => 'a
(cdr (cons 'a 'b)) => 'b
```

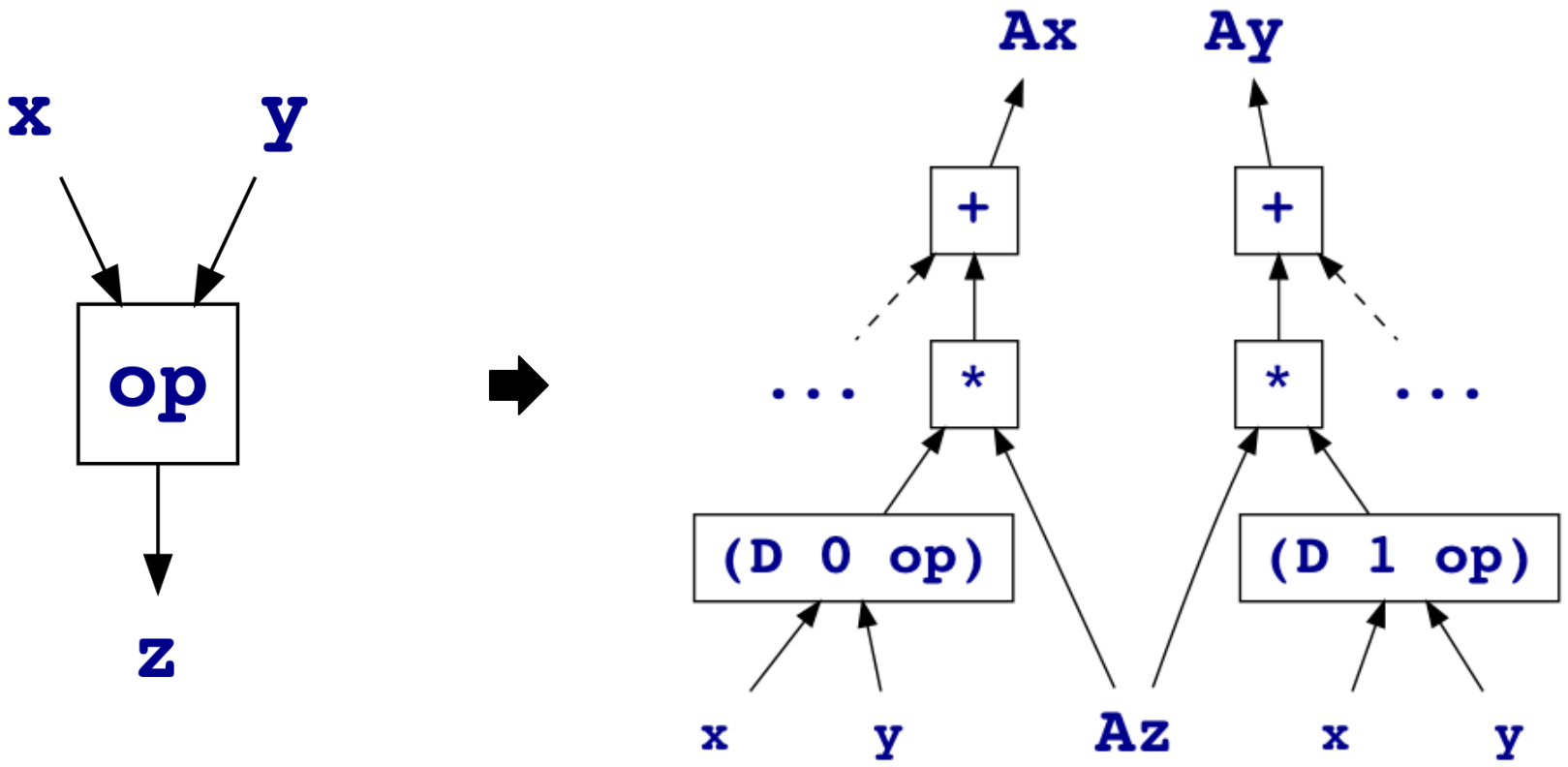
```
((D cons) (f x) (g y))
= (cons ((D f) x) ((D g) y))
```

```
((D car) (cons (f x) (g y)))
= ((D f) x)
```

```
((D cdr) (cons (f x) (g y)))
= ((D g) y)
```

try it!

Recap: Reverse-mode AD



```
(define (A/r result-tr indep-ids s)
  (define seed-id (top-id result-tr))
  (define seed-tr (trace-append s result-tr))

  (define-values (tr _ adjoints)
    (for/fold ([tr seed-tr]
               [adjoint-terms
                (hash seed-id
                      (list (top-id seed-tr)))]
               [adjoints (hash)])
              ([w (trace-items result-tr)])

      ; ...
      ))
  ; ...
  )
```

```
(for/fold ([sum 0]
           [prod 1])
          ([x (range 1 6)])
  (values (+ x sum)
          (* x prod)))
```

=>

```
15
120
```

```
(define (A/r result-tr indep-ids s)
  (define seed-id (top-id result-tr))
  (define seed-tr (trace-append s result-tr))

  (define-values (tr _ adjoints)
    (for/fold ([tr seed-tr]
               [adjoint-terms
                (hash seed-id
                      (list (top-id seed-tr)))]
               [adjoints (hash)])
              ([w (trace-items result-tr)])

      ; ...
      ))
  ; ...
  )
```

```
(define (A/r result-tr indep-ids s)
  (define seed-id (top-id result-tr))
  (define seed-tr (trace-append s result-tr))

  (define-values (tr _ adjoints)
    (for/fold ([tr seed-tr]
               [adjoint-terms
                (hash seed-id
                      (list (top-id seed-tr)))]
               [adjoints (hash)])
              ([w (trace-items result-tr)])

      ; ...
      ))
  ; ...
  )
```

```
(for/fold (...)
          ([w (trace-items result-tr)])
  (define Aw-terms
    (for/list ([k (hash-ref adjoint-terms (id w))])
      (trace-get k tr)))
  (define Aw
    (trace-append
     (foldl cons-add (car Aw-terms) (cdr Aw-terms))
     tr))
  (define-values (tr* adjoint-terms*)
    (A/r-prim-op w Aw adjoint-terms))
  {values tr*
          adjoint-terms*
          (hash-set adjoints (id w) (top-id Aw))})
```

```
(for/fold (...)
          ([w (trace-items result-tr)])
  (define Aw-terms
    (for/list ([k (hash-ref adjoint-terms (id w))])
      (trace-get k tr)))
  (define Aw
    (trace-append
     (foldl cons-add (car Aw-terms) (cdr Aw-terms))
     tr))
  (define-values (tr* adjoint-terms*)
    (A/r-prim-op w Aw adjoint-terms))
  {values tr*
          adjoint-terms*
          (hash-set adjoints (id w) (top-id Aw))})
```

```
(for/fold (...)
          ([w (trace-items result-tr)])
  (define Aw-terms
    (for/list ([k (hash-ref adjoint-terms (id w))])
      (trace-get k tr)))
  (define Aw
    (trace-append
     (foldl cons-add (car Aw-terms) (cdr Aw-terms))
     tr))
  (define-values (tr* adjoint-terms*)
    (A/r-prim-op w Aw adjoint-terms))
  {values tr*
          adjoint-terms*
          (hash-set adjoints (id w) (top-id Aw))})
```

```
(for/fold (...)
          ([w (trace-items result-tr)])
  (define Aw-terms
    (for/list ([k (hash-ref adjoint-terms (id w))])
      (trace-get k tr)))
  (define Aw
    (trace-append
     (foldl cons-add (car Aw-terms) (cdr Aw-terms))
     tr))
  (define-values (tr* adjoint-terms*)
    (A/r-prim-op w Aw adjoint-terms))
  {values tr*
          adjoint-terms*
          (hash-set adjoints (id w) (top-id Aw))})
```

```
(for/fold (...)
          ([w (trace-items result-tr)])
  (define Aw-terms
    (for/list ([k (hash-ref adjoint-terms (id w))])
      (trace-get k tr)))
  (define Aw
    (trace-append
     (foldl cons-add (car Aw-terms) (cdr Aw-terms))
     tr))
  (define-values (tr* adjoint-terms*)
    (A/r-prim-op w Aw adjoint-terms))
  {values tr*
          adjoint-terms*
          (hash-set adjoints (id w) (top-id Aw))})
```

```
; ...
(let* ([tr* (trace-add
             tr
             (make-assignment #:val 0.0))]
       [zero-id (top-id tr*)])
  (trace-prune
   (apply
    list&
    (for/list ([x indep-ids])
      (trace-get
       (hash-ref adjoints x zero-id)
       tr*)))))
```

```
w ← (cons x y)
```

=>

```
Ax ← (car Aw)
Ay ← (cdr Aw)
```

```
w ← (car xs)
```

=>

```
(car Axs) ← Aw
```

```
w ← (cdr xs)
```

=>

```
(cdr Axs) ← Aw
```

```
w ← (car xs)
```

=>

```
Axs ← (cons Aw (cons-zero (cdr xs)))
```

```
w ← (cdr xs)
```

=>

```
Axs ← (cons (cons-zero (car xs)) Aw)
```

try it!

Program transformation

Can apply the previous work to straight-line code, at compile time

`define` instead of **`assignment`**

Program transformation

```
#lang rackpropagator/↙
  straightline
(define (f x y)
  (define a (+ x y))
  (define b (+ a a))
  (define c (* a y))
  (define d 1.0)
  (+ c d))
```

```
(define (Df x y)
  (define a (+ x y))
  (define %2 1.0)
  (define %3 1.0)
  (define %4 (* %2 %3))
  (define %7 (* %4 y))
  (define %8 (* %4 a))
  (define %9 1.0)
  (define %10 (* %7 %9))
  (define %11 1.0)
  (define %12 (* %7 %11))
  (define %17 (+ %8 %12))
  (define %19 '())
  (define %20 (cons %17 %19))
  (cons %10 %20))
```

Program transformation

```
(define-syntax (define/d stx)
  (syntax-case stx ()
    [(_ (f args ...) body ...)
     (with-syntax
       ([(body* ...)
         (handle-assignments #'(args ...)
                             #'(body ...))])
       #'(define (f args ...)
           body* ...))]))
```

Program transformation

```
(define-syntax (define/d stx)
  (syntax-case stx ()
    [(_ (f args ...) body ...)
     (with-syntax
       ([(body* ...)
         (handle-assignments #'(args ...)
                             #'(body ...))])
       #'(define (f args ...)
           body* ...))]))

  (provide (rename-out [define/d define]))
```

Dual numbers

Dual numbers

sum-of-squares:

Given **a** and **b**

```
c ← (* a a)
d ← (* b b)
e ← (+ c d)
```

Dual numbers

sum-of-squares:

Given `a` and `b`

```
c ← (* a a)
d ← (* b b)
e ← (+ c d)
```

The "forward-mode" transformation:

```
dc ← (+ (* a da) (* da a))
dd ← (+ (* b db) (* db b))
de ← (+ dc dd)
```

Dual numbers

sum-of-squares:

Can interleve the operations computing *x* and **d***x*

```
c ← (* a a)
dc ← (+ (* a da) (* da a))
d ← (* b b)
dd ← (+ (* b db) (* db b))
e ← (+ c d)
de ← (+ dc dd)
```

- **dx** depends on **dy** if and only if **x** depends on **y**
- **dx** depends on **y** only if **x** depends on **y**

Dual numbers

Idea: treat the pair of `x` and `dx` as a single entity. Define combined operations.

Dual numbers

```
(struct dual-number (p d) #:transparent)
```

Dual numbers

```
(struct dual-number (p d) #:transparent)

(define (primal x)
  (cond
    [(dual-number? x) (dual-number-p x)]
    [(number? x) x]
    [else (raise-argument-error
           'primal "number? or dual-number?" x)]))
```

Dual numbers

```
    (struct dual-number (p d) #:transparent)

(define (dual x)
  (cond
    [(dual-number? x) (dual-number-d x)]
    [(number? x) (zero x)]
    [else (raise-argument-error
           'dual "number? or dual-number?" x)]))
```

Dual numbers

```
(define (dual-+ x y)
  (if (or (dual-number? x) (dual-number? y))
      (dual-number (+ (primal x) (primal y))
                   (+ (dual x) (dual y)))
      (+ x y)))
```

Dual numbers

```
(define (dual-+ x y)
  (if (or (dual-number? x) (dual-number? y))
      (dual-number (+ (primal x) (primal y))
                   (+ (dual x) (dual y)))
      (+ x y)))
```

Dual numbers

```
(define (dual-* x y)
  (if (or (dual-number? x) (dual-number? y))
      (dual-number (* (primal x) (primal y))
                   (+ (* (dual x) (primal y))
                      (* (primal x) (dual y))))
      (* x y)))
```

Dual numbers

```
(define (dual-* x y)
  (if (or (dual-number? x) (dual-number? y))
      (dual-number (* (primal x) (primal y))
                   (+ (* (dual x) (primal y))
                      (* (primal x) (dual y))))
      (* x y)))
```

Dual numbers

```
; ...
(define (dual-log x)
  (if (dual-number? x)
      (dual-number (log (primal x))
                   (/ (dual x) (primal x)))
      (log x)))
; ...
```

Dual numbers

- **only** need to define the primitive numerical functions
- Can be implemented with operator overloading
- A **local** program transformation

Dual numbers: Differentiation

```
(define ((D n f) . args)
  (let ([args* (for/list [(i (in-naturals))
                          (a args)]
                 (if (= i n)
                     (dual-number a 1)
                     (dual-number a 0)))])
    (get-dual-part (apply f args*))))
```

Dual numbers: Differentiation

```
(define ((D n f) . args)
  (let ([args* (for/list [(i (in-naturals))
                          (a args)]
                 (if (= i n)
                     (dual-number a 1)
                     (dual-number a 0)))])
    (get-dual-part (apply f args*))))
```

Dual numbers: Differentiation

```
(define ((D n f) . args)
  (let ([args* (for/list [(i (in-naturals))
                          (a args)]
                 (if (= i n)
                     (dual-number a 1)
                     (dual-number a 0)))])
    (get-dual-part (apply f args*))))
```

Dual numbers: Differentiation

```
(define ((D n f) . args)
  (let ([args* (for/list [(i (in-naturals))
                          (a args)]
                 (if (= i n)
                     (dual-number a 1)
                     (dual-number a 0)))])
    (get-dual-part (apply f args*))))
```

Dual numbers: Differentiation

```
(define ((D n f) . args)
  (let ([args* (for/list [(i (in-naturals))
                          (a args)]
                 (if (= i n)
                     (dual-number a 1)
                     (dual-number a 0)))])
    (get-dual-part (apply f args*))))
```

Dual numbers: Differentiation

```
(define ((D n f) . args)
  (let ([args* (for/list [(i (in-naturals))
                          (a args)]
                 (if (= i n)
                     (dual-number a 1)
                     (dual-number a 0)))])
    (get-dual-part (apply f args*))))
```

Helper function:

```
(get-dual-part
 (list (dual-number 0.0 1.0)
       2.0
       (cons (dual-number 3.0 0.0)
             (dual-number 4.0 5.0))))
```

=> `(1.0 0.0 (0.0 . 5.0))`

try it!

http://github.com/ots22/rackpropagator

References

TALK

From automatic differentiation to message passing

`https://youtu.be/NkJNcEed2NU`

Tom Minka

PAPER

The simple essence of automatic differentiation

`https://arxiv.org/abs/1804.00746`

Conal Elliot (2018)

TALK

The simple essence of automatic differentiation

`https://youtu.be/Sh13MtWGu18`

Conal Elliot

References

PAPER

Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator

https://www.bcl.hamilton.ie/~barak/papers/toplas-reverse.pdf

doi:10.1145/1330017.1330018

Pearlmutter & Siskind (2008)

PAPER

Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator

https://arxiv.org/abs/1803.10228

Fei Wang *et al.* (2018)

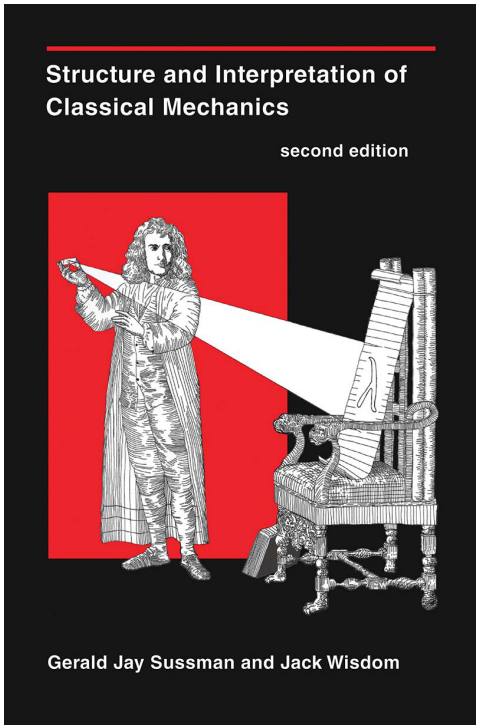
References

BOOK

Structure and Interpretation of Classical Mechanics (2nd ed.)

https://mitpress.mit.edu/sites/default/files/titles/content/sicm_edition_2/book.html

Gerald Jay Sussman & Jack Wisdom (2015)



References

WEBSITE

autodiff.org: Community Portal for Automatic Differentiation
`http://www.autodiff.org/`

BOOK

Beautiful Racket: an introduction to language-oriented programming using Racket, v1.6
`https://beautifulracket.com/`
Matthew Butterick