Efficient automatic differentiation made easy via elementary category theory

Conal Elliott

October 29, 2020

What is differentiation? (Fréchet)

On Banach spaces a and b,

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

$f\ a + \mathcal{D}\ f\ a\ \varepsilon$ approximates $f\ (a + \varepsilon)$ for small ε.

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

See *Calculus on Manifolds* by Michael Spivak.

What is *automatic* differentiation?

Differentiation of computable functions is not computable.

Instead, differentiate *recipes*: $[\![\bar{\mathcal{D}}\ p]\!] = \mathcal{D}\ [\![p]\!]$.

What is *automatic* differentiation?

Differentiation of computable functions is not computable.

Instead, differentiate *recipes*: $[\![\bar{\mathcal{D}}\ p]\!] = \mathcal{D}\ [\![p]\!]$.

Popular recipe forms: graphs, imperative programs, lambda calculus.

Differentiation composes messily in these forms

What is *automatic* differentiation?

Differentiation of computable functions is not computable.

Instead, differentiate *recipes*: $[\![\bar{\mathcal{D}}\ p]\!] = \mathcal{D}\ [\![p]\!]$.

Popular recipe forms: graphs, imperative programs, lambda calculus.

Differentiation composes messily in these forms,

but tidily in language of categories!

Composition

Sequential:

$$(\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$
$$(g \circ f)\ a = g\ (f\ a)$$

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

Composition

Sequential:

$$(\circ) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(g \circ f)\ a = g\ (f\ a)$$

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

Parallel:

$$(\vartriangle) :: (a \to c) \to (a \to d) \to (a \to c \times d)$$
$$(f \vartriangle g)\ a = (f\ a, g\ a)$$

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

Linear functions

Linear functions

Linear functions are their own derivatives everywhere.

$$
\begin{aligned}
\mathcal{D}\; \mathit{id}\;\; a &= \mathit{id} \\
\mathcal{D}\; \mathit{exl}\; a &= \mathit{exl} \\
\mathcal{D}\; \mathit{exr}\; a &= \mathit{exr} \\
&\ldots
\end{aligned}
$$

Compositionality

Chain rule:

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

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:

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

so that $\mathcal{D}\ f = \mathit{exr} \circ \hat{\mathcal{D}}\ f$.

Abstract algebra for functions

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

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

Plus laws and classes for arithmetic etc.

Automatic differentiation (specification)

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

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

Automatic differentiation (specification)

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

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

Specification: $\hat{\mathcal{D}}$ is a cartesian functor, i.e.,

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

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

The game: solve these equations for the RHS operations.

Automatic differentiation (solution)

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

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

$\textbf{instance}\ \mathit{Category}\ D\ \textbf{where}$

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

$\quad D\ g \circ D\ f = D\ (\lambda a \rightarrow \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))$

$\textbf{instance}\ \mathit{Cartesian}\ D\ \textbf{where}$

$\quad \mathit{exl} = \mathit{linearD}\ \mathit{exl}$

$\quad \mathit{exr} = \mathit{linearD}\ \mathit{exr}$

$\quad D\ f \vartriangle D\ g = D\ (\lambda a \rightarrow \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \textbf{in}\ ((b, c), f' \vartriangle g'))$

$\textbf{instance}\ \mathit{NumCat}\ D\ \textbf{where}$

$\quad \mathit{negateC} = \mathit{linearD}\ \mathit{negateC}$

$\quad \mathit{addC} = \mathit{linearD}\ \mathit{addC}$

$\quad \mathit{mulC} = D\ (\mathit{mulC} \vartriangle (\lambda(a, b) \rightarrow \lambda(da, db) \rightarrow b * da + a * db))$

Generalizing AD

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

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

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

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

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

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

Generalized AD

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

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

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

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

$$\begin{array}{l}
\textbf{instance}\ \ldots \Rightarrow \mathit{NumCat}\ D\ \textbf{where} \\
\quad \mathit{negateC} = \mathit{linearD}\ \mathit{negateC}\ \mathit{negateC} \\
\quad \mathit{addC} = \mathit{linearD}\ \mathit{addC}\ \mathit{addC} \\
\quad \mathit{mulC} = ??
\end{array}$$

Numeric operations

Specific to (linear) *functions*:

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

Numeric operations

Specific to (linear) *functions*:

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

Rephrase:

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

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

Now

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

New generalized vocabulary

$$
\begin{array}{l}
\textbf{class}\ \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Cocartesian}_{\times}\ (\leadsto)\ \textbf{where} \\
\quad \mathit{inl}\ :: a \leadsto (a \times b) \\
\quad \mathit{inr} :: b \leadsto (a \times b) \\
\quad (\triangledown) :: (a \leadsto c) \rightarrow (b \leadsto c) \rightarrow ((a \times b) \leadsto c)
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{ScalarCat}\ (\leadsto)\ a\ \textbf{where} \\
\quad \mathit{scale} :: a \rightarrow (a \leadsto a)
\end{array}
$$

Differentiation:

$$\mathcal{D}\ (f \triangledown g)\ (a, b) = \mathcal{D}\ f\ a \triangledown \mathcal{D}\ g\ b$$

The rest are linear.

Linear maps as functions

$$\textbf{newtype}\ a \multimap b = \mathit{LFun}\ (a \to b) \quad \text{-- } \mathit{linear}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Category}\ (\multimap)\ \textbf{where} \\
\quad \mathit{id}\ \ = \mathit{LFun}\ \mathit{id} \\
\quad (\circ) = \mathit{inNew}_2\ (\circ)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Cartesian}\ (\multimap)\ \textbf{where} \\
\quad \mathit{exl}\ \ = \mathit{LFun}\ \mathit{exl} \\
\quad \mathit{exr} = \mathit{LFun}\ \mathit{exr} \\
\quad (\vartriangle)\ = \mathit{inNew}_2\ (\vartriangle)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Cocartesian}_{\times}\ (\multimap)\ \textbf{where} \\
\quad \mathit{inl}\ \ = \mathit{LFun}\ (\lambda a \to (a, 0)) \\
\quad \mathit{inr} = \mathit{LFun}\ (\lambda b \to (0, b)) \\
\quad (\triangledown) = \mathit{inNew}_2\ (\lambda f\ g\ (a, b) \to f\ a + g\ b)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Multiplicative}\ s \Rightarrow \mathit{ScalarCat}\ (\multimap)\ s\ \textbf{where} \\
\quad \mathit{scale}\ s = \mathit{LFun}\ (s\ *)
\end{array}$$

Extracting a data representation

- Finally, extract a matrix or gradient vector.
- Very inefficient for gradient-based optimization!
- Alternatively, represent as "generalized matrices" ($M_s\ a\ b$).
  Then solve more homomorphisms.

Efficiency of composition

- 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.

Efficiency of composition

- 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 $(b \leadsto r) \to (a \leadsto r)$.
- Meaning: $f' \mapsto (\lambda h \to h \circ f')$.
- Construct $h \circ \mathcal{D}\ f\ a$ directly, without $\mathcal{D}\ f\ a$.

Old technique (Cayley 1854), vastly generalized by Yoneda.

Continuation category (specification)

$$\textbf{newtype}\ \mathit{ContC}\ ({\leadsto})\ r\ a\ b = \mathit{Cont}\ ((b \leadsto r) \rightarrow (a \leadsto r))$$

$$\begin{array}{l} \mathit{cont} :: \mathit{Category}\ ({\leadsto}) \Rightarrow (a \leadsto b) \rightarrow \mathit{ContC}\ ({\leadsto})\ r\ a\ b \\ \mathit{cont}\ f = \mathit{Cont}\ ({\circ}\, f) \end{array}$$

Specification: *cont* is a cartesian functor.

Continuation category (specification)

$$\mathbf{newtype}\ ContC\ (\leadsto)\ r\ a\ b = Cont\ ((b \leadsto r) \to (a \leadsto r))$$

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

Specification: *cont* is a cartesian functor.

We'll use an isomorphism:

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

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

Continuation category (solution)

$$
\begin{aligned}
&\textbf{instance } \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Category}\ (\mathit{ContC}\ (\leadsto)\ r)\ \textbf{where}\\
&\quad \mathit{id} = \mathit{Cont}\ \mathit{id}\\
&\quad \mathit{Cont}\ g \circ \mathit{Cont}\ f = \mathit{Cont}\ (f \circ g)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Cartesian}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ (\mathit{ContC}\ (\leadsto)\ r)\ \textbf{where}\\
&\quad \mathit{exl}\ = \mathit{Cont}\ (\mathit{join} \circ \mathit{inl})\\
&\quad \mathit{exr} = \mathit{Cont}\ (\mathit{join} \circ \mathit{inr})\\
&\quad (\vartriangle) = \mathit{inNew}_2\ (\lambda f\ g \rightarrow (f \triangledown g) \circ \mathit{unjoin})
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Cocartesian}_{\times}\ (\leadsto) \Rightarrow \mathit{Cocartesian}_{\times}\ (\mathit{ContC}\ (\leadsto)\ r)\ \textbf{where}\\
&\quad \mathit{inl}\ = \mathit{Cont}\ (\mathit{exl} \circ \mathit{unjoin})\\
&\quad \mathit{inr} = \mathit{Cont}\ (\mathit{exr} \circ \mathit{unjoin})\\
&\quad (\triangledown) = \mathit{inNew}_2\ (\lambda f\ g \rightarrow \mathit{join} \circ (f \vartriangle g))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{ScalarCat}\ (\leadsto)\ a \Rightarrow \mathit{ScalarCat}\ (\mathit{ContC}\ (\leadsto)\ r)\ a\ \textbf{where}\\
&\quad \mathit{scale}\ s = \mathit{Cont}\ (\mathit{scale}\ s)
\end{aligned}
$$

Reverse-mode AD without tears

$$D_{ContC\ M_s\ r}$$

Duality

- Vector space dual: $u^* = u \multimap s$, with u a vector space over s.
- If u has finite dimension, then $u^* \cong u$.
- Represent $a \multimap b$ by $b^* \to a^*$ by $b \to a$.
- *Ideal* for extracting gradient vector. Just apply to 1 (id).

Duality (specification)

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

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

where

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

Specification: *asDual* is a cartesian functor.

Duality (solution)

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

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

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

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

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

Backpropagation

Backpropagation

$$D_{Dual_{\multimap}}$$

Conclusions

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

Running examples

$$sqr :: Num\ a \Rightarrow a \to a$$
$$sqr\ a = a * a$$

$$magSqr :: Num\ a \Rightarrow a \times a \to a$$
$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

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

Running examples

$$sqr :: Num\ a \Rightarrow a \to a$$
$$sqr\ a = a * a$$

$$magSqr :: Num\ a \Rightarrow a \times a \to a$$
$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$cosSinProd :: Floating\ a \Rightarrow a \times a \to a \times a$$
$$cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \textbf{where}\ z = x * y$$

In categorical vocabulary:

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

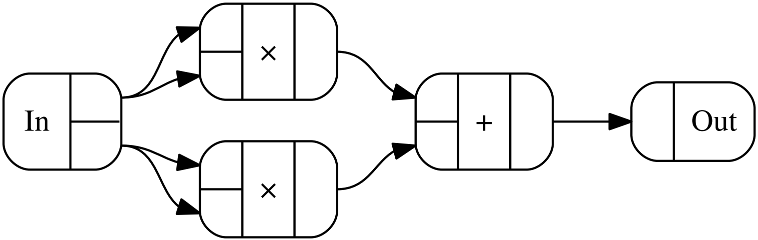
$$magSqr = addC \circ ((sqr \circ exl) \vartriangle (sqr \circ exr))$$

$$cosSinProd = (cosC \vartriangle sinC) \circ mulC$$

Visualizing computations

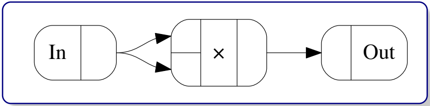
$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$magSqr = addC \circ ((sqr \circ exl) \vartriangle (sqr \circ exr))$$



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

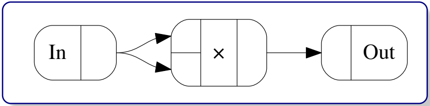
AD example



$$sqr\ a = a * a$$

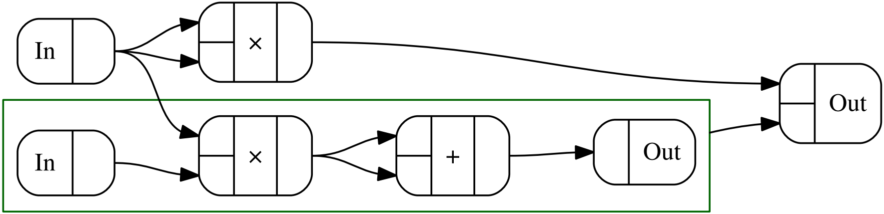
$$sqr = mulC \circ (id \vartriangle id)$$

AD example

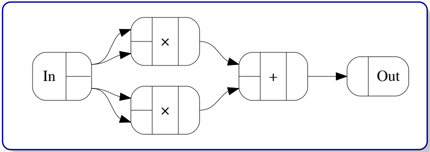


$$sqr\ a = a * a$$

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



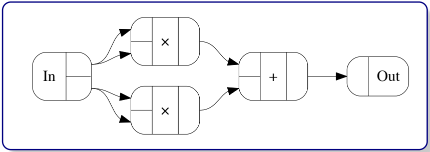
AD example



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

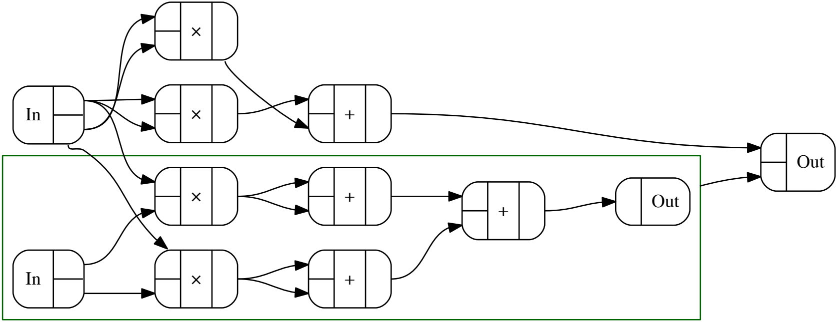
$$magSqr = addC \circ ((sqr \circ exl) \vartriangle (sqr \circ exr))$$

AD example

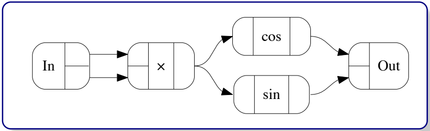


$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$magSqr = addC \circ ((sqr \circ exl) \vartriangle (sqr \circ exr))$$



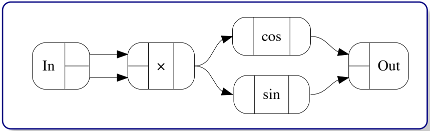
AD example



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

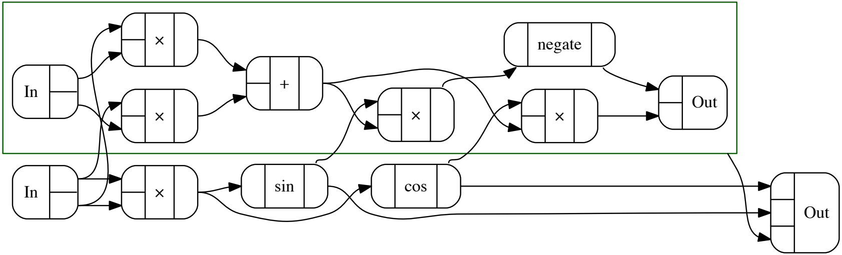
$$cosSinProd = (cosC \vartriangle sinC) \circ mulC$$

AD example

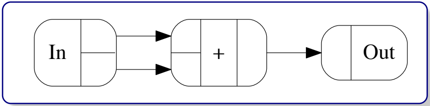


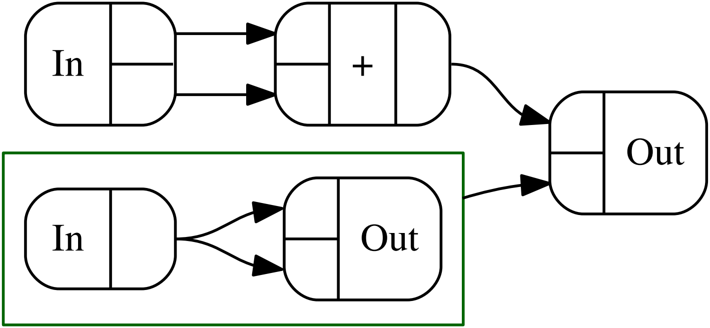
$$cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \textbf{where}\ z = x * y$$

$$cosSinProd = (cosC \vartriangle sinC) \circ mulC$$

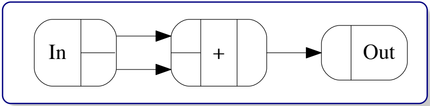


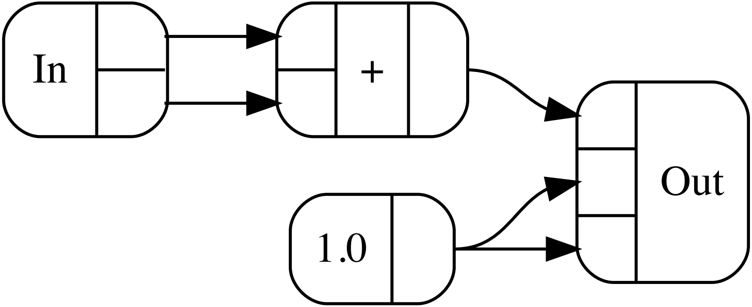
RAD example (dual function)



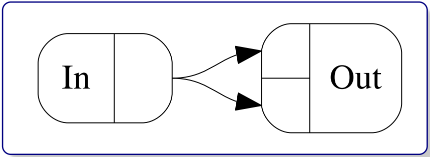


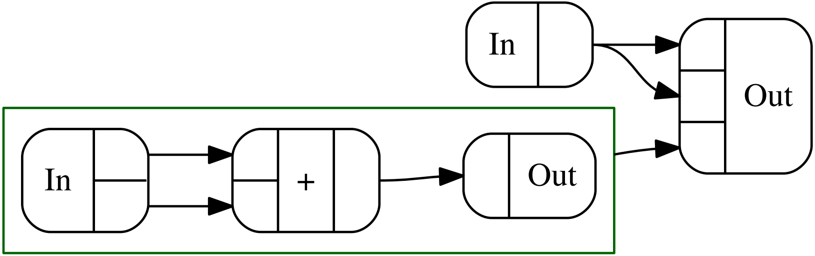
RAD example (dual vector)



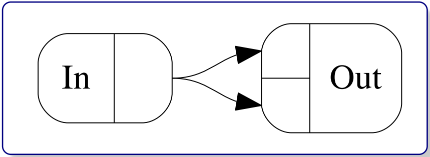


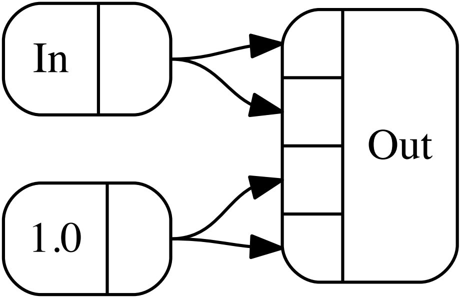
RAD example (dual function)



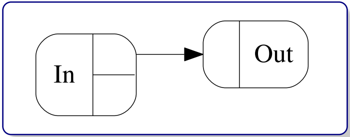


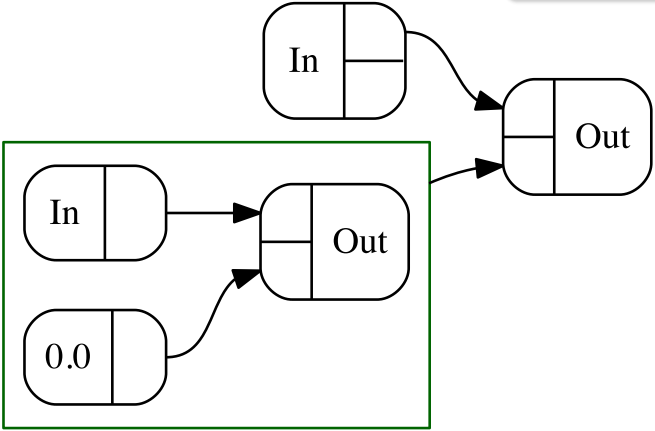
RAD example (vector)



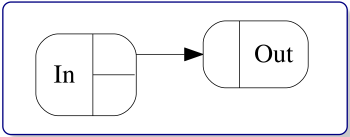


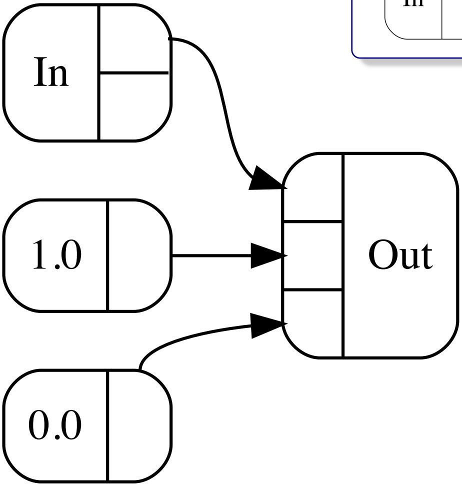
RAD example (dual function)



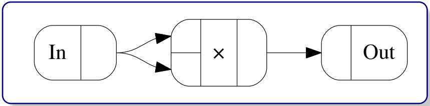


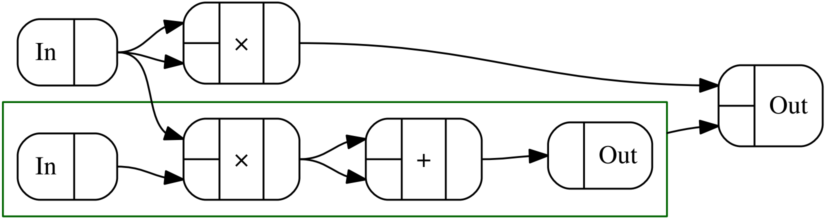
RAD example (dual vector)



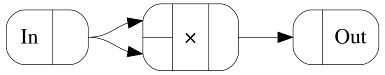


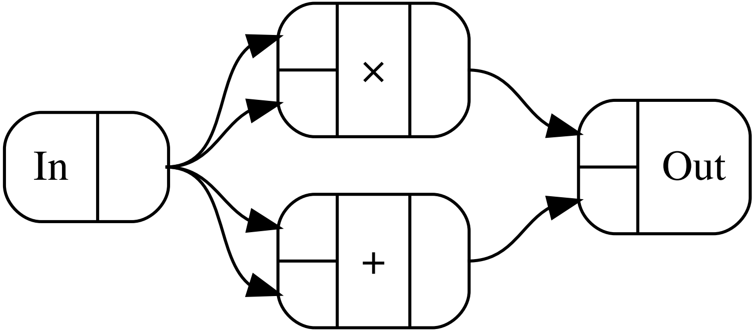
RAD example (dual function)



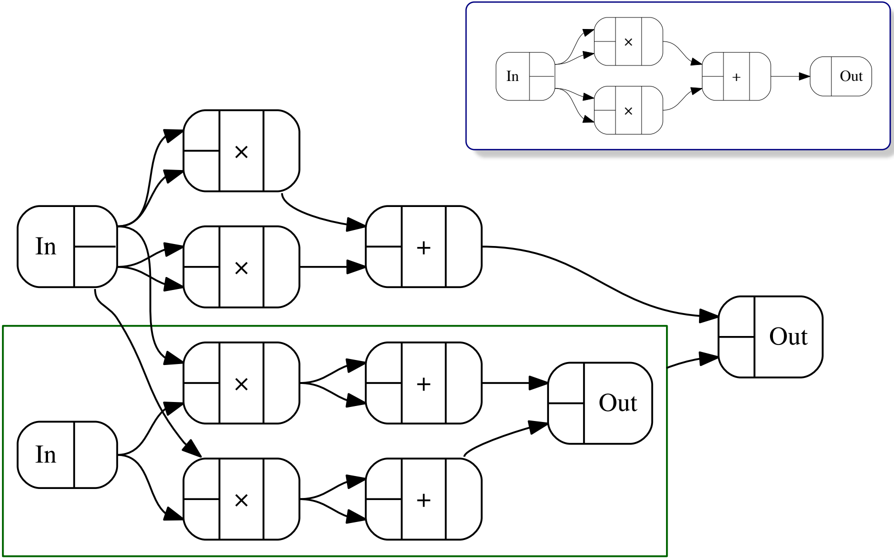


RAD example (dual vector)

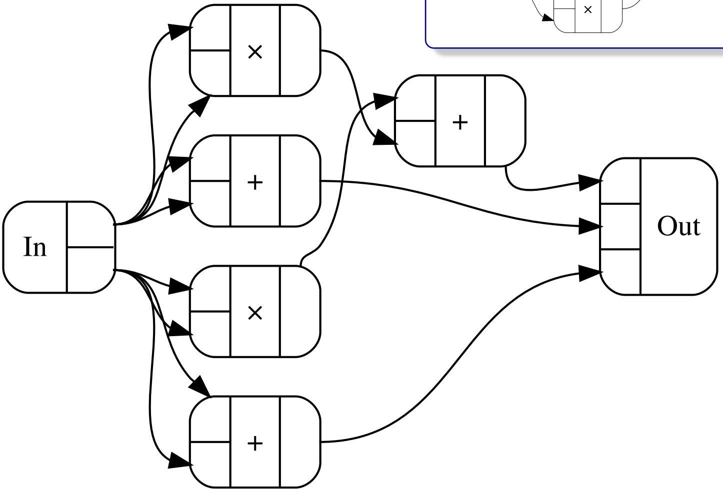


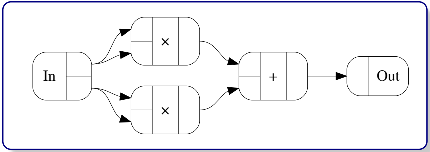


RAD example (dual function)

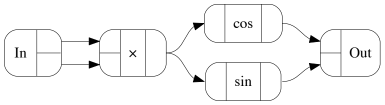


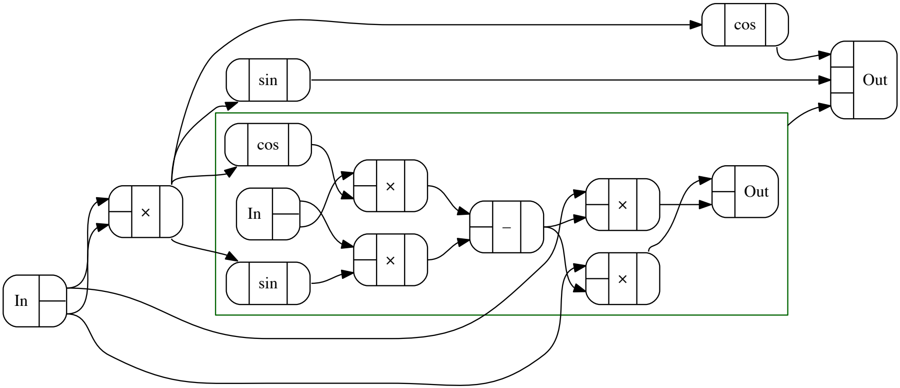
RAD example (dual vector)



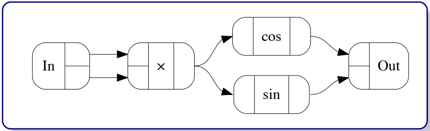


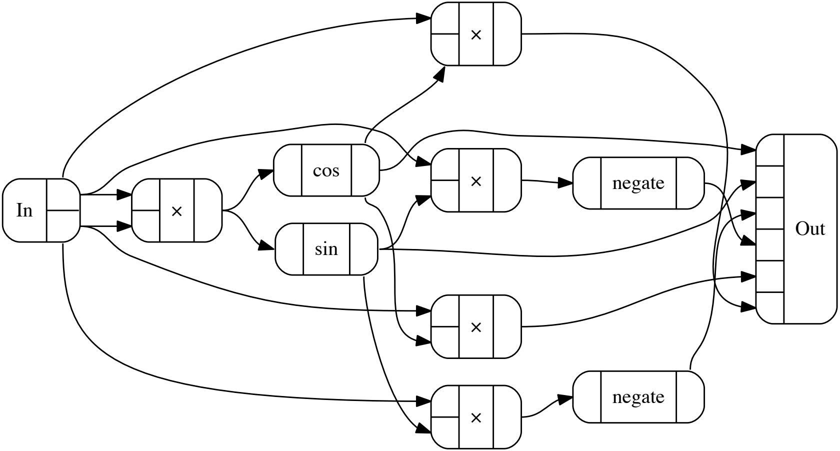
RAD example (dual function)





RAD example (matrix)





Reflections: recipe for success

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).

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.

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.

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.