Simple essence of AD

Artur Ezequiel Nelson

Universidade do Minho

24 de Maio

Index

Definition of Derivative

Definition

A function $f : \mathbb{R} \to \mathbb{R}$ is differentiable at $x \in \mathbb{R}$, if there exists a number $f'(x)$ such that:

$$\lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon} = f'(x)$$

$$\lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon} - f'(x) = 0 \Leftrightarrow \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x) - (\varepsilon \cdot f'(x))}{\varepsilon} = 0$$

Definition of Derivative

Definition

A function $f : \mathbb{R} \to \mathbb{R}$ is differentiable at $x \in \mathbb{R}$, if there exists a number $f'(x)$ such that:

$$\lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon} = f'(x)$$

$$\lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon} - f'(x) = 0 \Leftrightarrow \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x) - (\varepsilon \cdot f'(x))}{\varepsilon} = 0$$

Definition

A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is differentiable at $x \in \mathbb{R}^m$, if there exists a unique linear transformation $\mu : \mathbb{R}^m \to \mathbb{R}^n$ such that:

$$\lim_{\varepsilon \to 0} \frac{\|f(x+\varepsilon) - f(x) - \mu(\varepsilon)\|}{\|\varepsilon\|} = 0$$

Derivate as a linear map

Definition

Let $f :: a \to b$ be a function, where a and b are vectorial spaces that share a common underlying field. The first derivative definition is the following:

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

If we differentiate two times, we have:

$$\mathcal{D}^2 = \mathcal{D} \circ \mathcal{D} :: (a \to b) \to (a \to (a \multimap a \multimap b))$$

Rules for Differentiation - Sequential Composition

Theorem

Let $f :: a \to b$ and $g :: b \to c$ be two functions. Then the derivative of the composition of f and g is:

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

Rules for Differentiation - Sequential Composition

Unfortunately the previous theorem isn't a efficient recipe for composition. As such we will introduce a second derivative definition:

$$\mathcal{D}_0^+ :: (a \to b) \to ((a \to b) \times (a \to (a \multimap b)))$$
$$\mathcal{D}_0^+ \ f = (f, \mathcal{D}\ f)$$

With this, the chain rule will have the following expression:

$\mathcal{D}_0^+ (g \circ f)$

{definition of $\mathcal{D}_0^+$}

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

{theorem and definition of $g \circ f$}

$= (\lambda a \to g(f\ a), \lambda a \to \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a)$

Rules for Differentiation - Sequential Composition

Unfortunately the previous theorem isn't a efficient recipe for composition. As such we will introduce a second derivative definition:

$$\mathcal{D}_0^+ :: (a \to b) \to ((a \to b) \times (a \to (a \multimap b)))$$
$$\mathcal{D}_0^+ \ f = (f, \mathcal{D} \ f)$$

With this, the chain rule will have the following expression:
$\mathcal{D}_0^+ (g \circ f)$
{definition of $\mathcal{D}_0^+$}
$= (g \circ f, \mathcal{D} (g \circ f))$
{theorem and definition of $g \circ f$}
$= (\lambda a \to g(f\, a), \lambda a \to \mathcal{D}\, g\, (f\, a) \circ \mathcal{D}\, f\, a)$

Rules for Differentiation - Sequential Composition

Having in mind optimizations, we introduce the third and last derivative definition:

$$\mathcal{D}^+ :: (a \to b) \to (a \to (b \times (a \multimap b)))$$
$$\mathcal{D}^+\ f\ a = (f\ a, \mathcal{D}\ f\ a)$$

As $\times$ has more priority than $\to$ and $\multimap$, we can rewrite $\mathcal{D}^+$ as:

$$\mathcal{D}^+ :: (a \to b) \to (a \to b \times (a \multimap b))$$
$$\mathcal{D}^+\ f\ a = (f\ a, \mathcal{D}\ f\ a)$$

Rules for Differentiation - Sequential Composition

Having in mind optimizations, we introduce the third and last derivative definition:

$$\mathcal{D}^+ :: (a \to b) \to (a \to (b \times (a \multimap b)))$$
$$\mathcal{D}^+\ f\ a = (f\ a, \mathcal{D}\ f\ a)$$

As $\times$ has more priority than $\to$ and $\multimap$, we can rewrite $\mathcal{D}^+$ as:

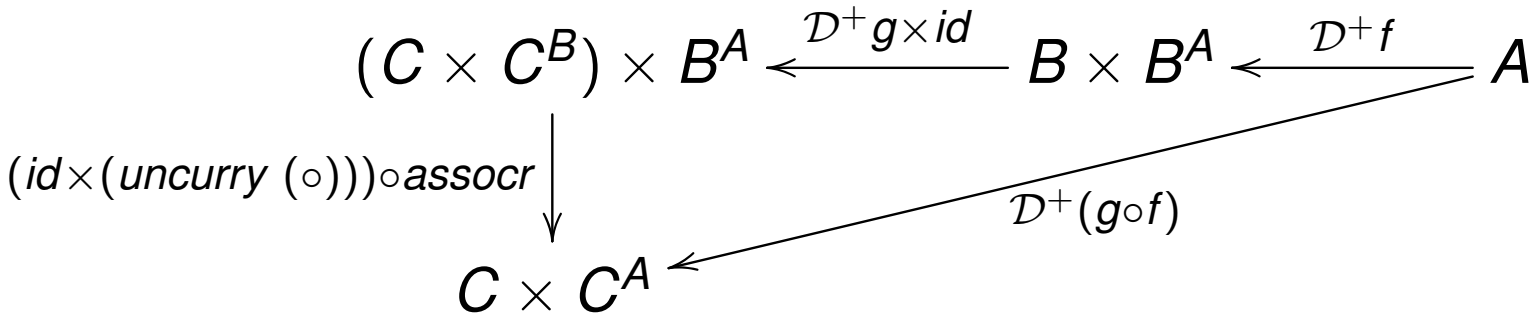
$$\mathcal{D}^+ :: (a \to b) \to (a \to b \times (a \multimap b))$$
$$\mathcal{D}^+\ f\ a = (f\ a, \mathcal{D}\ f\ a)$$

Rules for Differentiation - Sequential Composition

Corollary

$\mathcal{D}^+$ is (efficiently) compositional with respect to $(\circ)$, that is, in Haskell:

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



Rules for Differentiation - Parallel Composition

Another important way of combining functions is the operation cross, that combines two functions in parallel:

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

Theorem

Let $f :: a \to c$ and $g :: b \to d$ be two function. Then the cross rule is the following:

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

Rules for Differentiation - Parallel Composition

Another important way of combining functions is the operation cross, that combines two functions in parallel:

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

Theorem

Let $f :: a \to c$ and $g :: b \to d$ be two function. Then the cross rule is the following:

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

Rules for Differentiation - Parallel Composition

Corollary

The function $\mathcal{D}^+$ is compositional in relation to $(\times)$

$$\mathcal{D}^+\ (f \times g)\ (a, b) = \textbf{let}\ \{(c, f') = \mathcal{D}^+\ f\ a; (d, g') = \mathcal{D}^+\ g\ b\}$$
$$\textbf{in}\ ((c, d), f' \times g')$$

Derivative and Linear Functions

Definition

A function f is said to be linear when preserves addition and scalar multiplication.

$$f(a + a') = f\,a + f\,a'$$
$$f(s \cdot a) = s \cdot f\,a$$

Theorem

For all linear functions f, $\mathcal{D}\,f\,a = f$.

Corollary

For all linear functions f, $\mathcal{D}^+\,f = \lambda a \to (fa, f)$.

Derivative and Linear Functions

Definition

A function f is said to be linear when preserves addition and scalar multiplication.

$$f(a + a') = f\,a + f\,a'$$
$$f(s \cdot a) = s \cdot f\,a$$

Theorem

For all linear functions f, $\mathcal{D}\,f\;a = f$.

Corollary

For all linear functions f, $\mathcal{D}^+\,f = \lambda a \to (fa, f)$.

Derivative and Linear Functions

Definition

A function f is said to be linear when preserves addition and scalar multiplication.

$$f(a + a') = f\,a + f\,a'$$
$$f(s \cdot a) = s \cdot f\,a$$

Theorem

For all linear functions f, $\mathcal{D}\,f\;a = f$.

Corollary

For all linear functions f, $\mathcal{D}^+ f = \lambda a \to (fa, f)$.

A short introdution

- We want to calculate $\mathcal{D}^+$.
- However, $\mathcal{D}$ is not computable.
- Solution: reimplement corollaries using category theory.

A short introdution

- We want to calculate $\mathcal{D}^+$.
- However, $\mathcal{D}$ is not computable.
- Solution: reimplement corollaries using category theory.

A short introdution

- We want to calculate $\mathcal{D}^+$.
- However, $\mathcal{D}$ is not computable.
- Solution: reimplement corollaries using category theory.

A short introdution

Corollary 1.1

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

Corollary 2.1

$$\mathcal{D}^+\ (f \times g)\ (a, b) = \textbf{let}\ \{(c, f') = \mathcal{D}^+\ f\ a; (d, g') = \mathcal{D}^+\ g\ b\}$$
$$\textbf{in}\ ((c, d), f' \times g')$$

Corollary 3.1

For all linear functions f, $\mathcal{D}^+ f = \lambda a \to (fa, f)$.

Categories

A category is a collection of objects (sets and types) and morphisms (operation between objects), with 2 basic operations (identity and composition) of morfisms, and 2 laws:

- (C.1) $id \circ f = f \circ id = f$
- (C.2) $f \circ (g \circ h) = (f \circ g) \circ h$

Note

For this article, objects are data types and morfisms are functions

class *Category* k **where**
$id :: (a\ k\ a)$
$(\circ) :: (b\ k\ c) \to (a\ k\ b) \to (a\ k\ c)$

instance *Category* $(\to)$ **where**
$id = \lambda a \to a$
$g \circ f = \lambda a \to g\ (f\ a)$

Functors

A functor F between 2 categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

- given any object t $\in \mathcal{U}$ there exists an object F t $\in \mathcal{V}$
- given any morphism m :: a $\rightarrow$ b $\in \mathcal{U}$ there exists a morphism F m :: F a $\rightarrow$ F b $\in \mathcal{V}$
- F id ($\in \mathcal{U}$) = id ($\in \mathcal{V}$)
- F (f $\circ$ g) = F f $\circ$ F g

Note

Given this category properties (objects are data types) functors map types to themselves

Objective

$\mathcal{D}$ definition

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

Adapted definition for $\mathcal{D}$ type

$$\hat{\mathcal{D}} :: (a \to b) \to \mathcal{D}\ a\ b$$
$$\hat{\mathcal{D}}\ f = \mathcal{D}\ (\mathcal{D}^{+}\ f)$$

Our objective is to deduce an instance of a Category for $\mathcal{D}$ where $\hat{\mathcal{D}}$ is a functor.

Instance deduction

Using corollaries 3.1 and 1.1 we can determine that

- (DP.1) $\mathcal{D}^+\ id = \lambda a \to (id\ a, id)$
- (DP.2)

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

Saying that $\hat{\mathcal{D}}$ is a functor is equivalent to, for all f and g functions of appropriate types:
$id = \hat{\mathcal{D}}\ id = \mathcal{D}\ (\mathcal{D}^+\ id)$
$\hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f = \hat{\mathcal{D}}\ (g \circ f) = \mathcal{D}\ (\mathcal{D}^+\ (g \circ f))$

Instance deduction

Based on (DP.1) and (DP.2) we'll rewrite the above into the following definition:
$id = \mathcal{D}\ (\lambda a \to (id\ a, id))$
$\hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f = \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = \mathcal{D}^+\ f\ a; (c, g') = \mathcal{D}^+\ g\ b\}\ \textbf{in}\ (c, g' \circ f'))$

The first equation shown above has a trivial solution.

To solve the second we'll first solve a more general one:
$\mathcal{D}\ g \circ \mathcal{D}\ f = \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))$
This condition also leads us to a trivial solution inside our instance.

Instance deduction

Based on (DP.1) and (DP.2) we'll rewrite the above into the following definition:
$id = \mathcal{D}\ (\lambda a \to (id\ a, id))$
$\hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f = \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = \mathcal{D}^+\ f\ a; (c, g') = \mathcal{D}^+\ g\ b\}\ \textbf{in}\ (c, g' \circ f'))$

The first equation shown above has a trivial solution.

To solve the second we'll first solve a more general one:
$\mathcal{D}\ g \circ \mathcal{D}\ f = \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))$
This condition also leads us to a trivial solution inside our instance.

Instance deduction

Based on (DP.1) and (DP.2) we'll rewrite the above into the following definition:
$id = \mathcal{D}\ (\lambda a \to (id\ a, id))$
$\hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f = \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = \mathcal{D}^+\ f\ a; (c, g') = \mathcal{D}^+\ g\ b\}\ \textbf{in}\ (c, g' \circ f'))$

The first equation shown above has a trivial solution.

To solve the second we'll first solve a more general one:
$\mathcal{D}\ g \circ \mathcal{D}\ f = \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))$
This condition also leads us to a trivial solution inside our instance.

Instance deduction

$\hat{\mathcal{D}}$ definition for linear functions

$$
\begin{aligned}
&\mathit{linearD} :: (a \to b) \to \mathcal{D}\ a\ b \\
&\mathit{linearD}\ f = \mathcal{D}\ (\lambda a \to (f\ a, f))
\end{aligned}
$$

Categorical instance we've deduced

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

Instance proof

In order to prove that the instance is correct we must check if it follows laws (C.1) and (C.2).
First we must make a concession: that we only use morfisms arising from $\mathcal{D}^+$. If we do, then $\mathcal{D}^+$ is a functor.

(C.1) proof

id $\circ$ $\hat{\mathcal{D}}$ f
{ functor law for id (specification of $\hat{\mathcal{D}}$) }
= $\hat{\mathcal{D}}$ id $\circ$ $\hat{\mathcal{D}}$ f
{ functor law for ($\circ$) }
= $\hat{\mathcal{D}}$ (id $\circ$ f)
{ categorical law }
= $\hat{\mathcal{D}}$ f

Instance proof

(C.2) proof

$\hat{\mathcal{D}}$ h $\circ$ ($\hat{\mathcal{D}}$ g $\circ$ $\hat{\mathcal{D}}$ f)
{ 2x functor law for ($\circ$) }
= $\hat{\mathcal{D}}$ (h $\circ$ (g $\circ$ f))
{ categorical law }
= $\hat{\mathcal{D}}$ ((h $\circ$ g) $\circ$ f)
{ 2x functor law for ($\circ$) }
= ($\hat{\mathcal{D}}$ h $\circ$ $\hat{\mathcal{D}}$ g) $\circ$ $\hat{\mathcal{D}}$ f

Note

Those proofs don't require anything from $\mathcal{D}$ *and* $\hat{\mathcal{D}}$ aside from functor laws. As such, all other instances of categories created from a functor won't require further proving like this one did.

Monoidal categories and functors

Generalized parallel composition shall be defined using a monoidal category:

$$\textbf{class } \mathit{Category}\ k \Rightarrow \mathit{Monoidal}\ k \textbf{ where}$$
$$(\times) :: (a\ \texttt{`k`}\ c) \to (b\ \texttt{`k`}\ d) \to ((a \times b)\ \texttt{`k`}\ (c \times d))$$
$$\textbf{instance } \mathit{Monoidal}\ (\to) \textbf{ where}$$
$$f \times g = \lambda(a, b) \to (f\ a, g\ b)$$

Monoidal Functor definition

A monoidal functor F between categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

- F is a functor
- F (f $\times$ g) = F f $\times$ F g

Instance deduction

From corollary 2.1 we can deduce that:
$\mathcal{D}^+ (f \times g) = \lambda(a, b) \to \textbf{let } \{(c, f') = \mathcal{D}^+ \ f \ a; (d, g') = \mathcal{D}^+ \ g \ b\}$
$\textbf{in } ((c, d), f' \times g')$

Deriving F from $\hat{\mathcal{D}}$ leaves us with the following definition:
$\mathcal{D} \ (\mathcal{D}^+ \ f) \times \mathcal{D} \ (\mathcal{D}^+ \ g) = \mathcal{D} \ (\mathcal{D}^+ \ (f \times g))$

Using the same method as before, we replace $\mathcal{D}^+$ with its definition and generalize the condition:
$\mathcal{D} \ f \times \mathcal{D} \ g =$
$\mathcal{D} \ (\lambda(a, b) \to \textbf{let } \{(c, f') = f \ a; (d, g') = g \ b\} \textbf{ in } ((c, d), f' \times g'))$
and this is enough for our new instance.

Instance deduction

From corollary 2.1 we can deduce that:
$\mathcal{D}^+ (f \times g) = \lambda(a, b) \to \textbf{let } \{(c, f') = \mathcal{D}^+ \ f \ a; (d, g') = \mathcal{D}^+ \ g \ b\}$
$\textbf{in } ((c, d), f' \times g')$

Deriving F from $\hat{\mathcal{D}}$ leaves us with the following definition:
$\mathcal{D} \ (\mathcal{D}^+ \ f) \times \mathcal{D} \ (\mathcal{D}^+ \ g) = \mathcal{D} \ (\mathcal{D}^+ \ (f \times g))$

Using the same method as before, we replace $\mathcal{D}^+$ with its definition and generalize the condition:
$\mathcal{D} \ f \times \mathcal{D} \ g =$
$\mathcal{D} \ (\lambda(a, b) \to \textbf{let } \{(c, f') = f \ a; (d, g') = g \ b\} \textbf{ in } ((c, d), f' \times g'))$
and this is enough for our new instance.

Instance deduction

From corollary 2.1 we can deduce that:
$\mathcal{D}^+ (f \times g) = \lambda(a, b) \to \textbf{let } \{(c, f') = \mathcal{D}^+ \ f \ a; (d, g') = \mathcal{D}^+ \ g \ b\}$
$\textbf{in } ((c, d), f' \times g')$

Deriving F from $\hat{\mathcal{D}}$ leaves us with the following definition:
$\mathcal{D} \ (\mathcal{D}^+ \ f) \times \mathcal{D} \ (\mathcal{D}^+ \ g) = \mathcal{D} \ (\mathcal{D}^+ \ (f \times g))$

Using the same method as before, we replace $\mathcal{D}^+$ with its definition and generalize the condition:
$\mathcal{D} \ f \times \mathcal{D} \ g =$
$\mathcal{D} \ (\lambda(a, b) \to \textbf{let } \{(c, f') = f \ a; (d, g') = g \ b\} \textbf{ in } ((c, d), f' \times g'))$
and this is enough for our new instance.

Instance deduction

Categorical instance we've deduced

instance *Monoidal* $\mathcal{D}$ **where**

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

Cartesian categories and functors

```
class Monoidal k ⇒ Cartesian k where
  exl :: (a, b) `k` a
  exr :: (a, b) `k` b
  dup :: a `k` (a, a)
instance Cartesian (→) where
  exl = λ(a, b) → a
  exr = λ(a, b) → b
  dup = λa → (a, a)
```

A cartesian functor F between categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

- F is a monoidal functor
- F exl = exl
- F exp = exp
- F dup = dup

Instance deduction

From corollary 3.1 and from exl, exr and dup being linear functions we can deduce that:

$\mathcal{D}^+\ exl = \lambda p \to (exl\ p, exl)$

$\mathcal{D}^+\ exr = \lambda p \to (exr\ p, exr)$

$\mathcal{D}^+\ dup = \lambda p \to (dup\ a, dup)$

With this in mind we can arrive at our instance:

$exl = \mathcal{D}\ (\mathcal{D}^+\ exl)$

$exr = \mathcal{D}\ (\mathcal{D}^+\ exr)$

$dup = \mathcal{D}\ (\mathcal{D}^+\ dup)$

Instance deduction

Replacing $\mathcal{D}^+$ with its definition and remembering linearD's definition we can obtain:

exl = linearD exl
exr = linearD exr
dup = linearD dup

and convert this directly into a new instance:

Categorical instance we've deduced

instance *Cartesian D* **where**
  exl = *linearD exl*
  exr = *linearD exr*
  dup = *linearD dup*

Cocartesian category

This type of categories is the dual of the cartesian type of categories.

Note

In this article coproducts are categorical products, i.e., biproducts

Definition

class *Category k* $\Rightarrow$ *Cocartesian k* **where**
inl :: *a* ′ `k′` (*a*, *b*)
inr :: *b* ′ `k′` (*a*, *b*)
jam :: (*a*, *a*) ′ `k′` *a*

Cocartesian functors

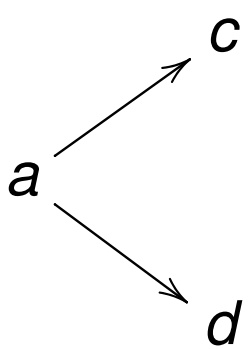
Cocartesian functor definition

A cocartesian functor F between categories $\mathcal{U}$ *and* $\mathcal{V}$ is such that:

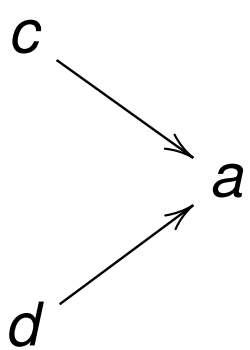
- F is a functor
- F inl = inl
- F inr = inr
- F jam = jam

Fork and Join

- $\Delta :: \mathit{Cartesian}\ k \Rightarrow (a\ \mathtt{`k`}\ c) \rightarrow (a\ \mathtt{`k`}\ d) \rightarrow (a\ \mathtt{`k`}\ (c \times d))$



- $\nabla :: \mathit{Cartesian}\ k \Rightarrow (c\ \mathtt{`k`}\ a) \rightarrow (d\ \mathtt{`k`}\ a) \rightarrow ((c \times d)\ \mathtt{`k`}\ a)$



Instance of $\to^+$

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

$$\textbf{instance}\ \mathit{Category}\ (\to^+)\ \textbf{where}$$

$$\quad \textbf{type}\ \mathit{Obj}\ (\to^+) = \mathit{Additive}$$

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

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

$$\textbf{instance}\ \mathit{Monoidal}\ (\to^+)\ \textbf{where}$$

$$\quad \mathit{AddFun}\ f \times \mathit{AddFun}\ g = \mathit{AddFun}\ (f \times g)$$

$$\textbf{instance}\ \mathit{Cartesian}\ (\to^+)\ \textbf{where}$$

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

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

$$\quad \mathit{dup} = \mathit{AddFun}\ \mathit{dup}$$

Instance of $\rightarrow^+$

instance *Cocartesian* $(\rightarrow^+)$ **where**

$$
\begin{aligned}
&\quad \mathit{inl} = \mathit{AddFun}\ \mathit{inlF} \\
&\quad \mathit{inr} = \mathit{AddFun}\ \mathit{inrF} \\
&\quad \mathit{jam} = \mathit{AddFun}\ \mathit{jamF}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inlF} :: \mathit{Additive}\ b \Rightarrow a \rightarrow a \times b \\
&\mathit{inrF} :: \mathit{Additive}\ a \Rightarrow b \rightarrow a \times b \\
&\mathit{jamF} :: \mathit{Additive}\ a \Rightarrow a \times a \rightarrow a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inlF} = \lambda a \rightarrow (a, 0) \\
&\mathit{inrF} = \lambda b \rightarrow (0, b) \\
&\mathit{jamF} = \lambda (a, b) \rightarrow a + b
\end{aligned}
$$

NumCat definition

class *NumCat k a* **where**
  $negateC :: a \; `k` \; a$
  $addC :: (a \times a) \; `k` \; a$
  $mulC :: (a \times a) \; `k` \; a$
  ...

instance *Num a* $\Rightarrow$ *NumCat* $(\rightarrow)$ *a* **where**
  $negateC = negate$
  $addC = uncurry \; (+)$
  $mulC = uncurry \; (*)$
  ...

$$\mathcal{D}\ (\mathit{negate}\ u) = \mathit{negate}\ (\mathcal{D}\ u)$$
$$\mathcal{D}\ (u + v) = \mathcal{D}\ u + \mathcal{D}\ v$$
$$\mathcal{D}\ (u * v) = u * \mathcal{D}\ v + v * \mathcal{D}\ u$$

- Imprecise on the nature of u and v.
- A precise and simpler definition would be to differentiate the operations themselves.

$$\textbf{class } \mathit{Scalable}\ k\ a \textbf{ where}$$
$$\quad \mathit{scale} :: a \to (a\ \text{`}\ k\ \text{`}\ a)$$
$$\textbf{instance } \mathit{Num}\ a \Rightarrow \mathit{Scalable}\ (\to^{+})\ a \textbf{ where}$$
$$\quad \mathit{scale}\ a = \mathit{AddFun}\ (\lambda da \to a * da)$$
$$\textbf{instance } \mathit{NumCat}\ D \textbf{ where}$$
$$\quad \mathit{negateC} = \mathit{linearD}\ \mathit{negateC}$$
$$\quad \mathit{addC} = \mathit{linearD}\ \mathit{addC}$$
$$\quad \mathit{mulC} = D\ (\lambda(a, b) \to (a * b, \mathit{scale}\ b\ \nabla\ \mathit{scale}\ a))$$
$$\textbf{instance } \mathit{FloatingCat}\ D \textbf{ where}$$
$$\quad \mathit{sinC} = D\ (\lambda a \to (\mathit{sin}\ a, \mathit{scale}\ (\mathit{cos}\ a)))$$
$$\quad \mathit{cosC} = D\ (\lambda a \to (\mathit{cos}\ a, \mathit{scale}\ (-\mathit{sin}\ a)))$$
$$\quad \mathit{expC} = D\ (\lambda a \to \textbf{let}\ e = \mathit{exp}\ a\ \textbf{in}\ (e, \mathit{scale}\ e))$$
$$\quad \ldots$$

Examples

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

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

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

With a compiler plugin we can obtain

$$sqr = mulC \circ (id\ \Delta\ id)$$
$$magSqr = addC \circ (mulC \circ (exl\ \Delta\ exl)\ \Delta\ mulC \circ (exr\ \Delta\ exr))$$
$$cosSinProd = (cosC\ \Delta\ sinC) \circ mulC$$

Generalizing Automatic Differentiation

$$\textbf{newtype}\ D_k\ a\ b = D\ (a \to b \times (a\ `k`\ b))$$

$$linearD :: (a \to b) \to (a\ `k`\ b) \to D_k\ a\ b$$
$$linearD\ f\ f' = D\ (\lambda a \to (f\ a, f'))$$

$$\textbf{instance}\ Category\ k \Rightarrow Category\ D_k\ \textbf{where}$$
$$\quad \textbf{type}\ Obj\ D_k = Additive \wedge Obj\ k\ ...$$

$$\textbf{instance}\ Monoidal\ k \Rightarrow Monoidal\ D_k\ \textbf{where}\ ...$$

$$\textbf{instance}\ Cartesian\ k \Rightarrow Cartesian\ D_k\ \textbf{where}\ ...$$

$$\textbf{instance}\ Cocartesian\ k \Rightarrow Cocartesian\ D_k\ \textbf{where}$$
$$\quad inl = linearD\ inlF\ inl$$
$$\quad inr = linearD\ inrF\ inr$$
$$\quad jam = linearD\ jamF\ jam$$

instance *Scalable k s* $\Rightarrow$ *NumCat* D_k *s* **where**

$$negateC = linearD\ negateC\ negateC$$

$$addC = linearD\ addC\ addC$$

$$mulC = D\ (\lambda(a, b) \rightarrow (a * b, scale\ b\ \nabla\ scale\ a))$$

Matrices

There exists three, non-exclusive, possibilities for a nonempty matrix W:

- width W = height W = 1;
- W is the horizontal juxtaposition of two matrices U and V, where height W = height U = height V and width W = width U + width V;
- W is the vertical juxtaposition of two matrices U and V, where width W = width U = width V and height W = height U + height V.

Extracting a Data Representation

In machine learning, a Gradient-based optimization works by searching for local minima in the domain of a differentiable function $f :: a \to s$. Each step in the search is in the direction opposite of the gradient of f, which is a vector form of $\mathcal{D}\ f$.

Given a linear map $f' :: U \multimap V$ represented as a function, it is possible to extract a Jacobian matrix by applying f to every vector in a basis of U.

Extracting a Data Representation

In machine learning, a Gradient-based optimization works by searching for local minima in the domain of a differentiable function $f :: a \to s$. Each step in the search is in the direction opposite of the gradient of f, which is a vector form of $\mathcal{D}\ f$.

Given a linear map $f' :: U \multimap V$ represented as a function, it is possible to extract a Jacobian matrix by applying f to every vector in a basis of U.

Generalized Matrices

Given a scalar field s, a free vector space has the form $p \to s$ for some p, where the cardinality of p is the dimension of the vector space and there exists a finite number of values for p.

In particular, we can represent vector spaces over a given field as a representable functor, i.e., a functor F such that:

$$\exists p \, \forall s \; F \; s \;\cong\; p \to s$$

Generalized Matrices

Given a scalar field s, a free vector space has the form $p \to s$ for some p, where the cardinality of p is the dimension of the vector space and there exists a finite number of values for p.

In particular, we can represent vector spaces over a given field as a representable functor, i.e., a functor F such that:

$$\exists p \, \forall s \; F \; s \; \cong \; p \to s$$

A short introdution

- We've derived and generalized an AD algorithm using categories.
- With fully right-associated compositions this algorithm becomes a foward-mode AD and with fully left-associated becomes a reverse-mode AD.
- We want to obtain generalized FAD and RAD algorithms.
- How do we describe this in Categorical notation?

Converting morfisms

Given a category k we can represent its morfisms the following way:

Left-Compose functions

$f :: a\ \texttt{`k`}\ b \Rightarrow (\circ f) :: (b\ \texttt{`k`}\ r) \rightarrow (a\ \texttt{`k`}\ r)$ where r is any object of k.

If h is the morfism we'll compose with f then h is the continuation of f.

Instance deduction

Defining new type

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

Functor derived from type

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

Instance deduction

instance $Category\ k \Rightarrow Category\ Cont^r_k$ **where**
$id = Cont\ id$
$Cont\ g \circ Cont\ f = Cont\ (f \circ g)$

instance $Monoidal\ k \Rightarrow Monoidal\ Cont^r_k$ **where**
$Conf\ f \times Cont\ g = Cont\ (join \circ (f \times g) \circ unjoin)$

instance $Cartesian\ k \Rightarrow Cartesian\ Cont^r_k$ **where**
$exl = Cont\ (join \circ inl); exr = Cont\ (join \circ inr)$
$dup = Cont\ (jam \circ unjoin)$

instance $Cocartesian\ k \Rightarrow Cocartesian\ Cont^r_k$ **where**
$inl = Cont\ (exl \circ unjoin); inr = Cont\ (exr \circ unjoin)$
$jam = Cont\ (join \circ dup)$

instance $Scalable\ k\ a \Rightarrow Scalable\ Cont^r_k\ a$ **where**
$scale\ s = Cont\ (scale\ s)$

A short introdution

Due to its widespread use in ML we'll talk about a specific case of RAD: computing gradients (derivatives of functions with scalar codomains).
A vector space A over a scalar field s has A $\multimap$ s as its dual.
Each linear map in A $\multimap$ s can be represented in the form of dot u for some u :: A where

Definition and instanciation

$$\textbf{class}\ \mathit{HasDot}\ (S)\ u\ \textbf{where}\ \mathit{dot} :: u \to (u \multimap s)$$

$$\textbf{instance}\ \mathit{HasDot}\ (\mathit{IR})\ \mathit{IR}\ \textbf{where}\ \mathit{dot} = \mathit{scale}$$

$$\begin{aligned}&\textbf{instance}\ (\mathit{HasDot}\ (S)\ a, \mathit{HasDot}\ (S)\ b)\\ &\quad \Rightarrow \mathit{HasDot}\ (S)\ (a \times b)\\ &\quad \textbf{where}\ \mathit{dot}\ (u, v) = \mathit{dot}\ u\ \Delta\ \mathit{dot}\ v\end{aligned}$$

Instance deduction

The internal representation of $Cont^{s}_{\multimap}$ a b is (b $\multimap$ s) $\to$ (a $\multimap$ s) which is isomorfic to (a $\to$ b).

Type definition for duality

newtype $Dual_k\ a\ b = Dual\ (b\ `\mathtt{k}`\ a)$

Instance deduction

All we need to do to create dual representations of linear maps is to convert from $Cont_k^S$ to $Dual_k$ using a functor:

Functor definition

$$
\begin{aligned}
&asDual :: (HasDot\ (S)\ a, HasDot\ (S)\ b) \Rightarrow \\
&\quad ((b\ \multimap\ s) \to (a\ \multimap\ s)) \to (b\ \multimap\ a) \\
&asDual\ (Cont\ f) = Dual\ (onDot\ f)
\end{aligned}
$$

where

$$
\begin{aligned}
&onDot :: (HasDot\ (S)\ a, HasDot\ (S)\ b) \Rightarrow \\
&\quad ((b\ \multimap\ s) \to (a\ \multimap\ s)) \to (b\ \multimap\ a) \\
&onDot\ f = dot^{-1} \circ f \circ dot
\end{aligned}
$$

Instance deduction

instance *Category k* $\Rightarrow$ *Category Dual*$_k$ **where**
  id $=$ *Dual id*
  Dual g $\circ$ *Dual f* $=$ *Dual* $(f \circ g)$

instance *Monoidal k* $\Rightarrow$ *Monoidal Dual*$_k$ **where**
  Dual f $\times$ *Dual g* $=$ *Dual* $(f \times g)$

instance *Cartesian k* $\Rightarrow$ *Cartesian Dual*$_k$ **where**
  exl $=$ *Dual inl*; *exr* $=$ *Dual inr*
  dup $=$ *Dual jam*

instance *Cocartesian k* $\Rightarrow$ *Cocartesian Dual*$_k$ **where**
  inl $=$ *Dual exl*; *inr* $=$ *Dual exr*
  jam $=$ *Dual dup*

instance *Scalable k* $\Rightarrow$ *Scalable Dual*$_k$ **where**
  scale s $=$ *Dual* (*scale s*)

Final notes

- (∇) *and* (Δ) mutually dualize
  $(\mathit{Dual}\ f\ \nabla\ \mathit{Dual}\ g) = \mathit{Dual}\ (f\ \Delta\ g)$ *and* $\mathit{Dual}\ f\ \Delta\ \mathit{Dual}\ g = \mathit{Dual}\ (f\ \nabla\ g))$
- Using the definition from chapter 8 we can determine that the duality of a matrix corresponds to its transposition

Fowards-mode Automatic Differentiation(FAD)

We can use the same deductions we've done in Cont and Dual to derive a category with full right-side association, thus creating a generized FAD algorithm.
This algorithm is far more appropriate for low dimension domains.

Type definition and functor from type

$$\textbf{newtype}\ \mathit{Begin}^r_k\ a\ b = \mathit{Begin}\ ((r\ \mathtt{'k'}\ a) \to (r\ \mathtt{'k'}\ b))$$

$$\mathit{begin} :: \mathit{Category}\ k \Rightarrow (a\ \mathtt{'k'}\ b) \to \mathit{Begin}^r_k\ a\ b$$
$$\mathit{begin}\ f = \mathit{Begin}\ (f\circ)$$

We can derive categorical instances from the functor above and we can choose r to be the scalar field s, noting that s $\multimap$ a is isomorfic to a.

Scaling Up

- Practical applications often involve high-dimensional spaces.
- Binary products are a very inefficient and unwieldy way of encoding high-dimensional spaces.
- A practical alternative is to consider n-ary products as representable functors.

```
class Category k ⇒ Monoidall k h where
  crossl :: h (a ‘ k ‘ b) → (h a ‘ k ‘ h b)
instance Zip h ⇒ Monoidall (→) h where
  crossl = zipWith id
```

```
class MonoidalI k h ⇒ CartesianI k h where
  exI  :: h (h a ‘ k ‘ a)
  replI :: a ‘ k ‘ h a

instance (Representable h, Zip h, Pointed h) ⇒
  CartesianI (→) h where
  exI = tabulate (flip index)
  replI = point
```

- The following is not the class the author was thinking

```
class Representable h where
  type Rep h :: ∗
  tabulate :: (Rep h → a) → h a
  index :: h a → Rep h → a
```

```
class MonoidalI k h ⇒ CocartesianI k h where
  inI :: h (a ‘ k ‘ h a)
  jamI :: h a ‘ k ‘ a

instance (MonoidalI k h, Zip h) ⇒ MonoidalI D_k h where
  crossI fs = D ((id × crossI) ∘ unzip ∘ crossI (fmap unD fs))

instance (CocartesianI (→) h, CartesianI k h, Zip h) ⇒
  CartesianI D_k h where
  exI = linearD exI exI
  replI = zipWith linearD replI replI

instance (CocartesianI k h, Zip h) ⇒ CocartesianI D_k h where
  inI = zipWith linearD inIF inI
  jamI = linearD sum jamI
```

```
class MonoidalI k h ⇒ CocartesianI k h where
  inI :: h (a ‘ k ‘ h a)
  jamI :: h a ‘ k ‘ a

instance (MonoidalI k h, Zip h) ⇒ MonoidalI D_k h where
  crossI fs = D ((id × crossI) ∘ unzip ∘ crossI (fmap unD fs))

instance (CocartesianI (→) h, CartesianI k h, Zip h) ⇒
  CartesianI D_k h where
  exI = zipWith linearD exI exI
  replI = linearD replI replI

instance (CocartesianI k h, Zip h) ⇒ CocartesianI D_k h where
  inI = zipWith linearD inIF inI
  jamI = linearD sum jamI
```

Conclusion

- Suggests that some of the work referred to does just a part of this article.
- This article ([Elliott 2018][2]) is a follow up of [Elliott 2017][1]
- Suggests that this implementation is simple, efficient, it can free memory dinamically (RAD) and is naturally parallel.
- Future work are detailed performace analysis; higher-order differentiation and automatic incrementation (continuing previous work [Elliott 2017][1])


ELLIOTT, C.
Compiling to categories.
Proc. ACM Program. Lang. 1, ICFP (Aug. 2017), 27:1–27:27.


ELLIOTT, C.
The simple essence of automatic differentiation.
Proc. ACM Program. Lang. 2, ICFP (July 2018), 70:1–70:29.