The simple essence of automatic differentiation

Conal Elliott

Target

January/June 2018

Machine learning: promise and problems

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

Machine learning: promise and problems

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

- AD algorithm (backprop) is complex and stateful.
- Complex graph APIs.

Machine learning: promise and problems

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

- AD algorithm (backprop) is complex and stateful.
- Complex graph APIs.

Solutions:

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

What's a derivative?

What's a derivative?

- Number

What's a derivative?

- Number
- Vector

What's a derivative?

- Number
- Vector
- Covector

What's a derivative?

- Number
- Vector
- Covector
- Matrix

What's a derivative?

- Number
- Vector
- Covector
- Matrix
- Higher derivatives

What's a derivative?

- Number
- Vector
- Covector
- Matrix
- Higher derivatives

Chain rule for each.

Derivatives as linear maps (Fréchet)

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

$\mathcal{D}\ f\ a$ is a local *linear* approximation to f at a.

Derivatives as linear maps (Fréchet)

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

$\mathcal{D}\ f\ a$ is a local *linear* approximation to f at a:

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

Composition

Sequential:

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

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

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.

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

Compositionality

Chain rule:

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

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

$$\begin{aligned}&\textbf{class } \mathit{Category}\ (\rightsquigarrow) \textbf{ where}\\ &\quad \mathit{id} \ :: a \rightsquigarrow a\\ &\quad (\circ) :: (b \rightsquigarrow c) \to (a \rightsquigarrow b) \to (a \rightsquigarrow c)\end{aligned}$$

$$\begin{aligned}&\textbf{class } \mathit{Category}\ (\rightsquigarrow) \Rightarrow \mathit{Cartesian}\ (\rightsquigarrow) \textbf{ where}\\ &\quad \mathit{exl} :: (a \times b) \rightsquigarrow a\\ &\quad \mathit{exr} :: (a \times b) \rightsquigarrow b\\ &\quad (\vartriangle) :: (a \rightsquigarrow c) \to (a \rightsquigarrow d) \to (a \rightsquigarrow (c \times d))\end{aligned}$$

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: $\hat{\mathcal{D}}$ 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.

Automatic differentiation (solution)

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

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

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

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

$$\textbf{instance}\ NumCat\ D\ \textbf{where}$$
$$negate = linearD\ negate$$
$$add = linearD\ add$$
$$mul = D\ (mul \vartriangle (\lambda(a, b) \to \lambda(da, db) \to b * da + a * db))$$

Running examples

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

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

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

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:

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

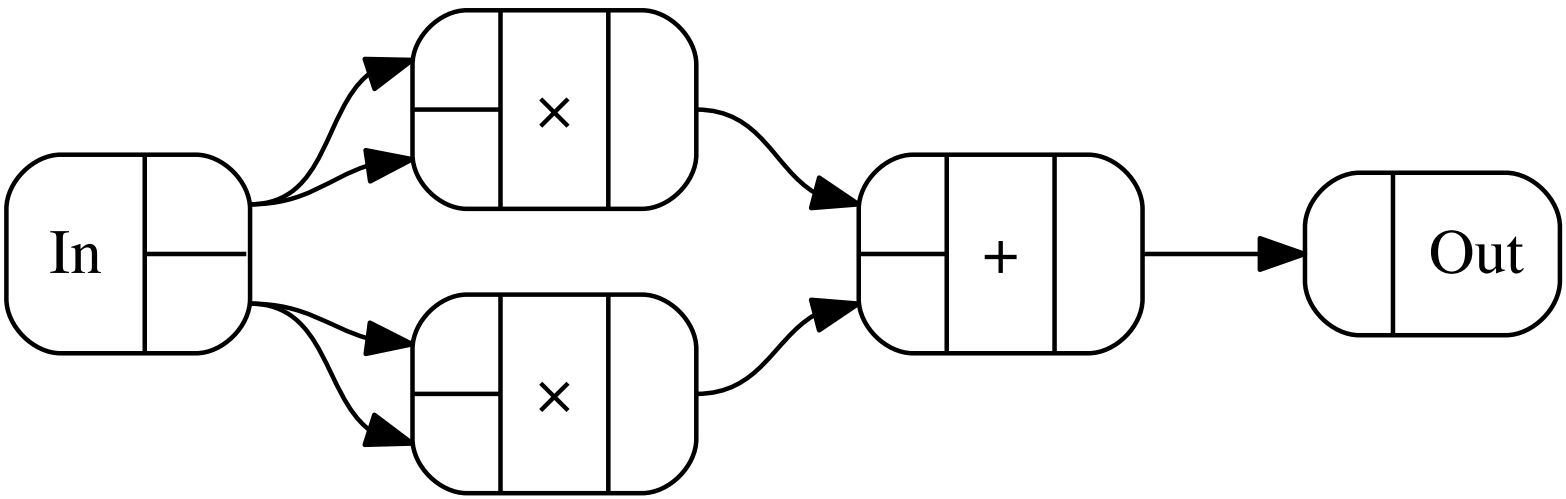
$$magSqr = add \circ ((sqr \circ exl) \vartriangle (sqr \circ exr))$$

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

Visualizing computations

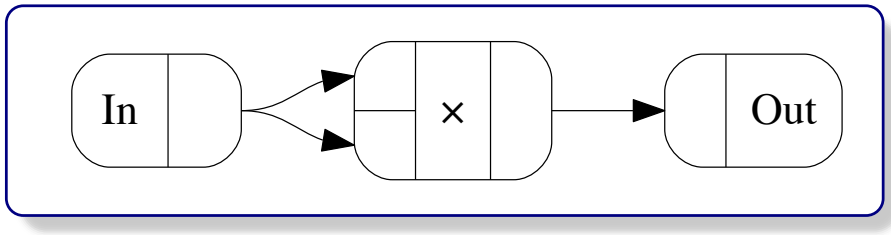
$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

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



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

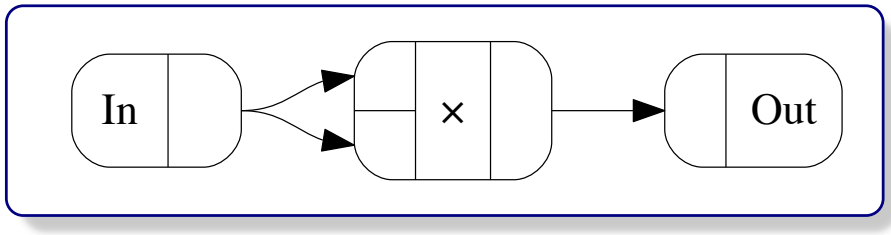
AD example



$$sqr\ a = a * a$$

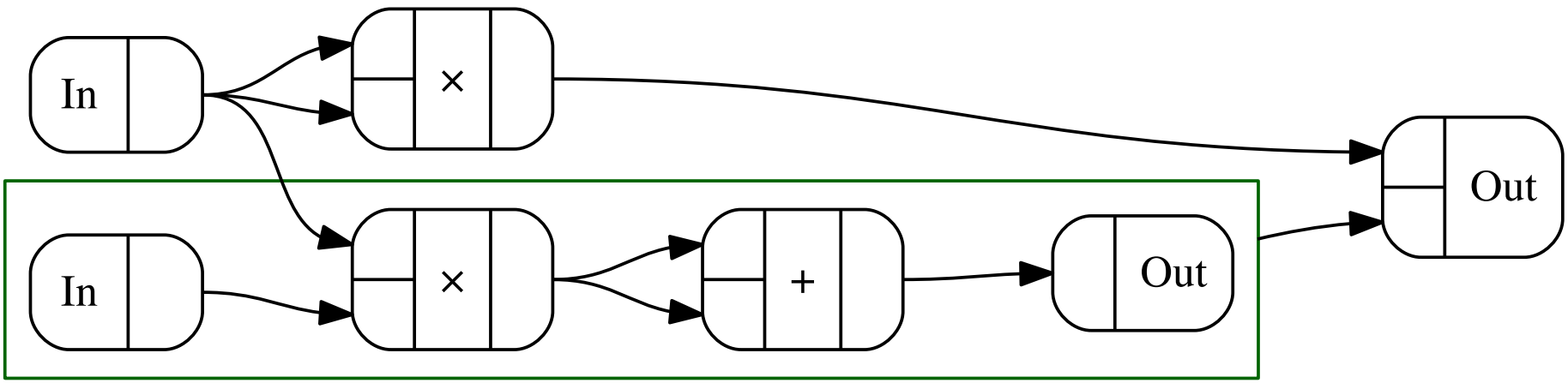
$$sqr = mul \circ (id \vartriangle id)$$

AD example

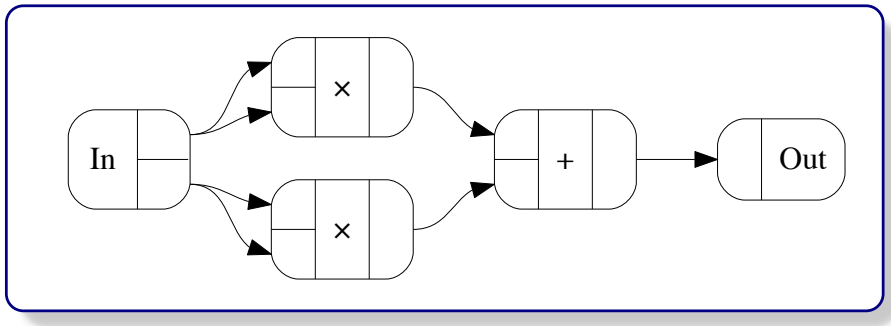


$$sqr\ a = a * a$$

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



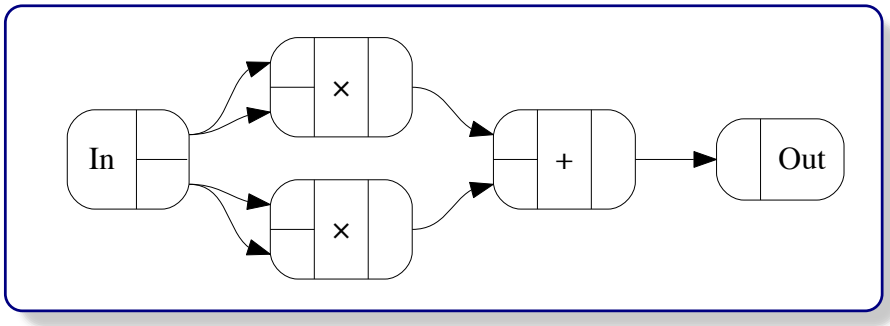
AD example



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

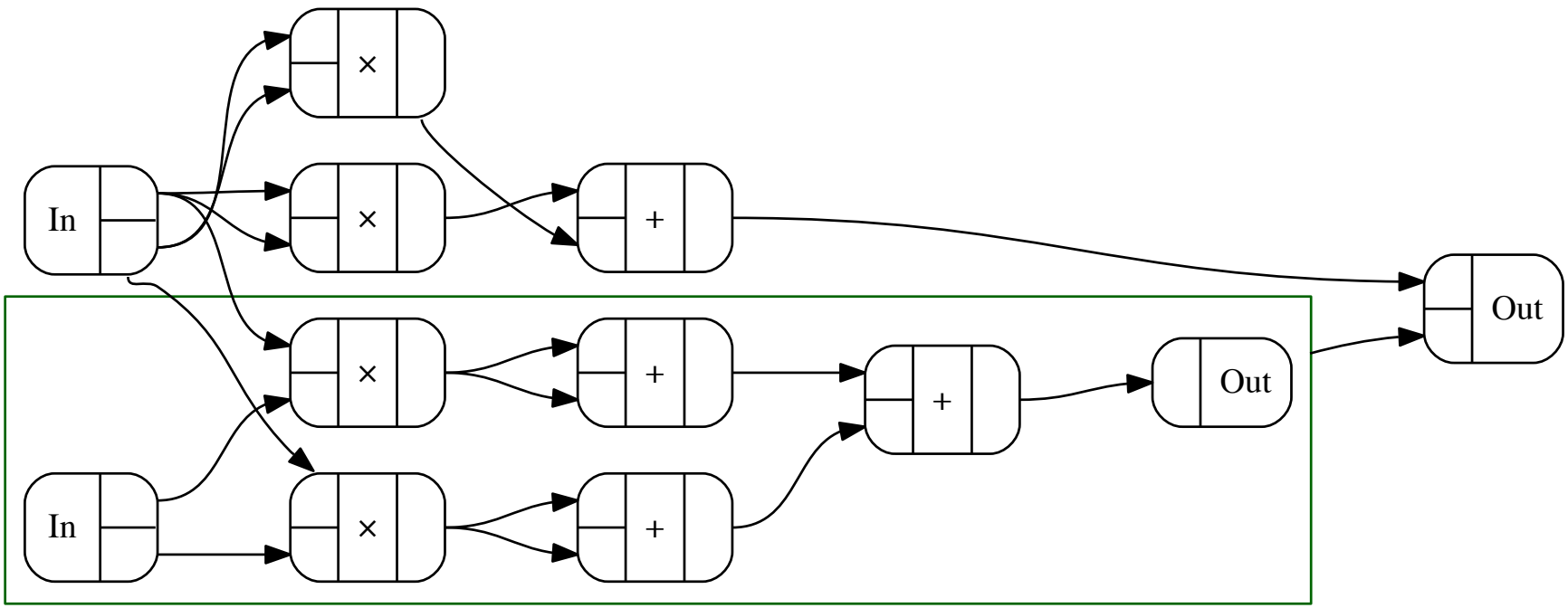
$$magSqr = add \circ ((sqr \circ exl) \vartriangle (sqr \circ exr))$$

AD example

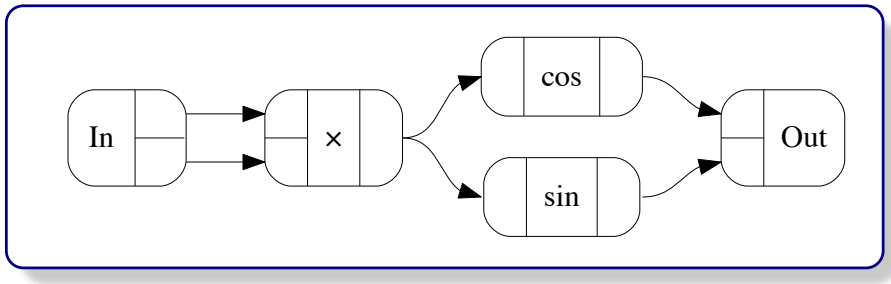


$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

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



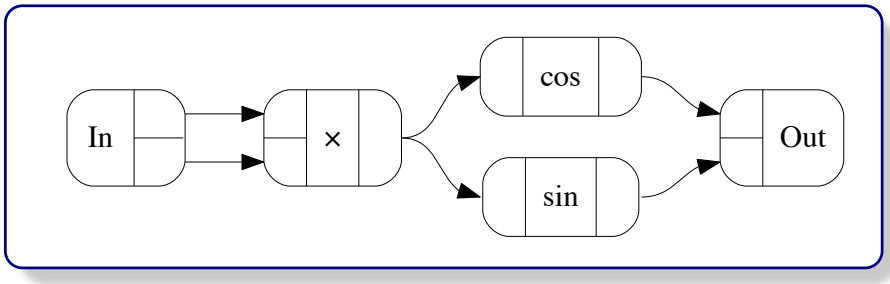
AD example



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

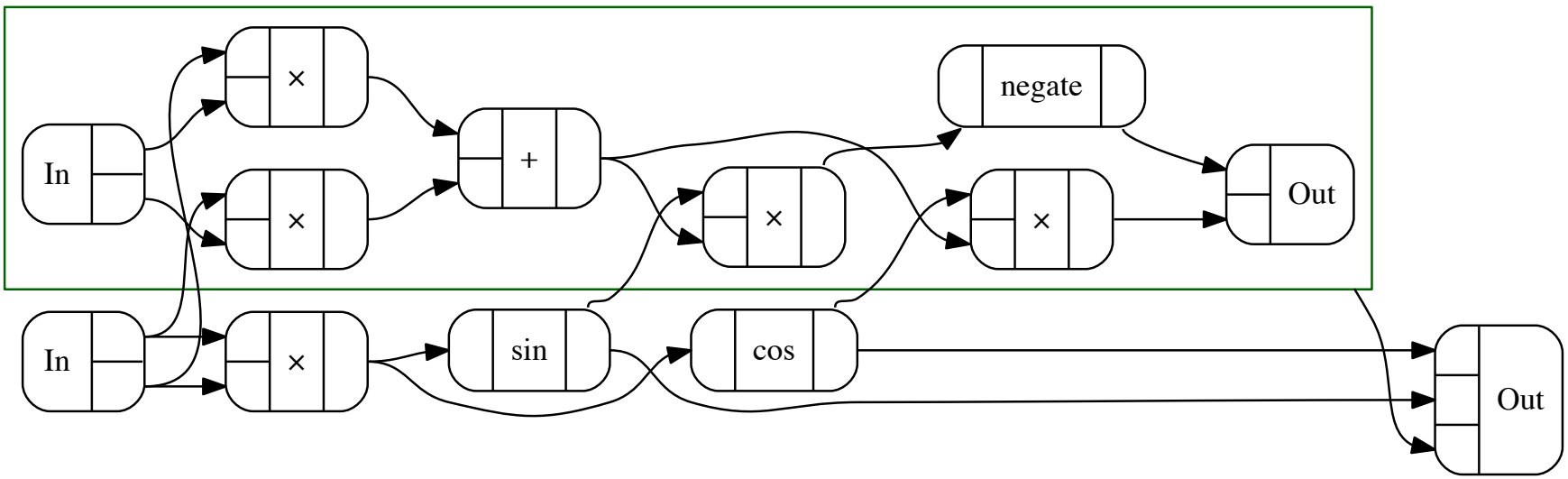
$$cosSinProd = (cos \vartriangle sin) \circ mul$$

AD example



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

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



Generalizing AD

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

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

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

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

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

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

Generalized AD

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

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

$$\textbf{instance}\ Category\ (\rightsquigarrow) \Rightarrow Category\ D_{(\rightsquigarrow)}\ \textbf{where}$$

$$id = linearD\ id\ id$$

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

$$\textbf{instance}\ Cartesian\ (\rightsquigarrow) \Rightarrow Cartesian\ D_{(\rightsquigarrow)}\ \textbf{where}$$

$$exl = linearD\ exl\ exl$$

$$exr = linearD\ exr\ exr$$

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

$$\textbf{instance}\ \ldots \Rightarrow NumCat\ D\ \textbf{where}$$

$$negate = linearD\ negate\ negate$$

$$add = linearD\ add\ add$$

$$mul = ??$$

Numeric operations

Specific to (linear) *functions*:

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

Numeric operations

Specific to (linear) *functions*:

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

Rephrase:

$$scale :: Multiplicative\ a \Rightarrow a \to (a \to a)$$
$$scale\ u = \lambda v \to u * v$$

$$(\triangledown) :: (a \to c) \to (b \to c) \to ((a \times b) \to c)$$
$$(f \triangledown g)\ (a, b) = f\ a + g\ b$$

Now

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

New generalized vocabulary

$$\begin{aligned}&\textbf{class } \mathit{Category}\ (\rightsquigarrow) \Rightarrow \mathit{Cocartesian}\ (\rightsquigarrow) \textbf{ where}\\ &\quad \mathit{inl} :: a \rightsquigarrow (a \times b)\\ &\quad \mathit{inr} :: b \rightsquigarrow (a \times b)\\ &\quad (\triangledown) :: (a \rightsquigarrow c) \to (b \rightsquigarrow c) \to ((a \times b) \rightsquigarrow c)\end{aligned}$$

$$\begin{aligned}&\textbf{class } \mathit{ScalarCat}\ (\rightsquigarrow)\ a \textbf{ where}\\ &\quad \mathit{scale} :: a \to (a \rightsquigarrow a)\end{aligned}$$

Differentiation:

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

The rest are linear.

Linear transformations as functions

$$\textbf{newtype}\ a \mathbin{\overset{+}{\to}} b = \mathit{AddFun}\ (a \to b)$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Category}\ (\overset{+}{\to})\ \textbf{where}\\
&\quad \mathit{id}\ \ = \mathit{AddFun}\ \mathit{id}\\
&\quad (\circ) = \mathit{inNew}_2\ (\circ)
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Cartesian}\ (\overset{+}{\to})\ \textbf{where}\\
&\quad \mathit{exl}\ = \mathit{AddFun}\ \mathit{exl}\\
&\quad \mathit{exr} = \mathit{AddFun}\ \mathit{exr}\\
&\quad (\triangle)\ = \mathit{inNew}_2\ (\triangle)
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Cocartesian}\ (\overset{+}{\to})\ \textbf{where}\\
&\quad \mathit{inl}\ = \mathit{AddFun}\ (\lambda a \to (a, 0))\\
&\quad \mathit{inr}\ = \mathit{AddFun}\ (\lambda b \to (0, b))\\
&\quad (\triangledown) = \mathit{inNew}_2\ (\lambda f\ g\ (a, b) \to f\ a + g\ b)
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Multiplicative}\ s \Rightarrow \mathit{ScalarCat}\ (\overset{+}{\to})\ s\ \textbf{where}\\
&\quad \mathit{scale}\ s = \mathit{AddFun}\ (s *)
\end{aligned}$$

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

$$\textbf{newtype}\ M_s\ a\ b = \ldots$$

$$applyM :: M_s\ a\ b \to (a \to b)$$

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

Compositional matrices

Sufficient to build arbitrary "matrices":

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

Types guarantee rectangularity.

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 $a \rightsquigarrow b$ by $(b \rightsquigarrow r) \to (a \rightsquigarrow r)$.
- 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)

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

$$cont :: Category\ (\rightsquigarrow) \Rightarrow (a \rightsquigarrow b) \to Cont^{r}_{(\rightsquigarrow)}\ a\ b$$
$$cont\ f = Cont\ (\circ f)$$

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

Continuation category (specification)

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

$$cont :: Category\ (\rightsquigarrow) \Rightarrow (a \rightsquigarrow b) \to Cont^{r}_{(\rightsquigarrow)}\ a\ b$$
$$cont\ f = Cont\ (\circ f)$$

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

We'll use an isomorphism:

$$join :: Cocartesian\ (\rightsquigarrow) \Rightarrow (c \rightsquigarrow a) \times (d \rightsquigarrow a) \to ((c \times d) \rightsquigarrow a)$$
$$unjoin :: Cocartesian\ (\rightsquigarrow) \Rightarrow ((c \times d) \rightsquigarrow a) \to (c \rightsquigarrow a) \times (d \rightsquigarrow a)$$

$$join\ (f, g) = f \triangledown g$$
$$unjoin\ h = (h \circ inl, h \circ inr)$$

Continuation category (solution)

$$\textbf{instance } \mathit{Category}\ (\rightsquigarrow) \Rightarrow \mathit{Category}\ \mathit{Cont}^r_{(\rightsquigarrow)} \textbf{ where}$$

$$\begin{aligned} \mathit{id} &= \mathit{Cont}\ \mathit{id} \\ \mathit{Cont}\ g \circ \mathit{Cont}\ f &= \mathit{Cont}\ (f \circ g) \end{aligned}$$

$$\textbf{instance } \mathit{Cartesian}\ (\rightsquigarrow) \Rightarrow \mathit{Cartesian}\ \mathit{Cont}^r_{(\rightsquigarrow)} \textbf{ where}$$

$$\begin{aligned} \mathit{exl} &= \mathit{Cont}\ (\mathit{join} \circ \mathit{inl}) \\ \mathit{exr} &= \mathit{Cont}\ (\mathit{join} \circ \mathit{inr}) \\ (\vartriangle) &= \mathit{inNew}_2\ (\lambda f\ g \rightarrow (f \vartriangledown g) \circ \mathit{unjoin}) \end{aligned}$$

$$\textbf{instance } \mathit{Cocartesian}\ (\rightsquigarrow) \Rightarrow \mathit{Cocartesian}\ \mathit{Cont}^r_{(\rightsquigarrow)} \textbf{ where}$$

$$\begin{aligned} \mathit{inl} &= \mathit{Cont}\ (\mathit{exl} \circ \mathit{unjoin}) \\ \mathit{inr} &= \mathit{Cont}\ (\mathit{exr} \circ \mathit{unjoin}) \\ (\vartriangledown) &= \mathit{inNew}_2\ (\lambda f\ g \rightarrow \mathit{join} \circ (f \vartriangle g)) \end{aligned}$$

$$\textbf{instance } \mathit{ScalarCat}\ (\rightsquigarrow)\ a \Rightarrow \mathit{ScalarCat}\ \mathit{Cont}^r_{(\rightsquigarrow)}\ a \textbf{ where}$$

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

Reverse-mode AD without tears

Reverse-mode AD without tears

$$D_{Cont^r_{M_s}}$$

Left-associating composition (RAD)

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

Left-associating composition (RAD)

- CPS-like category:
  - Represent $a \rightsquigarrow b$ by $(b \rightsquigarrow r) \to (a \rightsquigarrow r)$.
  - Meaning: $f \mapsto (\circ f)$.
  - Results in left-composition.
  - Initialize with $id :: r \rightsquigarrow r$.
  - 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., $(+\!\!+\ as)$.

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 $f :: u \multimap s$, $f = dot\ v$ for some $v :: u$.
- Gradients are derivatives of functions with scalar codomain.
- Represent $a \multimap b$ by $(b \multimap s) \to (a \multimap s)$ by $b \to a$.
- *Ideal* for extracting gradient vector. Just apply to 1 (id).

Duality (specification)

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

$$asDual :: Cont^{s}_{(\rightsquigarrow)}\ a\ b \to Dual_{(\rightsquigarrow)}\ a\ b$$
$$asDual\ (Cont\ f) = Dual\ (dot^{-1} \circ f \circ dot)$$

where

$$dot \quad :: u \to (u \multimap s)$$
$$dot^{-1} :: (u \multimap s) \to u$$

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

Duality (solution)

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

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

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

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

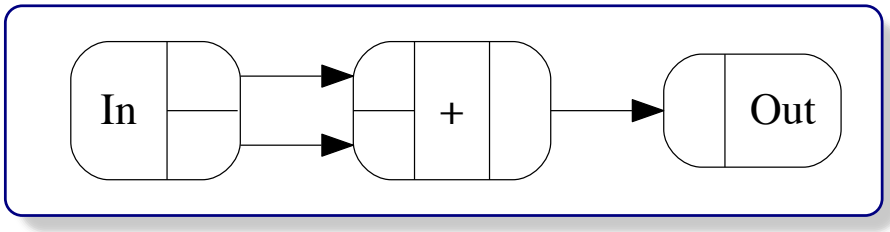
$$\textbf{instance}\ \mathit{ScalarCat}\ (\rightsquigarrow)\ s \Rightarrow \mathit{ScalarCat}\ \mathit{Dual}_{(\rightsquigarrow)}\ s\ \textbf{where}$$
$$\mathit{scale}\ s = \mathit{Dual}\ (\mathit{scale}\ s)$$

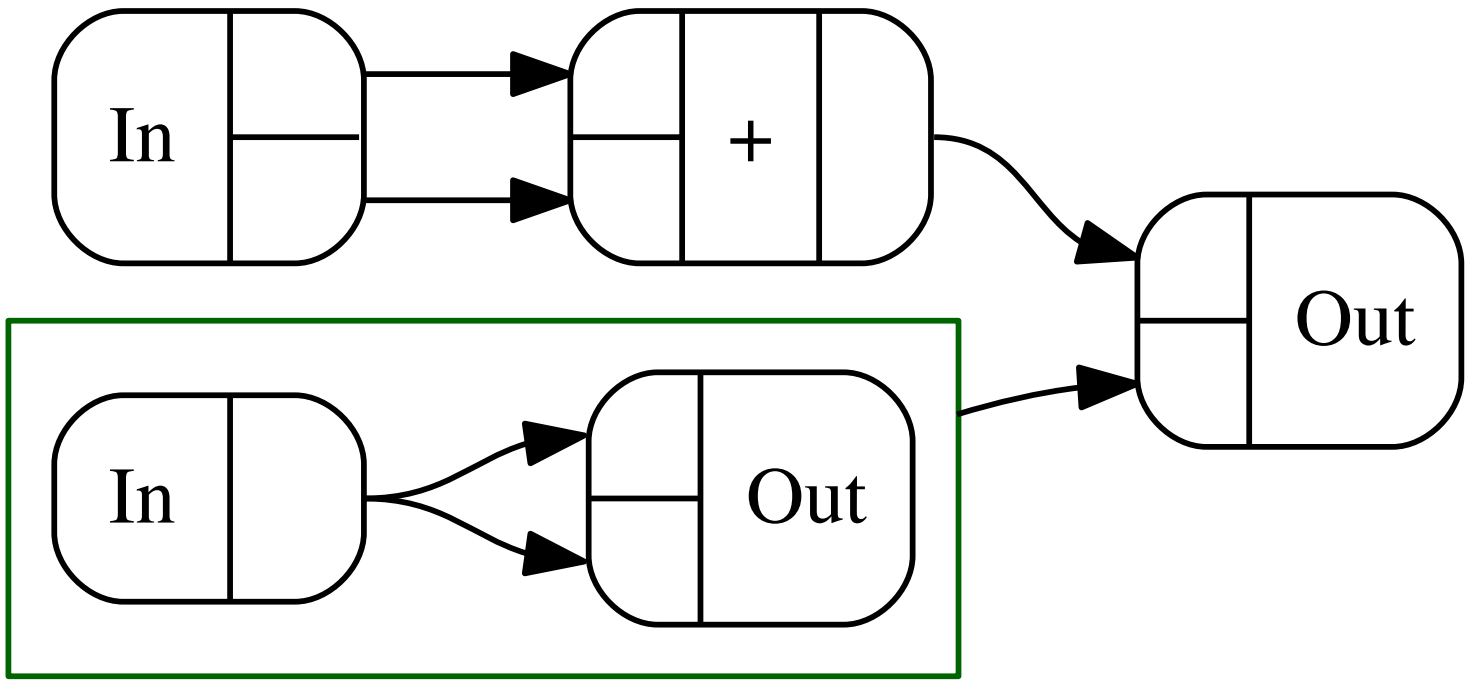
Backpropagation

Backpropagation

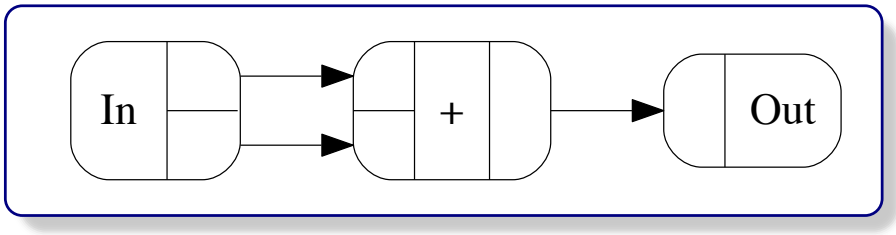
$$D_{Dual_{\rightarrow}+}$$

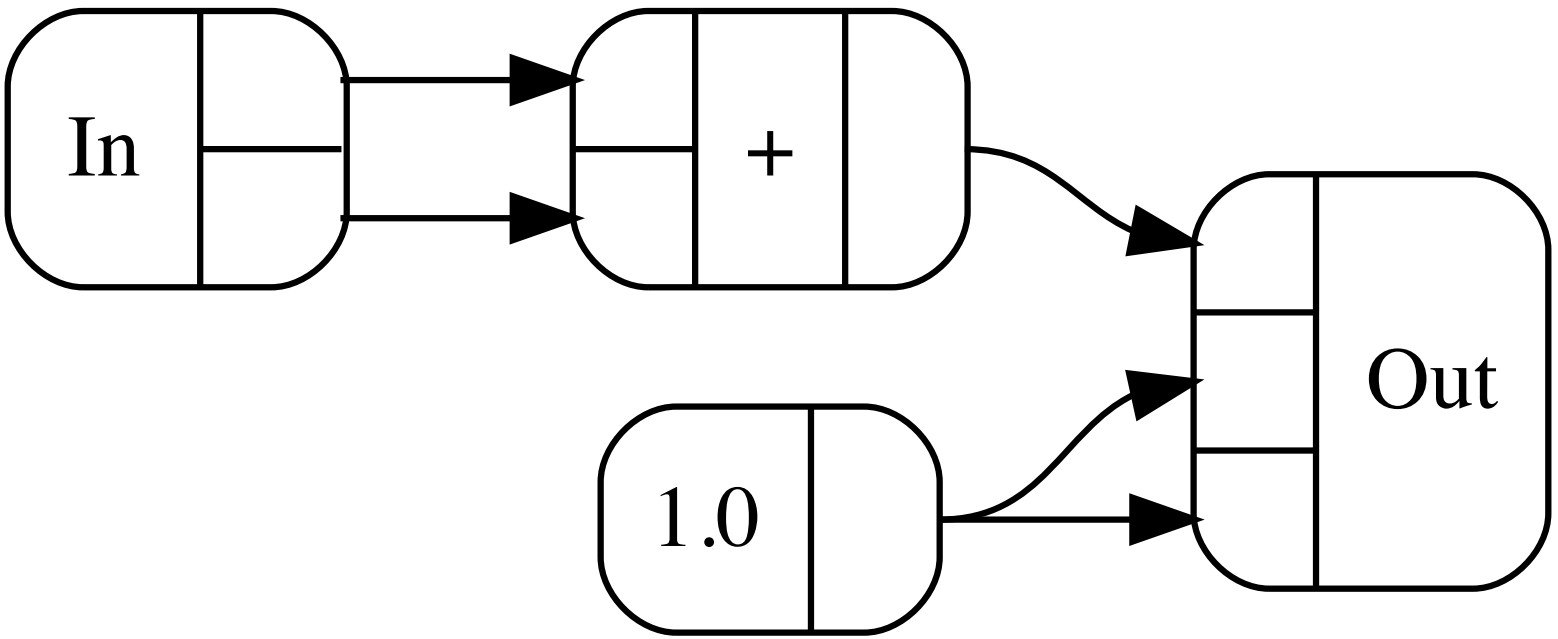
RAD example (dual function)



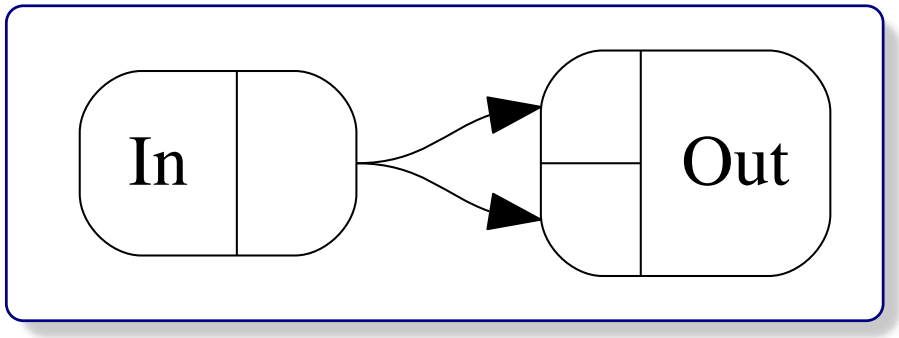


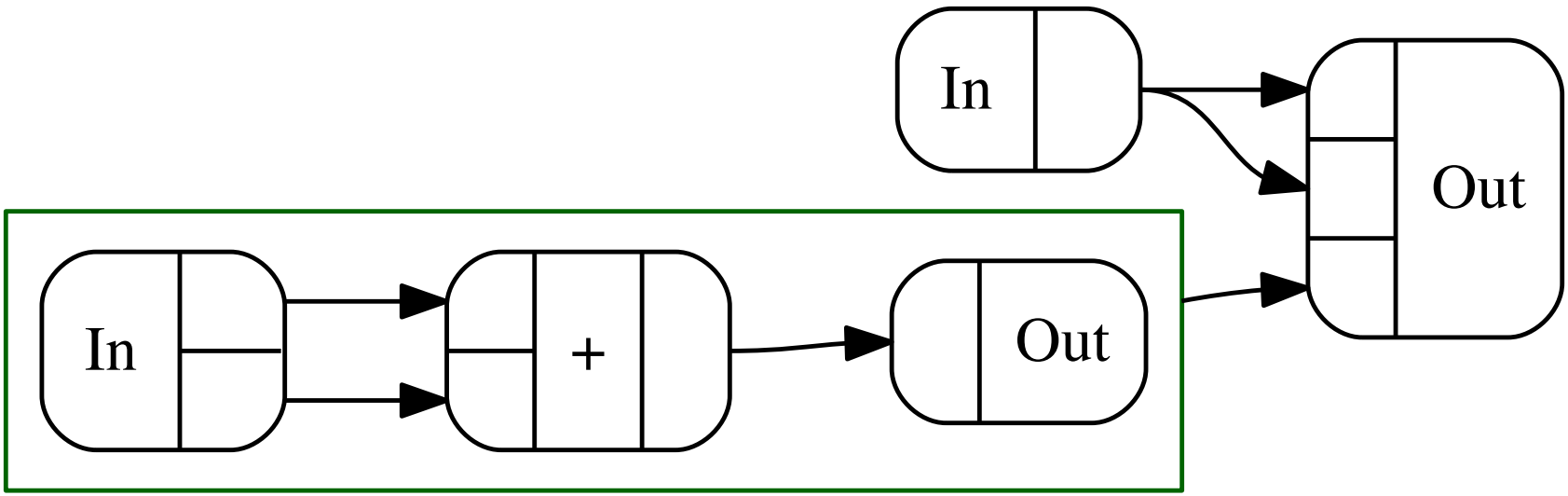
RAD example (dual vector)



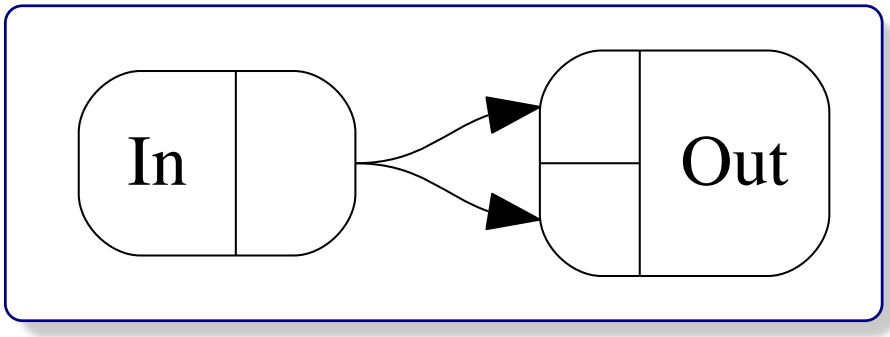


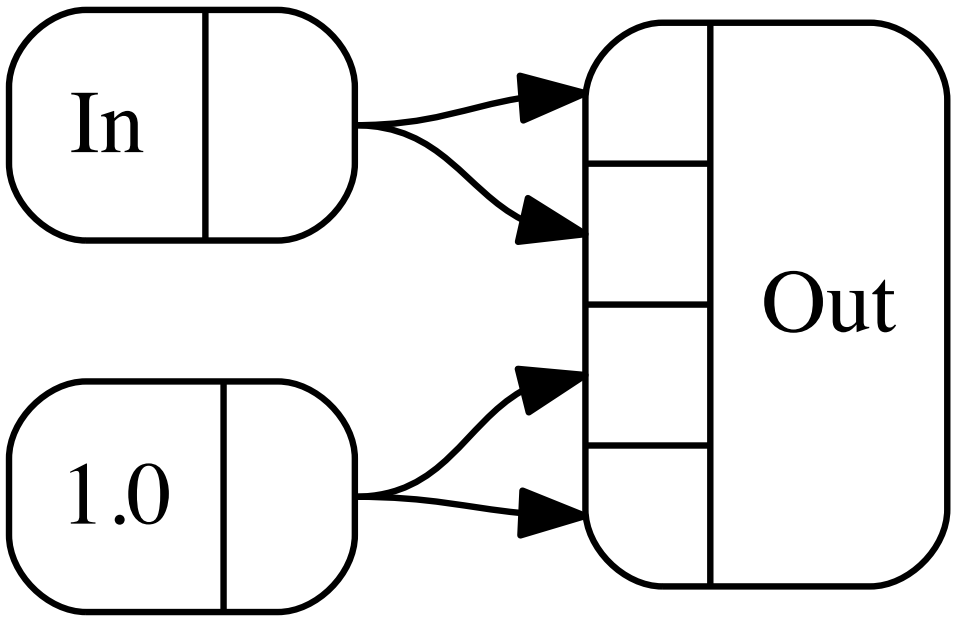
RAD example (dual function)



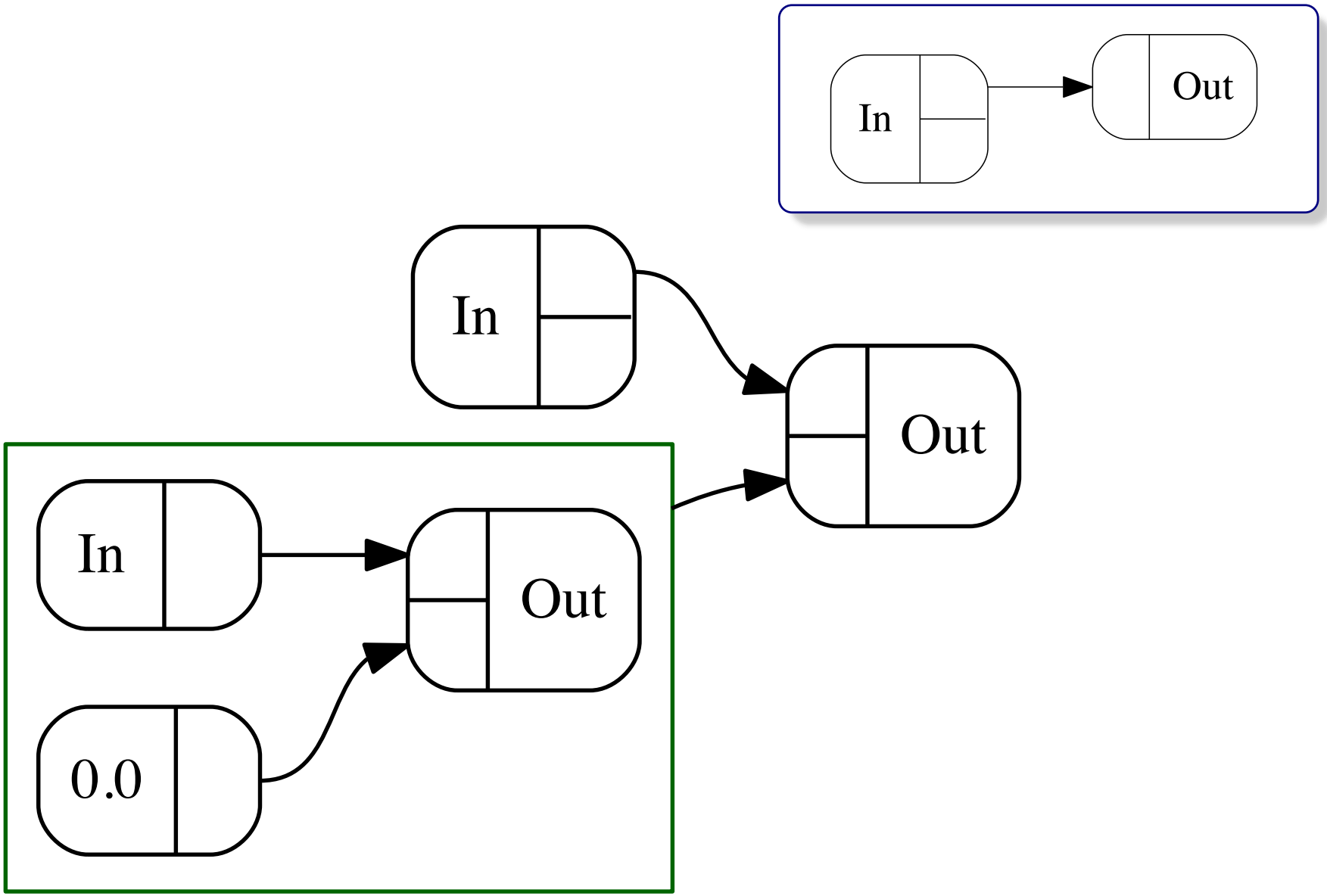


RAD example (vector)

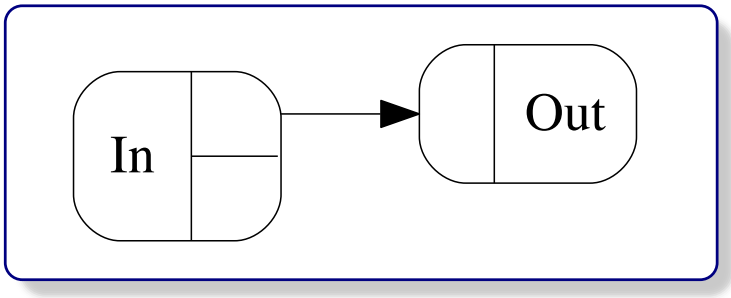


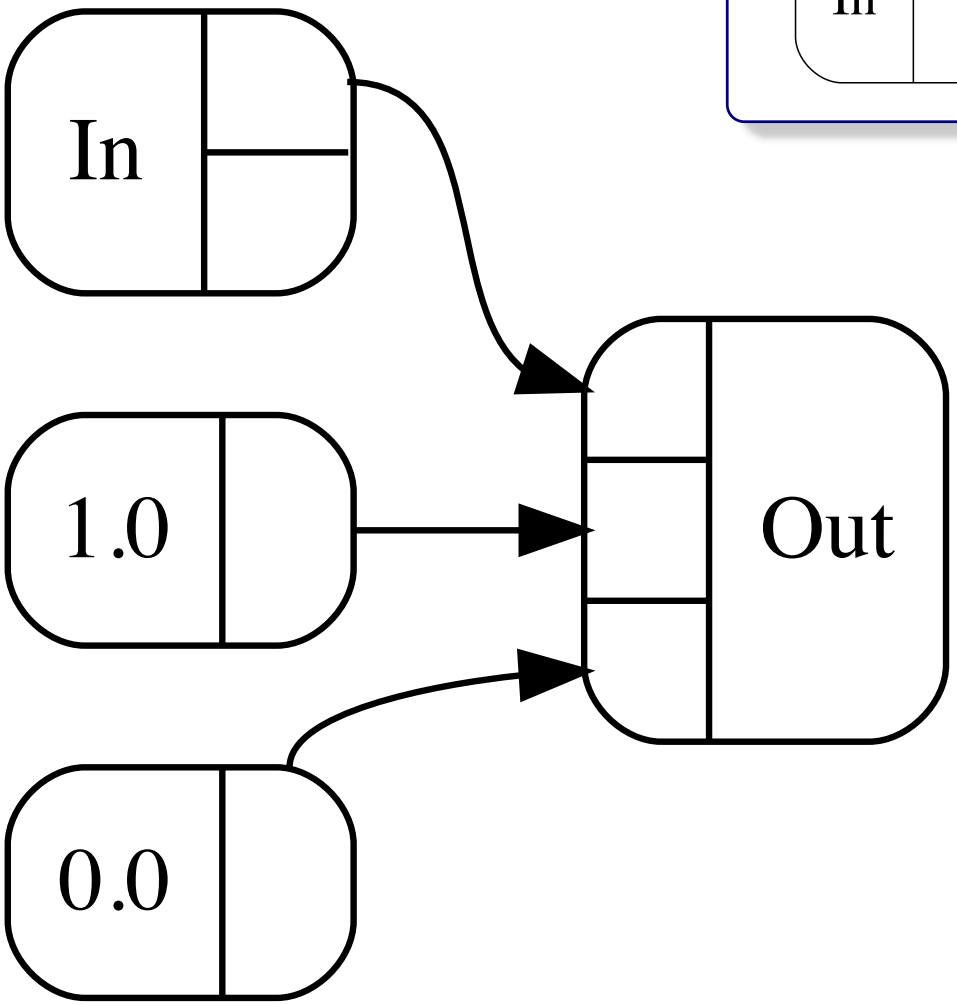


RAD example (dual function)

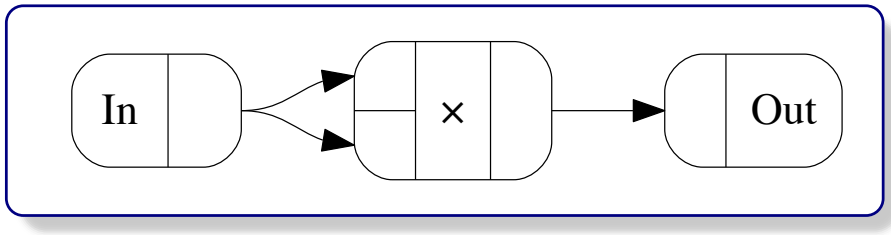


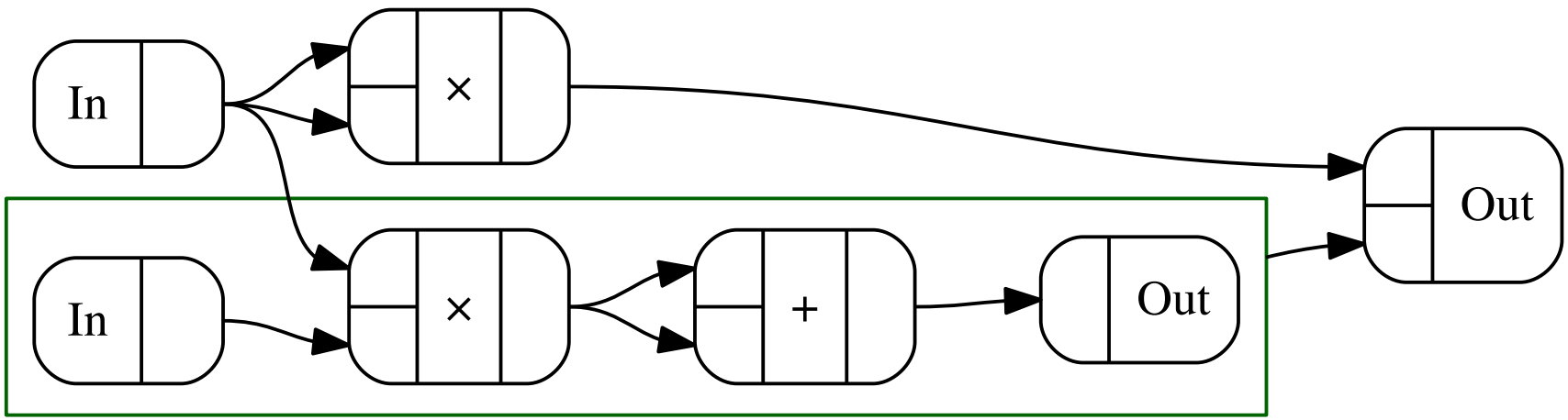
RAD example (dual vector)



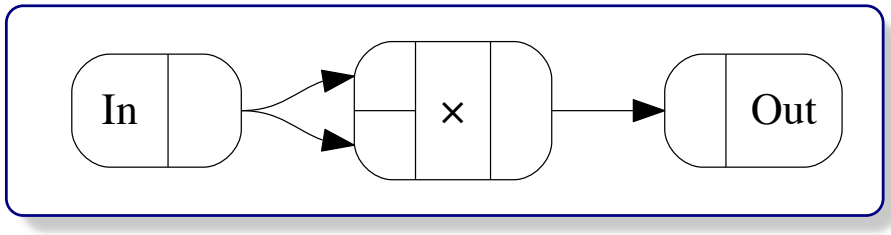


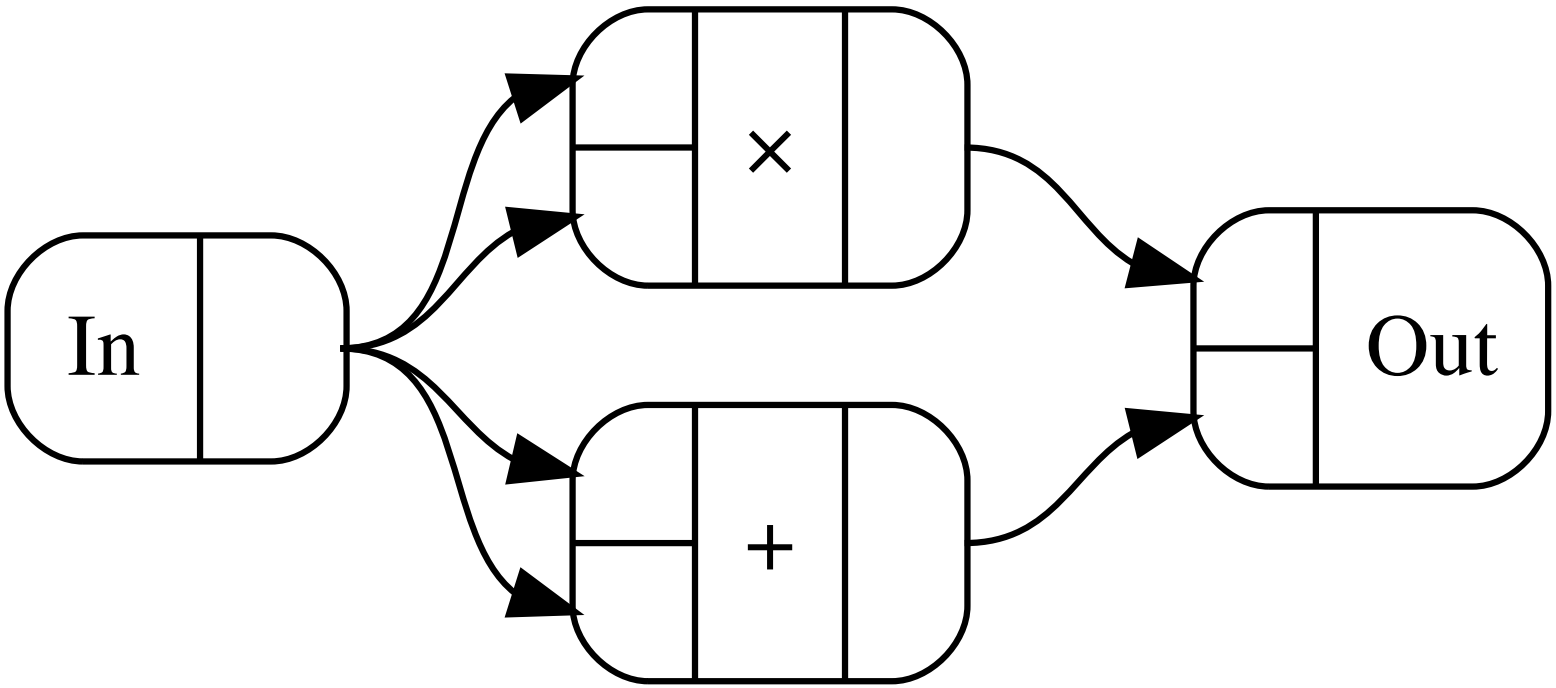
RAD example (dual function)



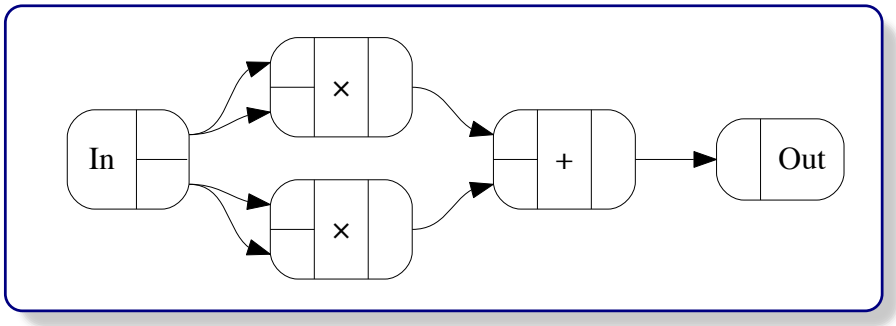


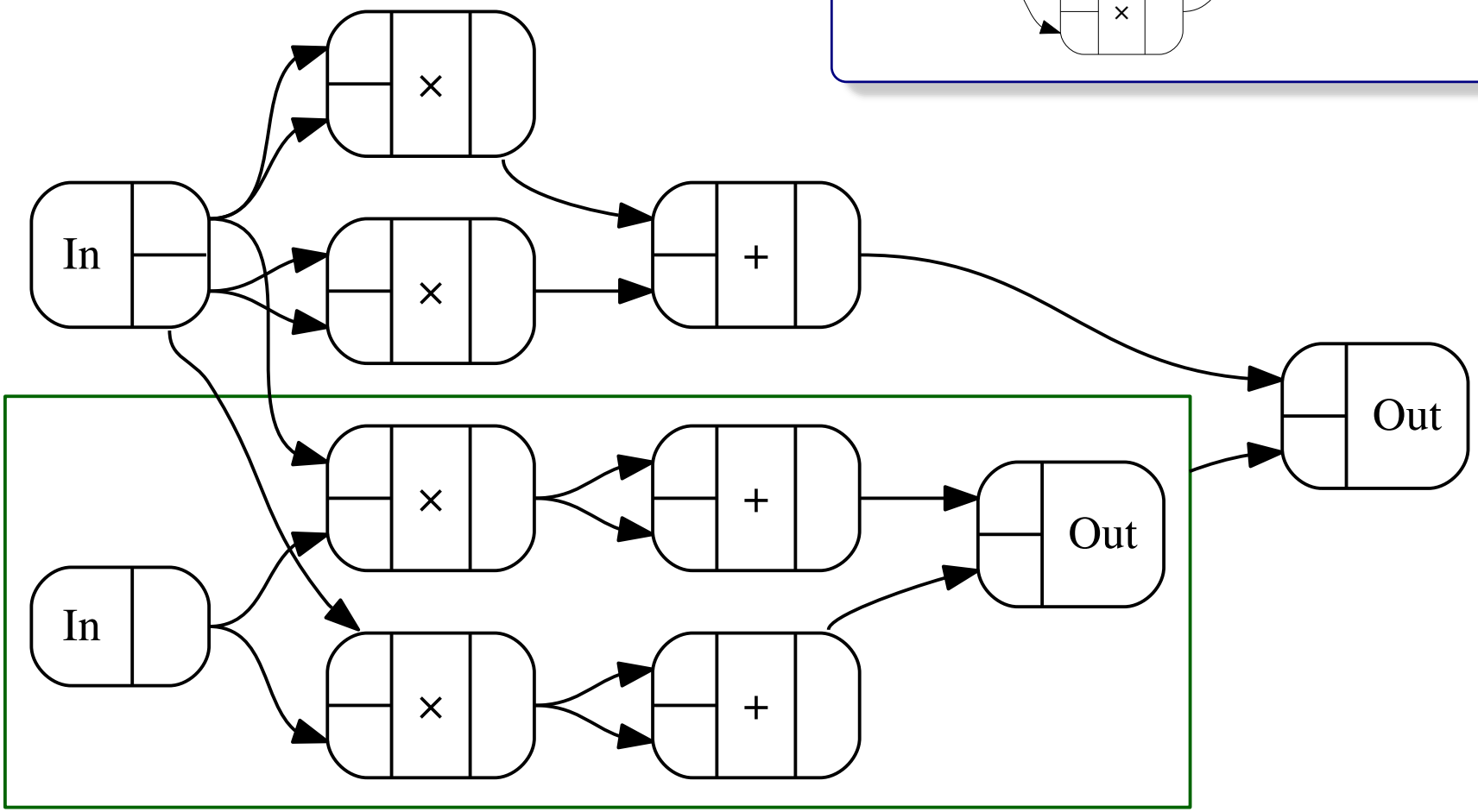
RAD example (dual vector)



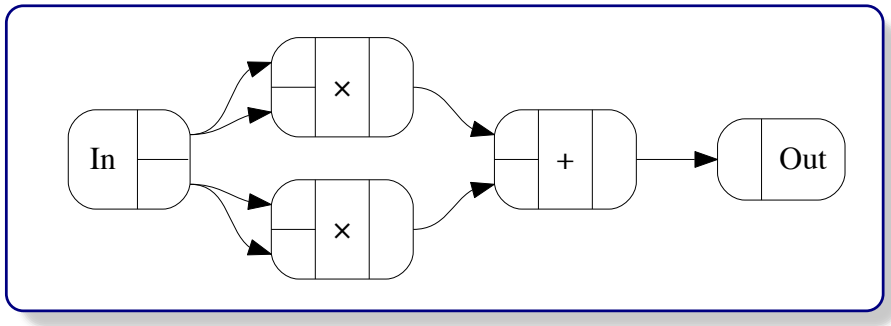


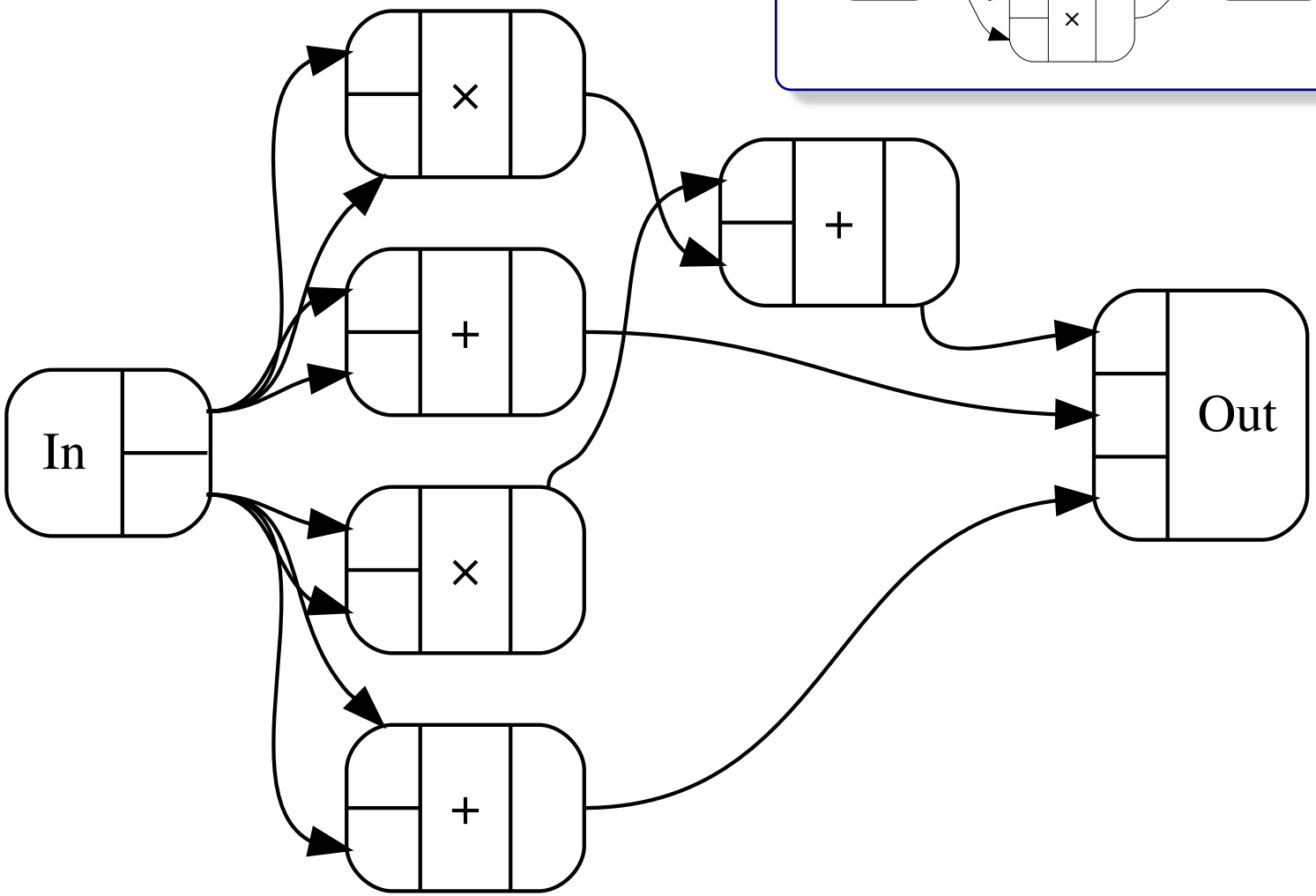
RAD example (dual function)



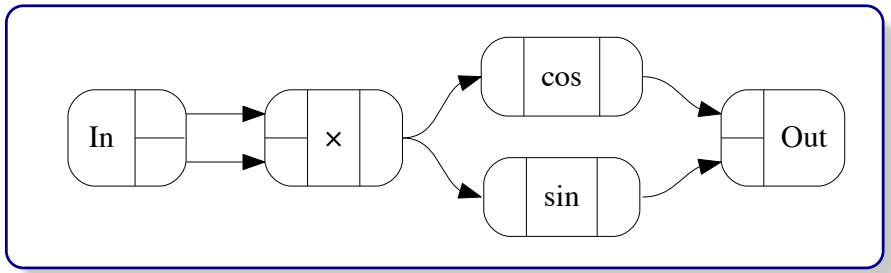


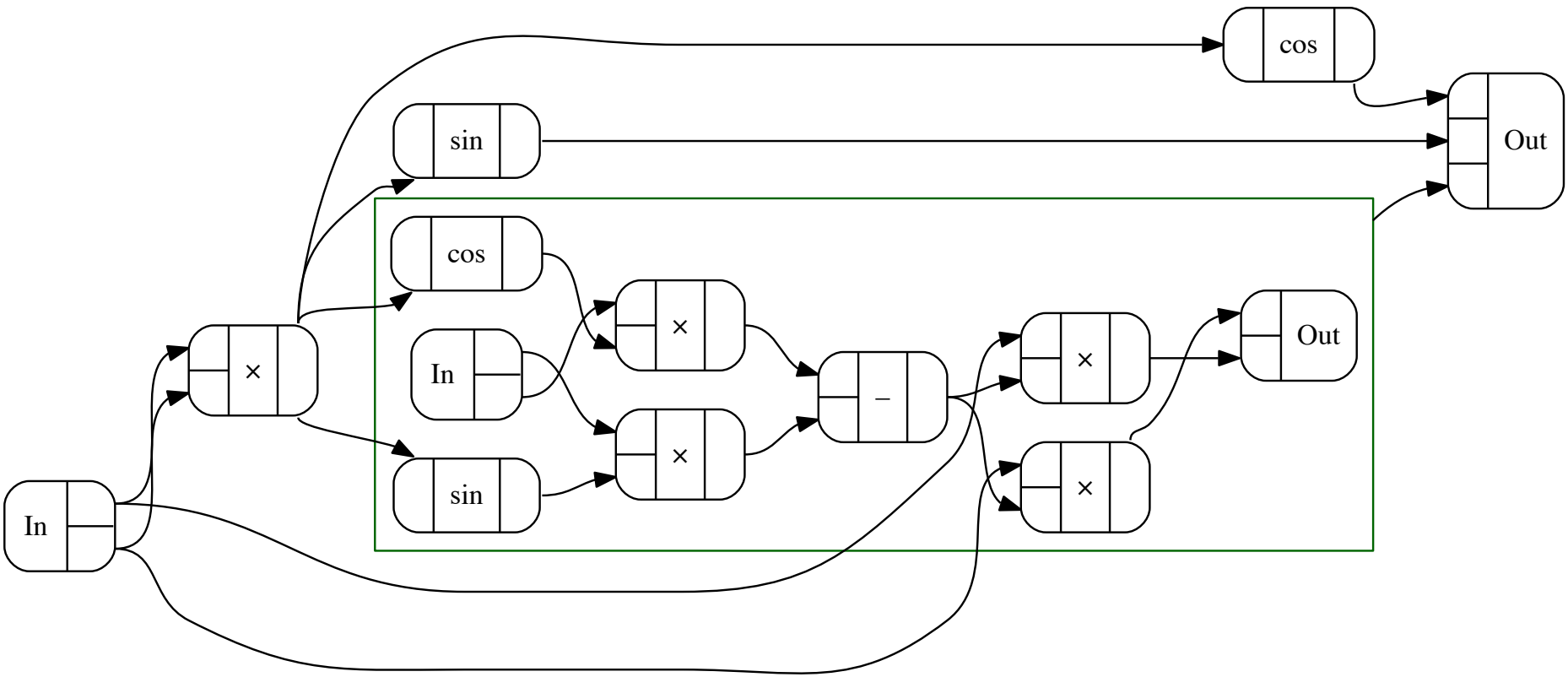
RAD example (dual vector)



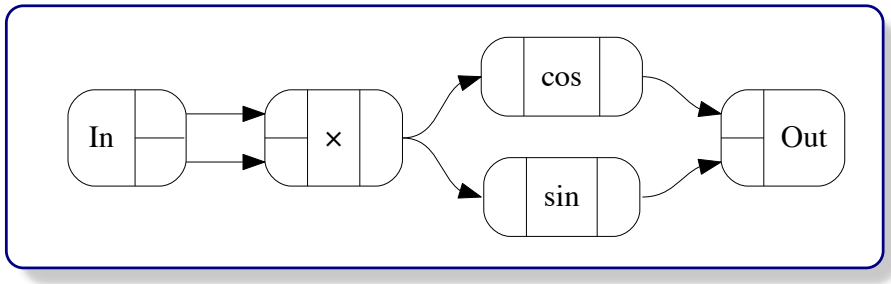


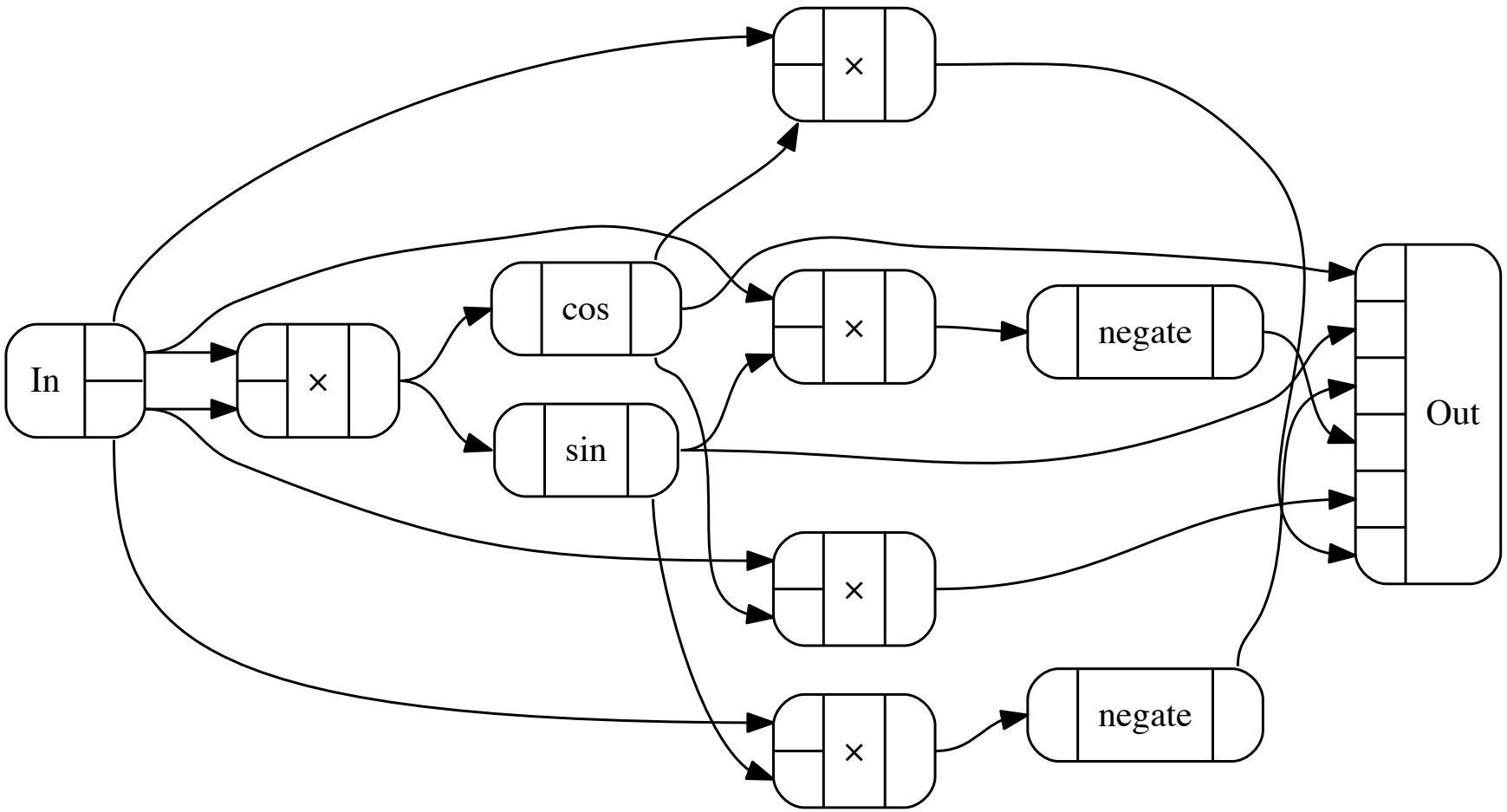
RAD example (dual function)



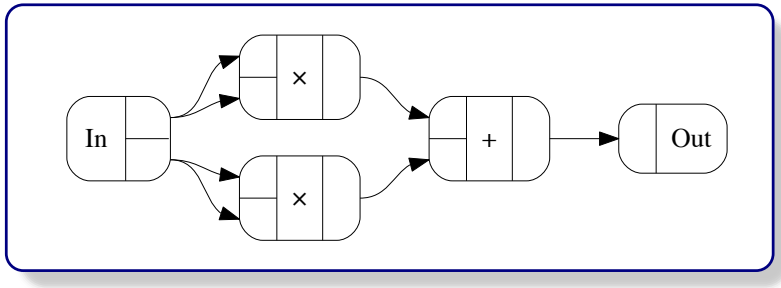


RAD example (matrix)

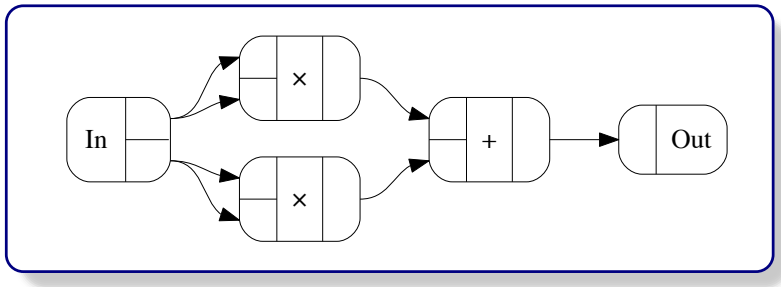


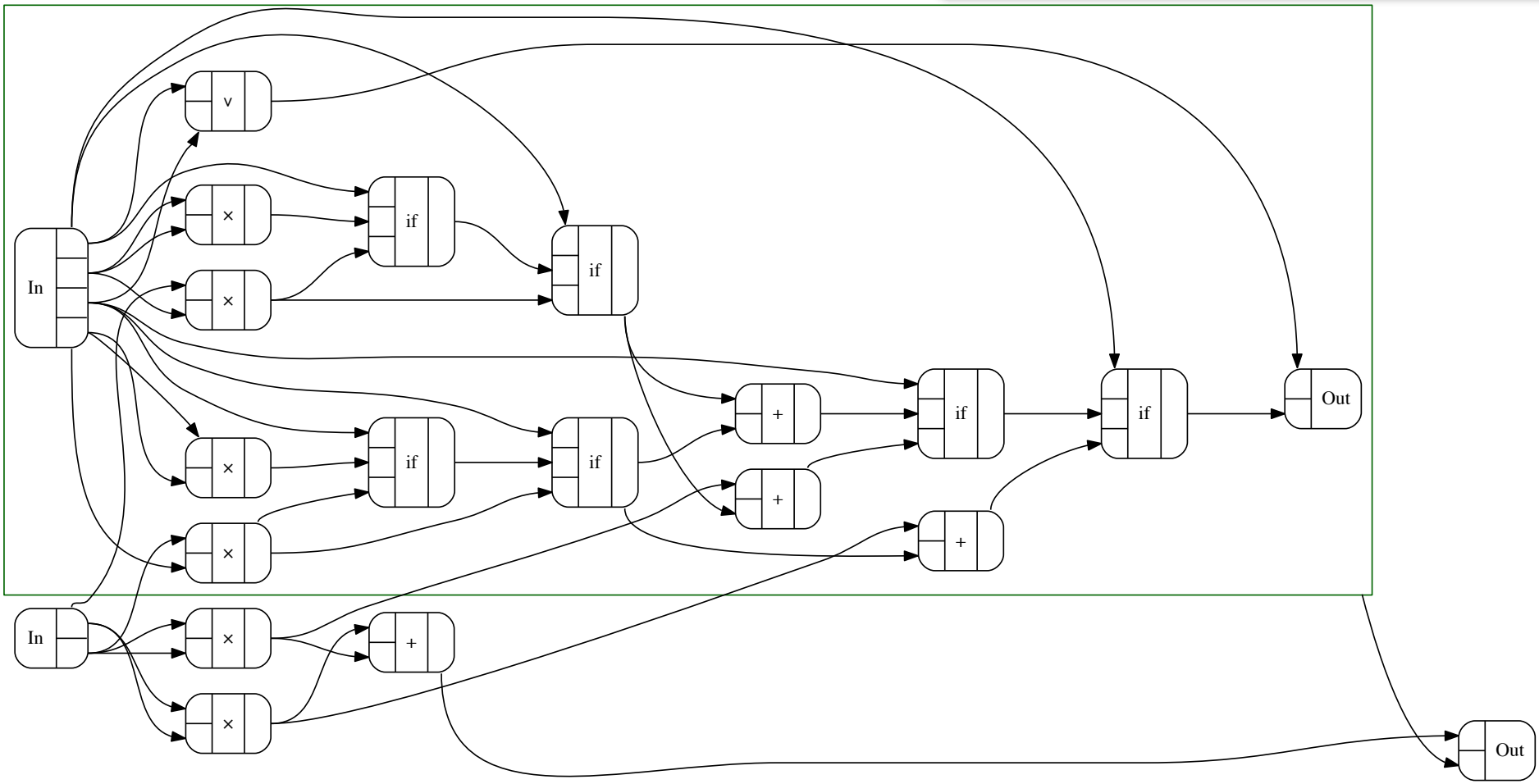


Incremental evaluation



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).
- ICFP 2018 paper: pictures, proofs, incremental computation.

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.

Extra topics

- Examples: derivatives as functions
- Examples: reverse mode
- Incremental evaluation
- Compositional matrices