

# Simple essence of AD

Artur Ezequiel Nelson

Universidade do Minho

24 de Maio

# Index

- 1 Categories
- 2 Fork and Join
- 3 Numeric operations
- 4 Generalizing Automatic Differentiation
- 5 Reverse-Mode Automatic Differentiation(RAD)
- 6 Gradient and Duality
- 7 Forward-Mode Automatic Differentiation(FAD)
- 8 Scaling Up
- 9 Related Work and conclusion
- 10 Bibliography

# Definition of Derivative

## Definition

Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  be a function. The derivative of  $f$  at point  $x \in \mathbb{R}$  is defined the following way:

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

This definition will also work with functions of types  $\mathbb{C} \rightarrow \mathbb{C}$  and  $\mathbb{R} \rightarrow \mathbb{R}^n$ .

# Definition of Derivative

## Definition

Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  be a function. The derivative of  $f$  at point  $x \in \mathbb{R}$  is defined the following way:

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

This definition will also work with functions of types  $\mathbb{C} \rightarrow \mathbb{C}$  and  $\mathbb{R} \rightarrow \mathbb{R}^n$ .

# Definition of Derivative

For functions  $F$  of types  $\mathbb{R}^m \rightarrow \mathbb{R}$  and  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  (with  $n > 1$ ), we need a different definition.

- For functions of type  $\mathbb{R}^m \rightarrow \mathbb{R}$ , it is necessary the introduction of the notion of partial derivatives,  $\frac{\partial F}{\partial x_j}$ , with  $j \in \{1, \dots, m\}$ .
- For functions of type  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  (with  $n > 1$ ), apart from the use of partial derivatives, it is necessary the use of Jacobian matrices  $\mathbf{J}_{i,j} = \frac{\partial F_i}{\partial x_j}$ , where  $i \in \{1, \dots, n\}$  and  $F_i$  is a function  $\mathbb{R}^m \rightarrow \mathbb{R}$ .

# Definition of Derivative

For functions  $F$  of types  $\mathbb{R}^m \rightarrow \mathbb{R}$  and  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  (with  $n > 1$ ), we need a different definition.

- For functions of type  $\mathbb{R}^m \rightarrow \mathbb{R}$ , it is necessary the introduction of the notion of partial derivatives,  $\frac{\partial F}{\partial x_j}$ , with  $j \in \{1, \dots, m\}$ .
- For functions of type  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  (with  $n > 1$ ), apart from the use of partial derivatives, it is necessary the use of Jacobian matrices  $\mathbf{J}_{i,j} = \frac{\partial F_i}{\partial x_j}$ , where  $i \in \{1, \dots, n\}$  and  $F_i$  is a function  $\mathbb{R}^m \rightarrow \mathbb{R}$ .

# Definition of Derivative

For functions  $F$  of types  $\mathbb{R}^m \rightarrow \mathbb{R}$  and  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  (with  $n > 1$ ), we need a different definition.

- For functions of type  $\mathbb{R}^m \rightarrow \mathbb{R}$ , it is necessary the introduction of the notion of partial derivatives,  $\frac{\partial F}{\partial x_j}$ , with  $j \in \{1, \dots, m\}$ .
- For functions of type  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  (with  $n > 1$ ), apart from the use of partial derivatives, it is necessary the use of Jacobian matrices  $\mathbf{J}_{i,j} = \frac{\partial F_i}{\partial x_j}$ , where  $i \in \{1, \dots, n\}$  and  $F_i$  is a function  $\mathbb{R}^m \rightarrow \mathbb{R}$ .

# Generalization and Chain Rule

Let **A** and **B** be two Jacobian matrices.

The chain rule in  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  is:

$$(\mathbf{A} \cdot \mathbf{B})_{i,j} = \sum_{k=1}^m \mathbf{A}_{i,k} \cdot \mathbf{B}_{k,j}$$



# Generalization and Chain Rule

Assuming that the notion of derivatives that we need matches with a linear map, where it is accepted the chain rule previously seen, we will define a new generalization:

$$\begin{aligned} \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} - f'(x) = 0 &\Leftrightarrow \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - (f(x)) + \varepsilon \cdot f'(x)}{\varepsilon} = 0 \\ &\Leftrightarrow \lim_{\varepsilon \rightarrow 0} \frac{\|f(x + \varepsilon) - (f(x)) + \varepsilon \cdot f'(x)\|}{\|\varepsilon\|} = 0 \end{aligned}$$

# Generalization and Chain Rule

Assuming that the notion of derivatives that we need matches with a linear map, where it is accepted the chain rule previously seen, we will define a new generalization:

$$\begin{aligned} \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} - f'(x) = 0 &\Leftrightarrow \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - (f(x)) + \varepsilon \cdot f'(x)}{\varepsilon} = 0 \\ &\Leftrightarrow \lim_{\varepsilon \rightarrow 0} \frac{\|f(x + \varepsilon) - (f(x)) + \varepsilon \cdot f'(x)\|}{\|\varepsilon\|} = 0 \end{aligned}$$

# Derivate as a linear map

## Definition

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

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

If we differentiate two times, we have:

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

# Rules for Differentiation - Sequential Composition

## Theorem

*Let  $f :: a \rightarrow b$  and  $g :: b \rightarrow c$  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:

$$\begin{aligned}\mathcal{D}_0^+ &:: (a \rightarrow b) \rightarrow ((a \rightarrow b) \times (a \rightarrow (a \multimap b))) \\ \mathcal{D}_0^+ f &= (f, \mathcal{D} f)\end{aligned}$$

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

$$\begin{aligned}&\mathcal{D}_0^+ (g \circ f) \\ &\{\text{definition of } \mathcal{D}_0^+\} \\ &= (g \circ f, \mathcal{D} (g \circ f)) \\ &\{\text{theorem and definition of } g \circ f\} \\ &= (\lambda a \rightarrow g(f a), \lambda a \rightarrow \mathcal{D} g (f a) \circ \mathcal{D} f a)\end{aligned}$$

# 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:

$$\begin{aligned}\mathcal{D}_0^+ &:: (a \rightarrow b) \rightarrow ((a \rightarrow b) \times (a \rightarrow (a \multimap b))) \\ \mathcal{D}_0^+ f &= (f, \mathcal{D} f)\end{aligned}$$

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

$$\begin{aligned}\mathcal{D}_0^+ (g \circ f) & \\ \{\text{definition of } \mathcal{D}_0^+\} & \\ = (g \circ f, \mathcal{D}(g \circ f)) & \\ \{\text{theorem and definition of } g \circ f\} & \\ = (\lambda a \rightarrow g(f a), \lambda a \rightarrow \mathcal{D} g (f a) \circ \mathcal{D} f a) &\end{aligned}$$

# Rules for Differentiation - Sequential Composition

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

$$\begin{aligned}\mathcal{D}^+ &:: (a \rightarrow b) \rightarrow (a \rightarrow (b \times (a \multimap b))) \\ \mathcal{D}^+ f a &= (f a, \mathcal{D} f a)\end{aligned}$$

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

$$\begin{aligned}\mathcal{D}^+ &:: (a \rightarrow b) \rightarrow (a \rightarrow b \times (a \multimap b)) \\ \mathcal{D}^+ f a &= (f a, \mathcal{D} f a)\end{aligned}$$

# Rules for Differentiation - Sequential Composition

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

$$\begin{aligned}\mathcal{D}^+ &:: (a \rightarrow b) \rightarrow (a \rightarrow (b \times (a \multimap b))) \\ \mathcal{D}^+ f a &= (f a, \mathcal{D} f a)\end{aligned}$$

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

$$\begin{aligned}\mathcal{D}^+ &:: (a \rightarrow b) \rightarrow (a \rightarrow b \times (a \multimap b)) \\ \mathcal{D}^+ f a &= (f a, \mathcal{D} f a)\end{aligned}$$



# Rules for Differentiation - Sequential Composition

## Corollary

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

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

$$\begin{array}{ccccc} (C \times C^B) \times B^A & \xleftarrow{\mathcal{D}^+ g \times id} & B \times B^A & \xleftarrow{\mathcal{D}^+ f} & A \\ \downarrow (id \times (uncurry (\circ))) \circ assocr & & & \swarrow \mathcal{D}^+ (g \circ f) & \\ C \times C^A & & & & \end{array}$$

# Rules for Differentiation - Parallel Composition

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

$$(\times) :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (a \times b \rightarrow c \times d)$$

$$f \times g = \lambda(a, b) \rightarrow (f\ a, g\ b)$$

## Theorem

*Let  $f :: a \rightarrow c$  and  $g :: b \rightarrow d$  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:

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

### Theorem

*Let  $f :: a \rightarrow c$  and  $g :: b \rightarrow d$  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  $(\times)$*

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

# 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  $f$ ,  $\mathcal{D}^+ f = \lambda a \rightarrow (f a, f)$ .*

# 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  $f$ ,  $\mathcal{D}^+ f = \lambda a \rightarrow (fa, f)$ .*

# 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  $f$ ,  $\mathcal{D}^+ f = \lambda a \rightarrow (fa, f)$ .*

# A short introduction

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



# A short introduction

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

# A short introduction

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

# A short introduction

## Corollary 1.1

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

## Corollary 2.1

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

## Corollary 3.1

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

# Categories

A category is a collection of objects (sets and types) and morphisms (operation between objects), with 2 basic operations (identity and composition) of morphisms, 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 paper, objects are data types and morphisms are functions

**class** *Category* *k* **where**

$id :: (a' k' a)$

$(\circ) :: (b' k' c) \rightarrow (a' k' b) \rightarrow (a' k' c)$

**instance** *Category*  $(\rightarrow)$  **where**

$id = \lambda a \rightarrow a$

$g \circ f = \lambda a \rightarrow g (f a)$

# 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  $F m :: F a \rightarrow F b \in \mathcal{V}$
- $F \text{ id } (\in \mathcal{U}) = \text{id } (\in \mathcal{V})$
- $F (f \circ g) = F f \circ F g$

## Note

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

# Objective

$\mathcal{D}$  definition

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

Adapted definition for  $\mathcal{D}$  type

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

$$\hat{\mathcal{D}} \ f = \mathcal{D} \ (\mathcal{D}^+ \ f)$$

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

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

Saying that  $\hat{\mathcal{D}}$  is a functor is equivalent to, for all  $f$  and  $g$  functions of appropriate types:

$$id = \hat{\mathcal{D}}\ id = \mathcal{D}\ (\mathcal{D}^+ id)$$

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

# Instance deduction

Based on (DP.1) and (DP.2) we'll rewrite the above into the following definition:

$$\begin{aligned} id &= \mathcal{D} (\lambda a \rightarrow (id \ a, id)) \\ \hat{\mathcal{D}} \ g \circ \hat{\mathcal{D}} \ f &= \mathcal{D} (\lambda a \rightarrow \mathbf{let} \ \{ (b, f') = \mathcal{D}^+ \ f \ a; (c, g') = \\ &\mathcal{D}^+ \ g \ b \} \ \mathbf{in} \ (c, g' \circ f')) \end{aligned}$$

The first equation shown above has a trivial solution.

To solve the second we'll first solve a more general one:

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

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:

$$\begin{aligned}
 id &= \mathcal{D} (\lambda a \rightarrow (id \ a, id)) \\
 \hat{\mathcal{D}} \ g \circ \hat{\mathcal{D}} \ f &= \mathcal{D} (\lambda a \rightarrow \mathbf{let} \ \{(b, f') = \mathcal{D}^+ \ f \ a; (c, g') = \\
 &\mathcal{D}^+ \ g \ b\} \ \mathbf{in} \ (c, g' \circ f'))
 \end{aligned}$$

The first equation shown above has a trivial solution.

To solve the second we'll first solve a more general one:

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

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:

$$\begin{aligned} id &= \mathcal{D} (\lambda a \rightarrow (id \ a, id)) \\ \hat{\mathcal{D}} \ g \circ \hat{\mathcal{D}} \ f &= \mathcal{D} (\lambda a \rightarrow \mathbf{let} \ \{ (b, f') = \mathcal{D}^+ \ f \ a; (c, g') = \\ &\mathcal{D}^+ \ g \ b \} \ \mathbf{in} \ (c, g' \circ f')) \end{aligned}$$

The first equation shown above has a trivial solution.

To solve the second we'll first solve a more general one:

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

This condition also leads us to a trivial solution inside our instance.

# Instance deduction

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

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

Categorical instance we've deduced

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

$$\text{id} = \text{linearD id}$$

$$\mathcal{D} \ g \circ \mathcal{D} \ f =$$

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

# 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

$$\begin{aligned} & \text{id} \circ \hat{\mathcal{D}} f \\ & \{ \text{functor law for id (specification of } \hat{\mathcal{D}}) \} \\ & = \hat{\mathcal{D}} \text{id} \circ \hat{\mathcal{D}} f \\ & \{ \text{functor law for } (\circ) \} \\ & = \hat{\mathcal{D}} (\text{id} \circ f) \\ & \{ \text{categorical law} \} \\ & = \hat{\mathcal{D}} f \end{aligned}$$

# Instance proof

## (C.2) proof

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

## Note

This 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:

**class** *Category*  $k \Rightarrow \text{Monoidal } k$  **where**

$(\times) :: (a \text{ ' } k \text{ ' } c) \rightarrow (b \text{ ' } k \text{ ' } d) \rightarrow ((a \times b) \text{ ' } k \text{ ' } (c \times d))$

**instance** *Monoidal*  $(\rightarrow)$  **where**

$f \times g = \lambda(a, b) \rightarrow (f \ a, g \ b)$

## 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:

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

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 it's definition and generalize the condition:

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

and this is enough for our new instance.

# Instance deduction

From corollary 2.1 we can deduce that:

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

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 it's definition and generalize the condition:

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

and this is enough for our new instance.



# Instance deduction

From corollary 2.1 we can deduce that:

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

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 it's definition and generalize the condition:

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

and this is enough for our new instance.

# Instance deduction

Categorical instance we've deduced

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

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

# Cartesian categories and functors

**class** *Monoidal*  $k \Rightarrow \text{Cartesian } k$  **where**

$\text{exl} :: (a, b) \rightarrow k \rightarrow a$

$\text{exr} :: (a, b) \rightarrow k \rightarrow b$

$\text{dup} :: a \rightarrow k \rightarrow (a, a)$

**instance** *Cartesian*  $(\rightarrow)$  **where**

$\text{exl} = \lambda(a, b) \rightarrow a$

$\text{exr} = \lambda(a, b) \rightarrow b$

$\text{dup} = \lambda a \rightarrow (a, a)$

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

- $F$  is a monoidal functor
- $F \text{ exl} = \text{exl}$
- $F \text{ exp} = \text{exp}$
- $F \text{ dup} = \text{dup}$

# Instance deduction

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

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

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

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

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 it's definition and remembering linearD's definition we can obtain:

$exl = \text{linearD } exl$

$exr = \text{linearD } exr$

$dup = \text{linearD } dup$

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 paper coproducts are categorical products, i.e., biproducts

## Definition

```
class Category  $k \Rightarrow \text{Cocartesian } k$  where  
  inl ::  $a \text{ ' } k \text{ ' } (a, b)$   
  inr ::  $b \text{ ' } k \text{ ' } (a, b)$   
  jam ::  $(a, a) \text{ ' } k \text{ ' } a$ 
```

# Cocartesian functors

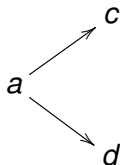
## Cocartesian functor definition

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

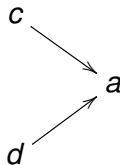
- $F$  is a functor
- $F \text{ inl} = \text{inl}$
- $F \text{ inr} = \text{inr}$
- $F \text{ jam} = \text{jam}$

# Fork and Join

- $\Delta :: \text{Cartesian } k \Rightarrow (a' k' c) \rightarrow (a' k' d) \rightarrow (a' k' (c \times d))$



- $\nabla :: \text{Cartesian } k \Rightarrow (c' k' a) \rightarrow (d' k' a) \rightarrow ((c \times d)' k' a)$





# Instance of $\rightarrow^+$

**newtype**  $a \rightarrow^+ b = \text{AddFun } (a \rightarrow b)$

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

**type** *Obj*  $(\rightarrow^+) = \text{Additive}$

$\text{id} = \text{AddFun id}$

$\text{AddFun } g \circ \text{AddFun } f = \text{AddFun } (g \circ f)$

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

$\text{AddFun } f \times \text{AddFun } g = \text{AddFun } (f \times g)$

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

$\text{exl} = \text{AddFun exl}$

$\text{exr} = \text{AddFun exr}$

$\text{dup} = \text{AddFun dup}$

Instance of  $\rightarrow^+$ 

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

*inl* = *AddFun inlF*

*inr* = *AddFun inrF*

*jam* = *AddFun jamF*

*inlF* :: *Additive*  $b \Rightarrow a \rightarrow a \times b$

*inrF* :: *Additive*  $a \Rightarrow b \rightarrow a \times b$

*jamF* :: *Additive*  $a \Rightarrow a \times a \rightarrow a$

*inlF* =  $\lambda a \rightarrow (a, 0)$

*inrF* =  $\lambda b \rightarrow (0, b)$

*jamF* =  $\lambda(a, b) \rightarrow a + b$

# NumCat definition

```
class NumCat k a where
```

```
  negateC :: a ' k ' a
```

```
  addC :: (a × a) ' k ' a
```

```
  mulC :: (a × a) ' k ' a
```

```
  ...
```

```
instance Num a ⇒ NumCat (→) a where
```

```
  negateC = negate
```

```
  addC = uncurry (+)
```

```
  mulC = uncurry (*)
```

```
  ...
```

$$\mathcal{D}(\text{negate } u) = \text{negate } (\mathcal{D} u)$$

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

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

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

**class** *Scalable* *k a* **where**

*scale* :: *a*  $\rightarrow$  (*a* ' *k* ' *a*)

**instance** *Num* *a*  $\Rightarrow$  *Scalable* ( $\rightarrow^+$ ) *a* **where**

*scale* *a* = *AddFun* ( $\lambda da \rightarrow a * da$ )

**instance** *NumCat* *D* **where**

*negateC* = *linearD* *negateC*

*addC* = *linearD* *addC*

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

**instance** *FloatingCat* *D* **where**

*sinC* = *D* ( $\lambda a \rightarrow (\sin a, \text{scale } (\cos a))$ )

*cosC* = *D* ( $\lambda a \rightarrow (\cos a, \text{scale } (-\sin a))$ )

*expC* = *D* ( $\lambda a \rightarrow \text{let } e = \exp a \text{ in } (e, \text{scale } e)$ )

...

# Examples

$$\text{sqr} :: \text{Num } a \Rightarrow a \rightarrow a$$

$$\text{sqr } a = a * a$$

$$\text{magSqr} :: \text{Num } a \Rightarrow a \times a \rightarrow a$$

$$\text{magSqr } (a, b) = \text{sqr } a + \text{sqr } b$$

$$\text{cosSinProd} :: \text{Floating } a \Rightarrow a \times a \rightarrow a \times a$$

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

With a compiler plugin we can obtain

$$\text{sqr} = \text{mulC} \circ (\text{id} \Delta \text{id})$$

$$\text{magSqr} = \text{addC} \circ (\text{mulC} \circ (\text{exl} \Delta \text{exl}) \Delta \text{mulC} \circ (\text{exr} \Delta \text{exr}))$$

$$\text{cosSinProd} = (\text{cosC} \Delta \text{sinC}) \circ \text{mulC}$$

# Generalizing Automatic Differentiation

**newtype**  $D_k a b = D (a \rightarrow b \times (a' k' b))$

*linearD* ::  $(a \rightarrow b) \rightarrow (a' k' b) \rightarrow D_k a b$

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

**instance** *Category*  $k \Rightarrow \text{Category } D_k$  **where**

**type** *Obj*  $D_k = \text{Additive} \wedge \text{Obj } k \dots$

**instance** *Monoidal*  $k \Rightarrow \text{Monoidal } D_k$  **where** ...

**instance** *Cartesian*  $k \Rightarrow \text{Cartesian } D_k$  **where** ...

**instance** *Cocartesian*  $k \Rightarrow \text{Cocartesian } D_k$  **where**

*inl* = *linearD inlF inl*

*inr* = *linearD inrF inr*

*jam* = *linearD jamF jam*

**instance** *Scalable*  $k\ s \Rightarrow \text{NumCat } D_k\ s$  **where**  
  *negateC* = *linearD negateC negateC*  
  *addC* = *linearD addC addC*  
  *mulC* = *D* ( $\lambda(a, b) \rightarrow (a * b, \text{scale } b \nabla \text{scale } a)$ )



# Matrices

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

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

# Extracting a Data Representation

In machine learning, a Gradient-based optimization works by searching for local minima in the domain of a differentiable function  $f :: a \rightarrow s$ . 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  $f :: a \rightarrow s$ . 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  $p \rightarrow s$  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:

$$\exists p \forall s \ F \ s \cong \ p \rightarrow s$$

# Generalized Matrices

Given a scalar field  $s$ , a free vector space has the form  $p \rightarrow s$  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:

$$\exists p \forall s \ F \ s \cong \ p \rightarrow s$$

# A short introduction

- We've derived and generalized an AD algorithm using categories
- With fully right-associated compositions this algorithm becomes a forward-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

$f :: a' \rightarrow k' \rightarrow b \Rightarrow (\circ f) :: (b' \rightarrow k' \rightarrow r) \rightarrow (a' \rightarrow k' \rightarrow r)$  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

**newtype**  $\text{Cont}_k^r a b = \text{Cont} ((b \rightarrow k' r) \rightarrow (a \rightarrow k' r))$

## Functor derived from type

$\text{cont} :: \text{Category } k \Rightarrow (a \rightarrow k' b) \rightarrow \text{Cont}_k^r a b$   
 $\text{cont } f = \text{Cont } (\circ f)$



# Instance deduction

**instance** *Category*  $k \Rightarrow \text{Category } \text{Cont}_k^r$  **where**  
*id* = *Cont id*

*Cont g*  $\circ$  *Cont f* = *Cont (f*  $\circ$  *g)*

**instance** *Monoidal*  $k \Rightarrow \text{Monoidal } \text{Cont}_k^r$  **where**  
*Conf f*  $\times$  *Cont g* = *Cont (join*  $\circ$  (*f*  $\times$  *g)*  $\circ$  *unjoin)*

**instance** *Cartesian*  $k \Rightarrow \text{Cartesian } \text{Cont}_k^r$  **where**  
*exl* = *Cont (join*  $\circ$  *inl)*; *exr* = *Cont (join*  $\circ$  *inr)*  
*dup* = *Cont (jam*  $\circ$  *unjoin)*

**instance** *Cocartesian*  $k \Rightarrow \text{Cocartesian } \text{Cont}_k^r$  **where**  
*inl* = *Cont (exl*  $\circ$  *unjoin)*; *inr* = *Cont (exr*  $\circ$  *unjoin)*  
*jam* = *Cont (join*  $\circ$  *dup)*

**instance** *Scalable*  $k$  *a*  $\Rightarrow \text{Scalable } \text{Cont}_k^r$  *a* **where**  
*scale s* = *Cont (scale s)*

## A short introduction

Due to it's 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 instantiation

```
class HasDot (S) u where dot :: u → (u → s)
instance HasDot (IR) IR where dot = scale
instance (HasDot (S) a, HasDot (S) b)
  ⇒ HasDot (S) (a × b)
where dot (u, v) = dot u Δ dot v
```

# Instance deduction

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

## Type definition for duality

```
newtype  $\text{Dual}_k a b = \text{Dual } (b \text{ ' k' } a)$ 
```

# 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

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

where

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

# 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

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

## Forward-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 generalized FAD algorithm.

This algorithm is far more appropriate for low dimension domains.

### Type definition and functor from type

```
newtype  $Begin_k^r$   $a\ b = Begin\ ((r\ 'k'\ a) \rightarrow (r\ 'k'\ b))$   

 $begin :: Category\ k \Rightarrow (a\ 'k'\ b) \rightarrow Begin_k^r\ a\ b$   

 $begin\ f = Begin\ (f \circ)$ 
```

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 isomorphic 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*  $k \Rightarrow \text{Monoidall } k \ h$  **where**  
    *crossl* ::  $h \ (a' \ k' \ b) \rightarrow (h \ a' \ k' \ h \ b)$

**instance** *Zip*  $h \Rightarrow \text{Monoidall } (\rightarrow) \ h$  **where**  
    *crossl* = *zipWith id*



```
class Monoidall k h  $\Rightarrow$  Cartesian1 k h where  
  exl  :: h (h a ' k ' a)  
  repll :: a ' k ' h a  
  
instance (Representable h, Zip h, Pointed h)  $\Rightarrow$   
  Cartesian1 ( $\rightarrow$ ) h where  
  exl = tabulate (flip index)  
  repll = point
```

- The following is not the class the author was thinking

```
class Representable h where  
  type Rep h :: *  
  tabulate :: (Rep h  $\rightarrow$  a)  $\rightarrow$  h a  
  index :: h a  $\rightarrow$  Rep h  $\rightarrow$  a
```

**class** *Monoidall* *k h*  $\Rightarrow$  *Cocartesian1* *k h* **where**

*inl* :: *h* (*a* ' *k* ' *h a*)

*jaml* :: *h a* ' *k* ' *a*

**instance** (*Monoidall* *k h*, *Zip h*)  $\Rightarrow$  *Monoidall* *D<sub>k</sub> h* **where**

*crossl* *fs* = *D* ((*id*  $\times$  *crossl*)  $\circ$  *unzip*  $\circ$  *crossl* (*fmap unD fs*))

**instance** (*Cocartesian1* ( $\rightarrow$ ) *h*, *Cartesian1* *k h*, *Zip h*)  $\Rightarrow$

*Cartesian1* *D<sub>k</sub> h* **where**

*exl* = *linearD* *exl* *exl*

*repll* = *zipWith linearD repll repll*

**instance** (*Cocartesian1* *k h*, *Zip h*)  $\Rightarrow$  *Cocartesian1* *D<sub>k</sub> h* **where**

*inl* = *zipWith linearD inlF inl*

*jaml* = *linearD* *sum jaml*

**class** *Monoidall* *k h*  $\Rightarrow$  *Cocartesian1* *k h* **where**

*inl* :: *h* (*a* ' *k* ' *h a*)

*jaml* :: *h a* ' *k* ' *a*

**instance** (*Monoidall* *k h*, *Zip h*)  $\Rightarrow$  *Monoidall* *D<sub>k</sub> h* **where**

*crossl* *fs* = *D* ((*id*  $\times$  *crossl*)  $\circ$  *unzip*  $\circ$  *crossl* (*fmap unD fs*))

**instance** (*Cocartesian1* ( $\rightarrow$ ) *h*, *Cartesian1* *k h*, *Zip h*)  $\Rightarrow$

*Cartesian1* *D<sub>k</sub> h* **where**

*exl* = *zipWith linearD exl exl*

*repll* = *linearD repll repll*

**instance** (*Cocartesian1* *k h*, *Zip h*)  $\Rightarrow$  *Cocartesian1* *D<sub>k</sub> h* **where**

*inl* = *zipWith linearD inlF inl*

*jaml* = *linearD sum jaml*

# Conclusion

- Suggests that some of the work referred to does just a part of this paper.
- This paper ([Elliott 2018][2]) is a follow up of [Elliott 2017][1]
- Suggests that this implementation is simple, efficient, it can free memory dynamically (RAD) and is naturally parallel.
- Future work are detailed performance 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.