The simple essence of automatic differentiation

Conal Elliott

Target

January/June 2018

Differentiable programming made easy

Current AI revolution runs on large data, speed, and AD, but

- AD algorithm (backprop) is complex and stateful.
- Graph APIs are complex and semantically dubious.

Solutions in this paper:

- AD: Simple, calculated, efficient, parallel-friendly, generalized.
- API: *derivative*.

What's a derivative?

- Number
- Vector
- Covector
- Matrix
- Higher derivatives

Chain rule for each.

What's a derivative?

$$\mathcal{D} :: (a \to b) \to (a \to (a \multimap b))$$

A local linear (affine) approximation:

$$\lim_{\varepsilon \to 0} \frac{\| f\ (a + \varepsilon) - (f\ a + \mathcal{D}\ f\ a\ \varepsilon) \|}{\| \varepsilon \|} = 0$$

See *Calculus on Manifolds* by Michael Spivak.

Composition

Sequential:

$$
\begin{array}{l}
(\circ) :: (b \to c) \to (a \to b) \to (a \to c) \\
(g \circ f)\ a = g\ (f\ a)
\end{array}
$$

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a \quad \text{-- chain rule}$$

Parallel:

$$
\begin{array}{l}
(\vartriangle) :: (a \to c) \to (a \to d) \to (a \to c \times d) \\
(f \vartriangle g)\ a = (f\ a, g\ a)
\end{array}
$$

$$\mathcal{D}\ (f \vartriangle g)\ a = \mathcal{D}\ f\ a \vartriangle \mathcal{D}\ g\ a$$

Linear functions

Linear functions are their own derivatives everywhere.

$$\begin{aligned}
\mathcal{D}\ \mathit{id}\ \ \ a &= \mathit{id} \\
\mathcal{D}\ \mathit{fst}\ \ a &= \mathit{fst} \\
\mathcal{D}\ \mathit{snd}\ a &= \mathit{snd} \\
&\ldots
\end{aligned}$$

Compositionality

Chain rule:

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ \ a) \circ \mathcal{D}\ f\ \ a \quad \text{-- non-compositional}$$

To fix, combine regular result with derivative:

$$\hat{\mathcal{D}} :: (a \to b) \to (a \to (b \times (a \multimap b)))$$
$$\hat{\mathcal{D}}\ f = f \vartriangle \mathcal{D}\ f \quad \text{-- specification}$$

Often much work in common to f and $\mathcal{D}\ f$.

Abstract algebra for functions

$$\begin{array}{l}
\textbf{class}\ \mathit{Category}\ (\leadsto)\ \textbf{where} \\
\quad \mathit{id}\ \ :: a \leadsto a \\
\quad (\circ) :: (b \leadsto c) \rightarrow (a \leadsto b) \rightarrow (a \leadsto c)
\end{array}$$

$$\begin{array}{l}
\textbf{class}\ \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ (\leadsto)\ \textbf{where} \\
\quad \mathit{exl}\ :: (a \times b) \leadsto a \\
\quad \mathit{exr} :: (a \times b) \leadsto b \\
\quad (\vartriangle)\ :: (a \leadsto c) \rightarrow (a \leadsto d) \rightarrow (a \leadsto (c \times d))
\end{array}$$

Plus laws and classes for arithmetic etc.

Automatic differentiation

$$\textbf{newtype}\ D\ a\ b = D\ (a \rightarrow b \times (a \multimap b))$$

$$\hat{\mathcal{D}} :: (a \rightarrow b) \rightarrow D\ a\ b$$
$$\hat{\mathcal{D}}\ f = D\ (f \vartriangle \mathcal{D}\ f) \quad \text{-- not computable}$$

Specification: $\hat{\mathcal{D}}$ preserves *Category* and *Cartesian* structure:

$$\hat{\mathcal{D}}\ id = id$$
$$\hat{\mathcal{D}}\ (g \circ f) = \hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f$$

$$\hat{\mathcal{D}}\ exl = exl$$
$$\hat{\mathcal{D}}\ exr = exr$$
$$\hat{\mathcal{D}}\ (f \vartriangle g) = \hat{\mathcal{D}}\ f \vartriangle \hat{\mathcal{D}}\ g$$

The game: solve these equations for the RHS operations.

Solution: simple automatic differentiation

$$\mathbf{newtype}\ D\ a\ b = D\ (a \to b \times (a \multimap b))$$

$$linearD\ f = D\ (\lambda a \to (f\ a, f))$$

$$\begin{array}{l}
\mathbf{instance}\ Category\ D\ \mathbf{where} \\
\quad id = linearD\ id \\
\quad D\ g \circ D\ f = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \mathbf{in}\ (c, g' \circ f'))
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ Cartesian\ D\ \mathbf{where} \\
\quad exl\ = linearD\ exl \\
\quad exr = linearD\ exr \\
\quad D\ f \vartriangle D\ g = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \mathbf{in}\ ((b, c), f' \vartriangle g'))
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ NumCat\ D\ \mathbf{where} \\
\quad negate = linearD\ negate \\
\quad add = linearD\ add \\
\quad mul = D\ (mul \vartriangle (\lambda(a, b) \to \lambda(da, db) \to b * da + a * db))
\end{array}$$

Running examples

$$\begin{aligned} &\mathit{sqr} :: \mathit{Num}\ a \Rightarrow a \to a \\ &\mathit{sqr}\ a = a * a \end{aligned}$$

$$\begin{aligned} &\mathit{magSqr} :: \mathit{Num}\ a \Rightarrow a \times a \to a \\ &\mathit{magSqr}\ (a, b) = \mathit{sqr}\ a + \mathit{sqr}\ b \end{aligned}$$

$$\begin{aligned} &\mathit{cosSinProd} :: \mathit{Floating}\ a \Rightarrow a \times a \to a \times a \\ &\mathit{cosSinProd}\ (x, y) = (\mathit{cos}\ z, \mathit{sin}\ z)\ \mathbf{where}\ z = x * y \end{aligned}$$

In categorical vocabulary:

$$\mathit{sqr} = \mathit{mul} \circ (\mathit{id} \vartriangle \mathit{id})$$

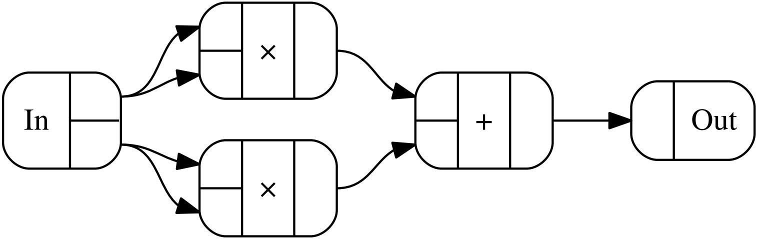
$$\mathit{magSqr} = \mathit{add} \circ (\mathit{mul} \circ (\mathit{exl} \vartriangle \mathit{exl}) \vartriangle \mathit{mul} \circ (\mathit{exr} \vartriangle \mathit{exr}))$$

$$\mathit{cosSinProd} = (\mathit{cos} \vartriangle \mathit{sin}) \circ \mathit{mul}$$

Visualizing computations

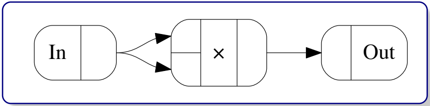
$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$magSqr = add \circ (mul \circ (exl \vartriangle exl) \vartriangle mul \circ (exr \vartriangle exr))$$



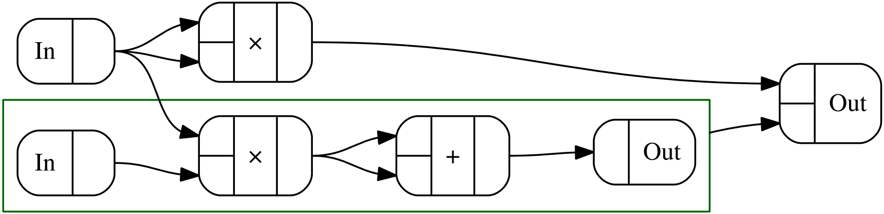
Auto-generated from Haskell code. See *Compiling to categories*.

AD example

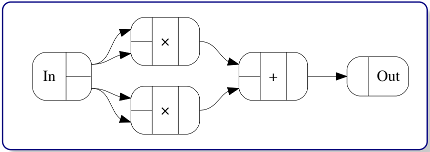


$$sqr\ a = a * a$$

$$sqr = mul \circ (id \vartriangle id)$$

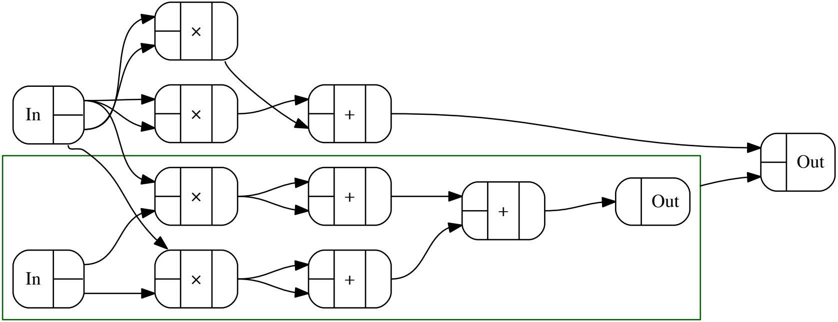


AD example

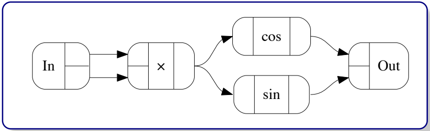


$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$magSqr = add \circ (mul \circ (exl \vartriangle exl) \vartriangle mul \circ (exr \vartriangle exr))$$

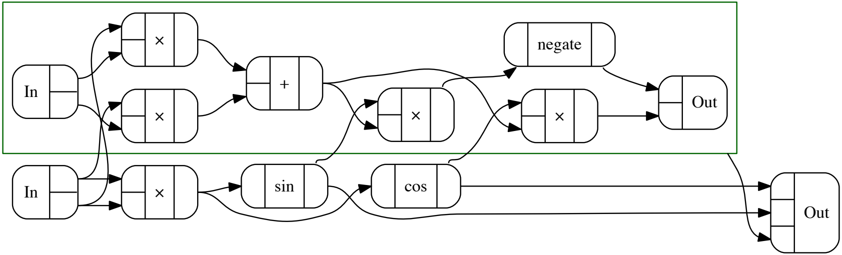


AD example



$$cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \mathbf{where}\ z = x * y$$

$$cosSinProd = (cos \vartriangle sin) \circ mul$$



Generalizing AD

$\mathbf{newtype}\ D\ a\ b = D\ (a \to b \times (a \multimap b))$

$linearD\ f = D\ (\lambda a \to (f\ a, f))$

$\mathbf{instance}\ Category\ D\ \mathbf{where}$
$\quad id = linearD\ id$
$\quad D\ g \circ D\ f = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \mathbf{in}\ (c, g' \circ f'))$

$\mathbf{instance}\ Cartesian\ D\ \mathbf{where}$
$\quad exl = linearD\ exl$
$\quad exr = linearD\ exr$
$\quad D\ f \vartriangle D\ g = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \mathbf{in}\ ((b, c), f' \vartriangle g'))$

Each D operation just uses corresponding $(\multimap)$ operation.

Generalize from $(\multimap)$ to other cartesian categories.

Generalized AD

$$\mathbf{newtype}\ D_{(\leadsto)}\ a\ b = D\ (a \to b \times (a \leadsto b))$$

$$\mathit{linearD}\ f\ f' = D\ (\lambda a \to (f\ \ a, f'))$$

$$\begin{array}{l}
\mathbf{instance}\ \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Category}\ D_{(\leadsto)}\ \mathbf{where} \\
\quad \mathit{id} = \mathit{linearD}\ \mathit{id}\ \mathit{id} \\
\quad D\ g \circ D\ f = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ \ a; (c, g') = g\ \ b\}\ \mathbf{in}\ (c, g' \circ f'))
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ \mathit{Cartesian}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ D_{(\leadsto)}\ \mathbf{where} \\
\quad \mathit{exl}\ = \mathit{linearD}\ \mathit{exl}\ \mathit{exl} \\
\quad \mathit{exr} = \mathit{linearD}\ \mathit{exr}\ \mathit{exr} \\
\quad D\ f \vartriangle D\ g = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ \ a; (c, g') = g\ \ a\}\ \mathbf{in}\ ((b, c), f' \vartriangle g'))
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ \ldots \Rightarrow \mathit{NumCat}\ D\ \mathbf{where} \\
\quad \mathit{negate} = \mathit{linearD}\ \mathit{negate}\ \mathit{negate} \\
\quad \mathit{add}\ = \mathit{linearD}\ \mathit{add}\ \mathit{add} \\
\quad \mathit{mul}\ = ??
\end{array}$$

Numeric operations

Specific to (linear) *functions*:

$$mul = D\ (mul \vartriangle (\lambda(a,b) \rightarrow \lambda(da,db) \rightarrow b * da + a * db))$$

Rephrase:

$$\begin{aligned}&scale :: Multiplicative\ a \Rightarrow a \rightarrow (a \multimap a)\\&scale\ u = \lambda v \rightarrow u * v\end{aligned}$$

$$\begin{aligned}&(\triangledown) :: (a \multimap c) \rightarrow (b \multimap c) \rightarrow ((a \times b) \multimap c)\\&f \triangledown g = \lambda(a,b) \rightarrow f\ a + g\ b\end{aligned}$$

Now

$$mul = D\ (mul \vartriangle (\lambda(a,b) \rightarrow scale\ b \triangledown scale\ a))$$

Linear arrow (biproduct) vocabulary

```
class Category (⇝) where
  id  :: a ⇝ a
  (∘) :: (b ⇝ c) → (a ⇝ b) → (a ⇝ c)

class Category (⇝) ⇒ Cartesian (⇝) where
  exl :: (a × b) ⇝ a
  exr :: (a × b) ⇝ b
  (△) :: (a ⇝ c) → (a ⇝ d) → (a ⇝ (c × d))

class Category (⇝) ⇒ Cocartesian (⇝) where
  inl :: a ⇝ (a × b)
  inr :: b ⇝ (a × b)
  (▽) :: (a ⇝ c) → (b ⇝ c) → ((a × b) ⇝ c)

class ScalarCat (⇝) a where
  scale :: a → (a ⇝ a)
```

Linear transformations as functions

$$\textbf{newtype}\ a \rightarrow^{+} b = \mathit{AddFun}\ (a \rightarrow b)$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Category}\ (\rightarrow^{+})\ \textbf{where} \\
\quad \mathit{id}\ \ = \mathit{AddFun}\ \mathit{id} \\
\quad (\circ) = \mathit{inNew}_2\ (\circ)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Cartesian}\ (\rightarrow^{+})\ \textbf{where} \\
\quad \mathit{exl}\ = \mathit{AddFun}\ \mathit{exl} \\
\quad \mathit{exr} = \mathit{AddFun}\ \mathit{exr} \\
\quad (\vartriangle)\ = \mathit{inNew}_2\ (\vartriangle)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Cocartesian}\ (\rightarrow^{+})\ \textbf{where} \\
\quad \mathit{inl}\ = \mathit{AddFun}\ (,0) \\
\quad \mathit{inr} = \mathit{AddFun}\ (0,) \\
\quad (\triangledown) = \mathit{inNew}_2\ (\lambda f\ g\ (x, y) \rightarrow f\ x + g\ y)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Multiplicative}\ s \Rightarrow \mathit{ScalarCat}\ (\rightarrow^{+})\ s\ \textbf{where} \\
\quad \mathit{scale}\ s = \mathit{AddFun}\ (s\ *)
\end{array}$$

Extracting a data representation

- How to extract a matrix or gradient vector?
- Sample over a domain *basis* (rows of identity matrix).
- For n-dimensional *domain*,
  - Make n passes.
  - Each pass works on n-D sparse ("one-hot") input.
  - Very inefficient.
- For gradient-based optimization,
  - High-dimensional domain.
  - Very low-dimensional (1-D) codomain.

Generalized matrices

$\textbf{newtype}\ M_s\ a\ b = L\ (V_s\ b\ (V_s\ a\ s))$

$applyL :: M_s\ a\ b \rightarrow (a \rightarrow b)$

Require *applyL* to preserve structure. Solve for methods.

Core vocabulary

Sufficient to build arbitrary "matrices":

$$
\begin{array}{lll}
\mathit{scale} :: a \to (a \leadsto a) & & \text{-- } 1 \times 1 \\
(\triangledown) & :: (a \leadsto c) \to (b \leadsto c) \to ((a \times b) \leadsto c) & \text{-- horizontal juxt} \\
(\vartriangle) & :: (a \leadsto c) \to (a \leadsto d) \to (a \leadsto (c \times d)) & \text{-- vertical juxt}
\end{array}
$$

Types guarantee rectangularity.

Efficiency of composition

- Arrow composition is associative.
- Some associations are more efficient than others, so
  - Associate optimally.
  - Equivalent to *matrix chain multiplication* — $O(n \log n)$.
  - Choice determined by *types*, i.e., compile-time information.
- All-right: “forward mode AD” (FAD).
- All-left: “reverse mode AD” (RAD).
- RAD is *much* better for gradient-based optimization.

Left-associating composition (RAD)

- CPS-like category:
    - Represent $a \leadsto b$ by $(b \leadsto r) \rightarrow (a \leadsto r)$.
    - Meaning: $f \mapsto (\circ\, f)$.
    - Results in left-composition.
    - Initialize with $id :: r \leadsto r$.
    - Construct $h \circ \mathcal{D}\, f\ a$ directly, without $\mathcal{D}\, f\ a$.

Continuation category

$$\textbf{newtype}\ \mathit{Cont}^{r}_{(\leadsto)}\ a\ b = \mathit{Cont}\ ((b \leadsto r) \to (a \leadsto r))$$

$$\begin{aligned}&\mathit{cont} :: \mathit{Category}\ (\leadsto) \Rightarrow (a \leadsto b) \to \mathit{Cont}^{r}_{(\leadsto)}\ a\ b\\ &\mathit{cont}\ f = \mathit{Cont}\ (\circ\, f)\end{aligned}$$

Require *cont* to preserve structure. Solve for methods.

We'll use an isomorphism:

$$\begin{aligned}&\mathit{join} &&:: \mathit{Cocartesian}\ (\leadsto) \Rightarrow (c \leadsto a) \times (d \leadsto a) \to ((c \times d) \leadsto a)\\ &\mathit{unjoin} &&:: \mathit{Cocartesian}\ (\leadsto) \Rightarrow ((c \times d) \leadsto a) \to (c \leadsto a) \times (d \leadsto a)\end{aligned}$$

$$\begin{aligned}&\mathit{join}\ (f, g) &&= f \triangledown g\\ &\mathit{unjoin}\ h &&= (h \circ \mathit{inl}, h \circ \mathit{inr})\end{aligned}$$

Continuation category (solution)

$\textbf{instance}\ \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Category}\ \mathit{Cont}^{r}_{(\leadsto)}\ \textbf{where}$

$\quad \mathit{id} = \mathit{Cont}\ \mathit{id}$

$\quad \mathit{Cont}\ g \circ \mathit{Cont}\ f = \mathit{Cont}\ (f \circ g)$

$\textbf{instance}\ \mathit{Cartesian}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ \mathit{Cont}^{r}_{(\leadsto)}\ \textbf{where}$

$\quad \mathit{exl}\ = \mathit{Cont}\ (\mathit{join} \circ \mathit{inl})$

$\quad \mathit{exr} = \mathit{Cont}\ (\mathit{join} \circ \mathit{inr})$

$\quad (\vartriangle) = \mathit{inNew}_2\ (\lambda f\ g \rightarrow (f \triangledown g) \circ \mathit{unjoin})$

$\textbf{instance}\ \mathit{Cocartesian}\ (\leadsto) \Rightarrow \mathit{Cocartesian}\ \mathit{Cont}^{r}_{(\leadsto)}\ \textbf{where}$

$\quad \mathit{inl}\ = \mathit{Cont}\ (\mathit{exl} \circ \mathit{unjoin})$

$\quad \mathit{inr} = \mathit{Cont}\ (\mathit{exr} \circ \mathit{unjoin})$

$\quad (\triangledown) = \mathit{inNew}_2\ (\lambda f\ g \rightarrow \mathit{join} \circ (f \vartriangle g))$

$\textbf{instance}\ \mathit{ScalarCat}\ (\leadsto)\ a \Rightarrow \mathit{ScalarCat}\ \mathit{Cont}^{r}_{(\leadsto)}\ a\ \textbf{where}$

$\quad \mathit{scale}\ s = \mathit{Cont}\ (\mathit{scale}\ s)$

Reverse-mode AD without tears

$$\mathcal{D}_{Cont^{r}_{M_s}}$$

Left-associating composition (RAD)

- CPS-like category:
  - Represent $a \leadsto b$ by $(b \leadsto r) \to (a \leadsto r)$.
  - Meaning: $f \mapsto (\circ\, f)$.
  - Results in left-composition.
  - Initialize with $\mathit{id} :: r \leadsto r$.
  - Construct $h \circ \mathcal{D}\ f\ \ a$ directly, without $\mathcal{D}\ f\ \ a$.
- We've seen this trick before:
  - Transforming naive *reverse* from quadratic to linear.
  - List generalizes to monoids, and monoids to categories.

One of my favorite papers

- *Continuation-Based Program Transformation Strategies*
  Mitch Wand, 1980, JACM.
- Introduce a continuation argument, e.g., $[\,a\,] \to [\,a\,]$.
- Notice the continuations that arise, e.g., $(+\!\!+\ as)$.
- Find a *data* representation, e.g., $as :: [\,a\,]$
- Identify associative operation that represents composition, e.g., $(+\!\!+)$, since $(+\!\!+\ bs) \circ (+\!\!+\ as) = (+\!\!+\ (as +\!\!+ bs))$.

Duality

- Vector space dual: $u \multimap s$, with u a vector space over s.
- If u has finite dimension, then $u \multimap s \cong u$.
- For $f :: u \multimap s$, $f = \mathit{dot}\ v$ for some $v :: u$.
- Gradients are derivatives of functions with scalar codomain.
- Represent $a \multimap b$ by $(b \multimap s) \rightarrow (a \multimap s)$ by $b \rightarrow a$.
- *Ideal* for extracting gradient vector. Just apply to 1 (id).

Duality

$$\textbf{newtype}\ \mathit{Dual}_{(\leadsto)}\ a\ b = \mathit{Dual}\ (b \leadsto a)$$

$$\begin{aligned}&\mathit{asDual} :: \mathit{Cont}^{s}_{(\leadsto)}\ a\ b \to \mathit{Dual}_{(\leadsto)}\ a\ b\\ &\mathit{asDual}\ (\mathit{Cont}\ f) = \mathit{Dual}\ (\mathit{dot}^{-1} \circ f \circ \mathit{dot})\end{aligned}$$

where

$$\begin{aligned}&\mathit{dot} &&:: u \to (u \multimap s)\\ &\mathit{dot}^{-1} &&:: (u \multimap s) \to u\end{aligned}$$

Require *asDual* to preserve structure. Solve for methods.

Duality (solution)

$$\textbf{newtype}\ \mathit{Dual}_{(\leadsto)}\ a\ b = \mathit{Dual}\ (b \leadsto a)$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Category}\ \mathit{Dual}_{(\leadsto)}\ \textbf{where}\\
&\quad \mathit{id}\ \ = \mathit{Dual}\ \mathit{id}\\
&\quad (\circ) = \mathit{inNew}_2\ (\mathit{flip}\ (\circ))
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Cocartesian}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ \mathit{Dual}_{(\leadsto)}\ \textbf{where}\\
&\quad \mathit{exl}\ = \mathit{Dual}\ \mathit{inl}\\
&\quad \mathit{exr} = \mathit{Dual}\ \mathit{inr}\\
&\quad (\vartriangle)\ = \mathit{inNew}_2\ (\triangledown)
\end{aligned}$$

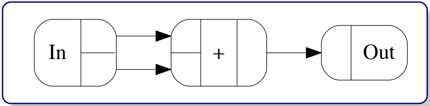
$$\begin{aligned}
&\textbf{instance}\ \mathit{Cartesian}\ (\leadsto) \Rightarrow \mathit{Cocartesian}\ \mathit{Dual}_{(\leadsto)}\ \textbf{where}\\
&\quad \mathit{inl}\ = \mathit{Dual}\ \mathit{exl}\\
&\quad \mathit{inr} = \mathit{Dual}\ \mathit{exr}\\
&\quad (\triangledown)\ = \mathit{inNew}_2\ (\vartriangle)
\end{aligned}$$

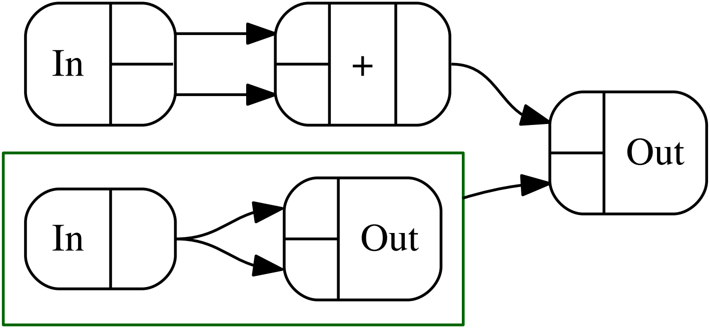
$$\begin{aligned}
&\textbf{instance}\ \mathit{ScalarCat}\ (\leadsto)\ s \Rightarrow \mathit{ScalarCat}\ \mathit{Dual}_{(\leadsto)}\ s\ \textbf{where}\\
&\quad \mathit{scale}\ s = \mathit{Dual}\ (\mathit{scale}\ s)
\end{aligned}$$

Backpropagation

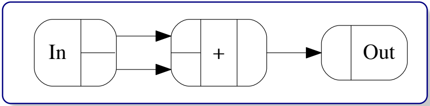
$$D_{Dual_{\rightarrow^{+}}}$$

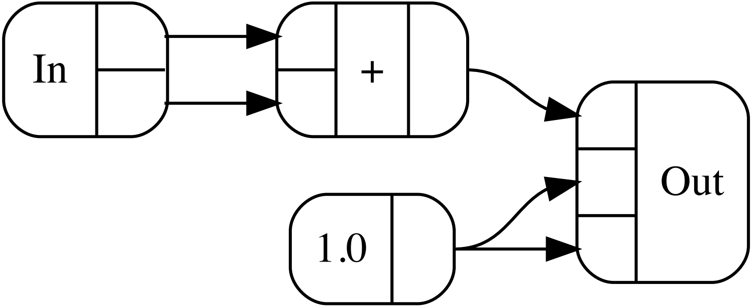
RAD example (dual function)



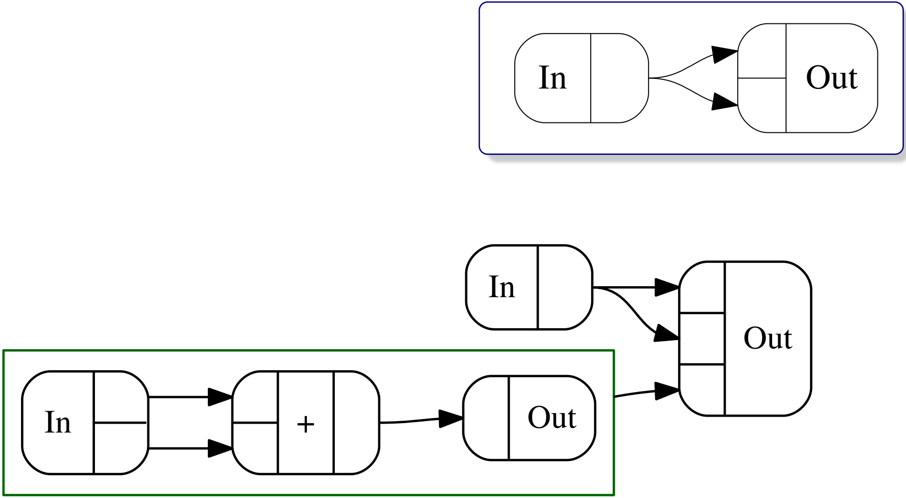


RAD example (dual vector)

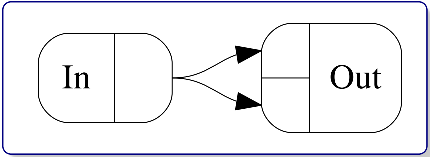


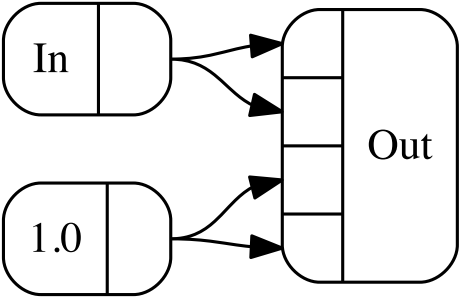


RAD example (dual function)

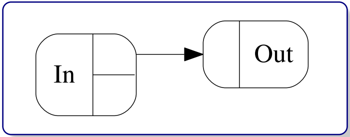


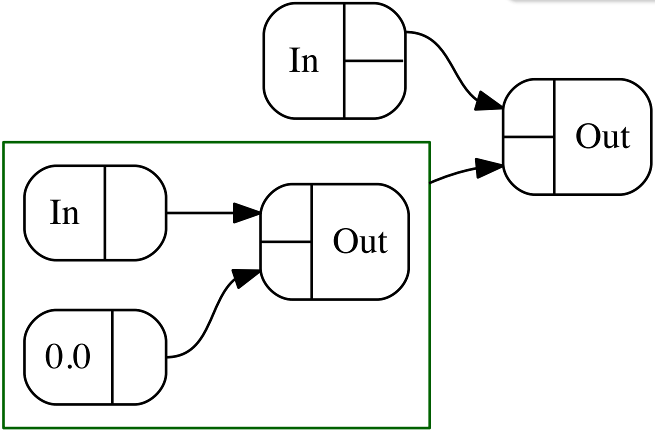
RAD example (vector)



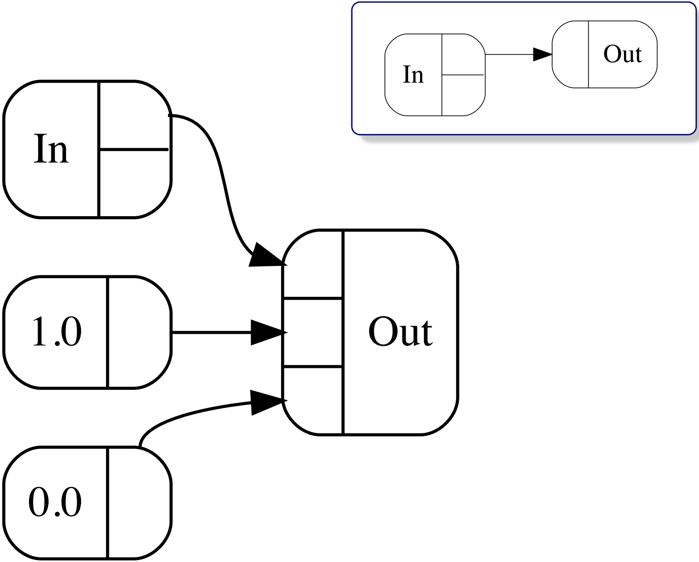


RAD example (dual function)

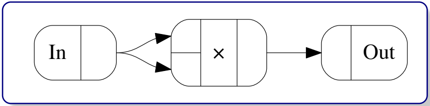


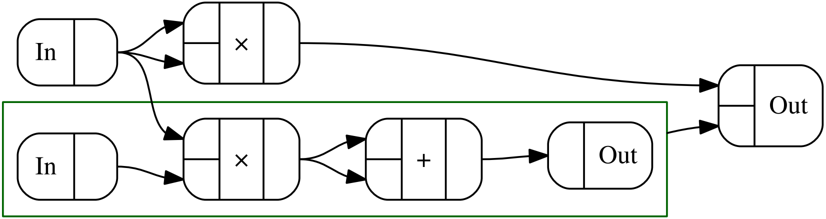


RAD example (dual vector)

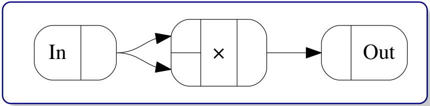


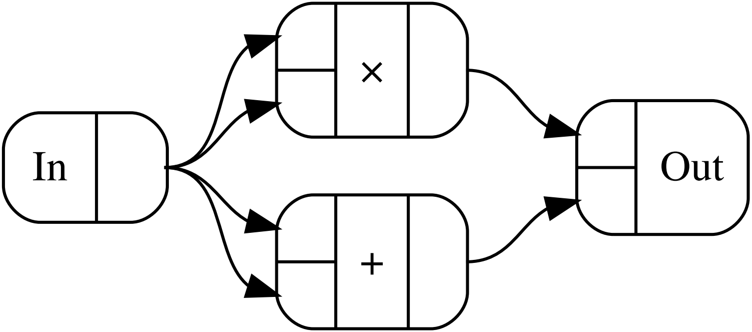
RAD example (dual function)



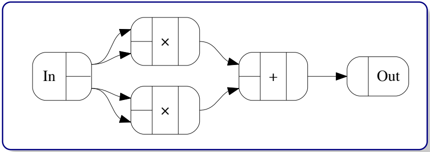


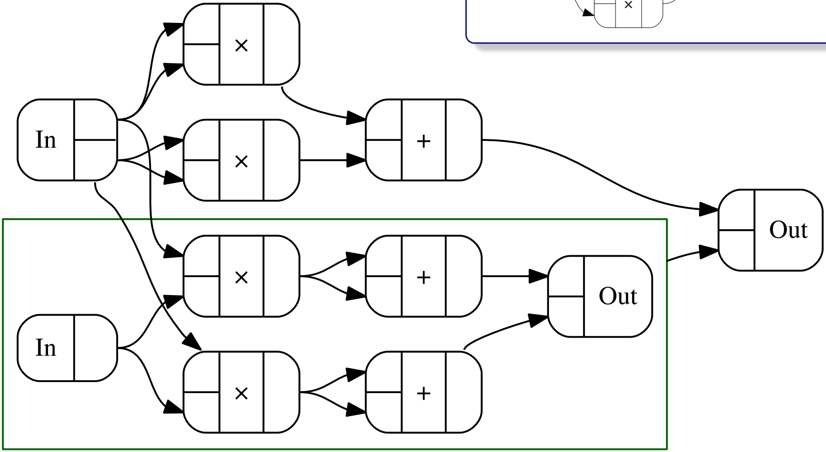
RAD example (dual vector)



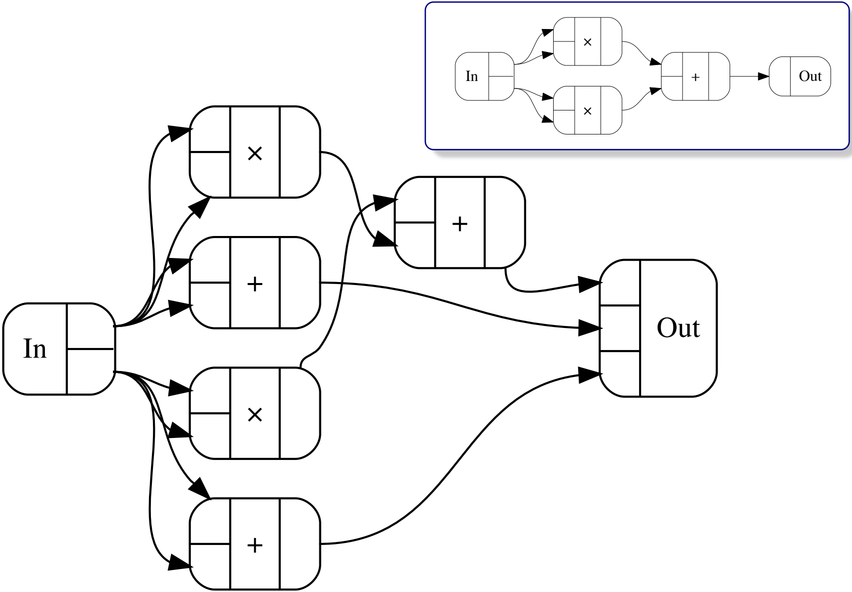


RAD example (dual function)

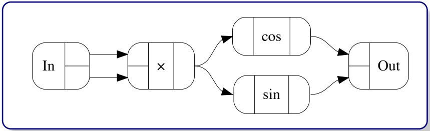


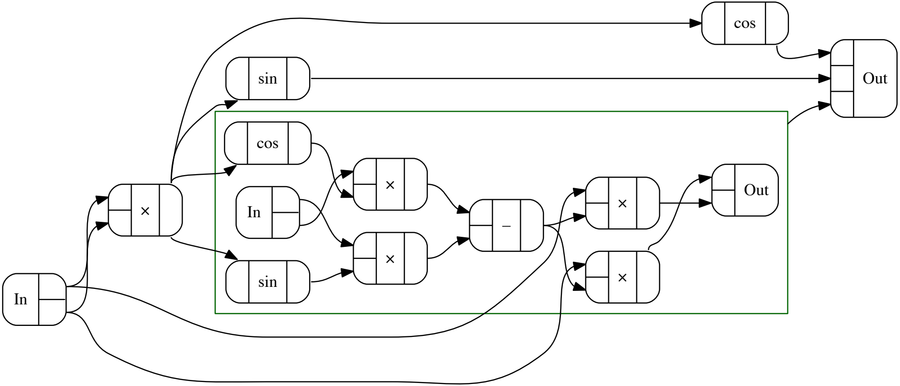


RAD example (dual vector)

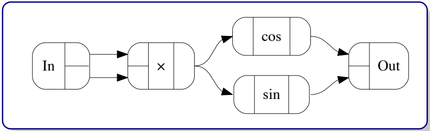


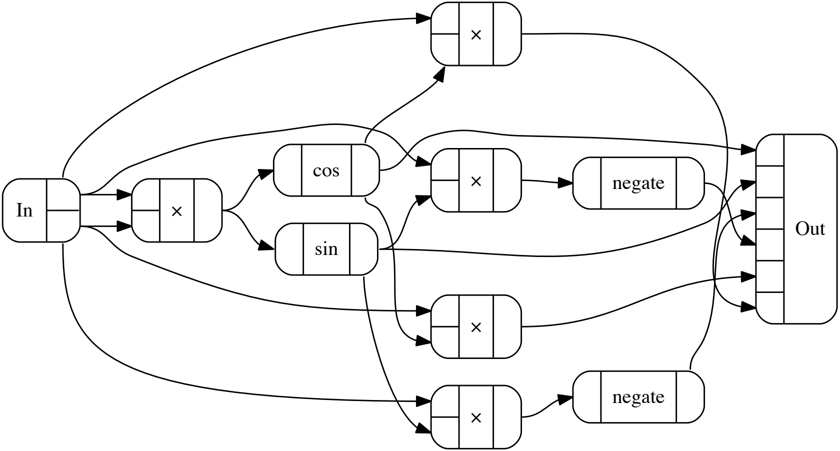
RAD example (dual function)



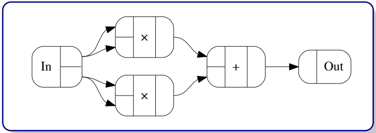


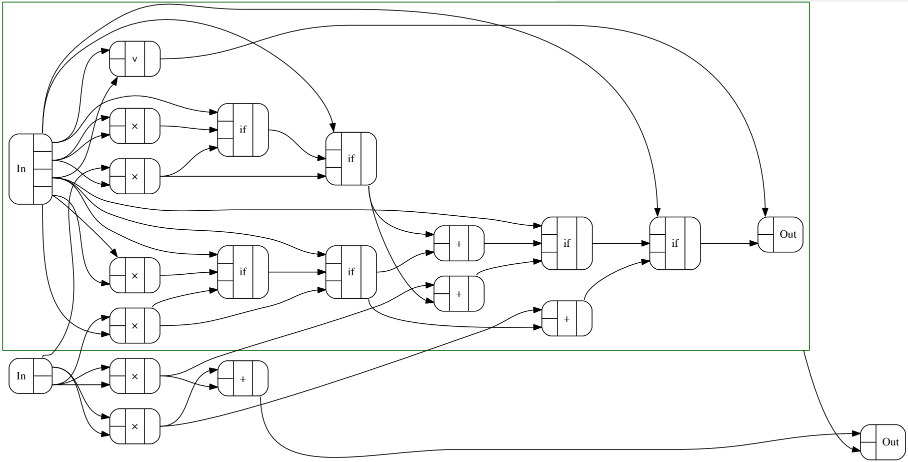
RAD example (matrix)





Incremental evaluation





Conclusions

- Simple AD algorithm, specializing to forward, reverse, mixed.
- No graphs, tapes, tags, partial derivatives, or mutation.
- Parallel-friendly and low memory use.
- Calculated from simple, regular algebra problems.
- Generalizes to derivative categories other than linear maps.
- Differentiate regular Haskell code (via plugin).
- More details in my ICFP 2018 paper.

Reflections: recipe for success

Key principles:

- Capture main concepts as first-class values.
- Focus on abstract notions, not specific representations.
- Calculate efficient implementation from simple specification.

Not previously applied to AD (afaik).

Quandary: Most programming languages poor for function-like things.

Solution: *Compiling to categories*.

Symbolic vs automatic differentiation

Often described as opposing techniques:

- *Symbolic*:
  - Apply differentiation rules symbolically.
  - Can duplicate much work.
  - Needs algebraic manipulation.
- *Automatic*:
  - FAD: easy to implement but often inefficient.
  - RAD: efficient but tricky to implement.

My view: *AD is SD done by a compiler.*

Compilers already work symbolically and preserve sharing.