Studying Automatic Differentiation

Group 114:
Artur Queiroz PG38014, Ezequiel Moreira PG38413, Nelson Loureiro PG37020

June 2019

1 Introduction

The objective of this document is to expose the work done for the project of the course *Laboratórios de Engenharia Informática* for the period of 2018/2019 whose major work was the study of an article about Automatic Differentiation ([1]), under the guidance of supervisors José Nuno Oliveira and Pedro Patrício. This project was proposed to our group and was chosen thanks to the fact that the article has a semi-implemented Automatic Differentiation algorithm with Correct By Construction techniques, a subject that has been growing in influence the last few years.

We have also implemented the ideas expressed in the article with moderate success.

2 Differentiable function

Since automatic differentiation (AD) has to do with computing derivatives, let's begin by considering what derivatives are. Using the definitions from chapter 2 of Michael Spivak's book ([3]):

Definition 2.1. A function $f : \mathbb{R} \to \mathbb{R}$ is differentiable at $x \in \mathbb{R}$, if there exists a real $f'(x)$ such that

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

This is, we can say that the derivate of f at x, represented as $f'(x)$, is a local linear approximation of f at x. But this definition is one case of a more general definition.

Using simple transformations in the above equation, we have the following equivalences:

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

And with this, we can show the referred more general definition.

Definition 2.2. 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

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

Since $f'(x)$ is a linear transformation, we can represent the derivative of a function as a Jacobian Matrix, where in the Definition 2.1, we can represent the derivative as a matrix with a single element. We reached a good generalization and let's now move on to an implementation in Haskell.

2.1 First Definition of Derivative

Let $f :: a \to b$ be a function, where a and b are vectorial spaces that share a common underlying field. The derivative of f at some value in a is a linear transformation from a to b, which we will write as '$a \multimap b$'. The numbers, vectors and matrices mentioned above are all different representations of linear maps. Written in Haskell style:

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

where the function $\mathcal{D}$ receives a function of type $(a \to b)$ and a value of its domain has its input and yields a linear transformation of type $(a \multimap b)$.

If we differentiate two times, we have:

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

The type $(a \multimap a \multimap b)$ is a linear map that yields a linear map, which is the curried form of a bilinear map.

2.2 Sequential Composition

Turning now to the composition of functions, we have the following Theorem, known as Chain Rule.

Theorem 2.1. *Let $f :: a \to b$ and $g :: b \to c$ be two functions. Then the derivative of the composition between f and g is*

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

Note 2.1. *From the definition of composition, we know that $g \circ f$ has type $(a \to c)$, then $\mathcal{D}\ (g \circ f)\ a$ has type $(a \multimap c)$. We also know that $\mathcal{D}\ f\ a$ has type $(a \multimap b)$ and $\mathcal{D}\ g\ (f\ a)$ has type $(b \multimap c)$, so the composition of these two functions has type $(a \multimap c)$. Thus, both sides of the equation have the same type.*

Unfortunately, Theorem 2.1 does not give us a good compositional recipe for differentiating sequential compositions because $\mathcal{D}\ (g \circ f)$ is not constructed solely from $\mathcal{D}\ f$ and $\mathcal{D}\ g$, it also needs f itself.

For that reason it would be better to propose a second derivative definition.

2.3 Second Derivative Definition

The second derivative definition is the following:

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

$\mathcal{D}_0^+ f = (f, \mathcal{D} f)$

As desired, this altered specification is compositional:

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

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

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

$\{$theorem 2.1 and definition of $g \circ f\}$

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

The derivative of the composition $\mathcal{D}_0^+ (g \circ f)$ is entirely obtained from the components of $\mathcal{D}_0^+ g$ and $\mathcal{D}_0^+ f$. Note that the two components of the definition use $(f a)$, requiring redundant work, resulting in an impractically expensive algorithm. Because of this we need a new derivative specification.

2.4 Third Derivative Definition

The third derivative definition is the following:

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

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

With this last optimization we have the following Corollary:

Corollary 2.1. *$\mathcal{D}^+$ is (efficiently) compositional with respect to $(\circ)$. Specifically, in Haskell,*

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

2.5 Parallel Composition

Now let's introduce another important way of combining functions, the 'cross' operation, that combines two functions in parallel. The cross operation is as follows:

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

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

The following Theorem, designated Cross Rule, combines the above derivative definition and the cross operation.

Theorem 2.2. *Let $f :: a \to c$ and $g :: b \to d$ be two functions. Then the cross rule is the following:*

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

Note 2.2. *The function $\mathcal{D} f\ a$ has the type $(a \multimap c)$ and $\mathcal{D} g\ b$ has the type $(b \multimap d)$, so both sides of the equation are of the type $(a \times b \multimap c \times d)$.*

The Theorem 2.2 gives us what is necessary to write the next Corollary.

Corollary 2.2. *The function $\mathcal{D}^+$ is compositional with respect to $(\times)$. Specifically,*

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

The compositions we've talked about thus far preserve linearity, which we'll explain in greater detail next section.

2.6 Linear Functions

A function f is said to be linear when it distributes over (preserves the structure of) vector addition and scalar multiplication, i.e.,

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

Theorem 2.3. *For all linear functions f, $\mathcal{D} f\ a = f$.*

Since the derivative of a function is a local linear approximation of f at a, this Theorem tells us that linear functions are their own perfect linear approximations.

For example, considering the function $id = \lambda a \to a$, the previous Theorem tells us that $\mathcal{D}\, id\ a = id$. Expressing in matrix form, the derivative of id can be represented by the matrix [1] (matrix with the single element 1) or by an identity matrix.

The following Corollary of Theorem 2.3 tells us how to construct $\mathcal{D}^+ f$ for all linear functions.

Corollary 2.3. *For all linear functions f, $\mathcal{D}^+ f = \lambda a \to (fa, f)$.*

3 Automatic Differentiation(AD) algorithm

In order to achieve the major goal of the article we must implement $\mathcal{D}^+$. But there's a problem: $\mathcal{D}$ is not computable.

However through the use of the corollaries we've defined so far we can deduce an alternative implementation using category theory as its basis.

3.1 Category and respective functor

A category consists in a collection of objects (generalisation of sets and types) and morphisms (operations between objects).

Every category has defined in it $id :: a \to a$ (identity for objects of a category) and morphism composition ($\circ$), such that for 2 morphisms $f :: a \to b$ and $g :: b \to c$ of a given category we can define $g \circ f :: a \to c$. Furthermore, there are a couple of categorical laws that must be followed:

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

With this in mind we can express categories as an Haskell class and present an instance for ($\to$):

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

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

An F functor between $\mathcal{U}$ and $\mathcal{V}$ categories is such that a) for each u object of $\mathcal{U}$ we have that F u is an object in $\mathcal{V}$ and b) for a morphism $f :: a \to b$ of $\mathcal{U}$ there exists $F\ f :: F\ a \to F\ b$ morphism in $\mathcal{V}$.

In addition to this a functor must preserve categorical structure. For this particular case, F id $\in \mathcal{U}$ = id ($\in \mathcal{V}$) and F (f $\circ$ g) = F f $\circ$ F g must be true for F functor.

Given the prior definitions and taking into account the various corollaries we'll deduce how to implement $\mathcal{D}^+$.

3.1.1 Instance deduction

To do this we first define a new type of data:

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

With this new type we adapt $\mathcal{D}^+$ definition to use it, creating $\hat{\mathcal{D}}$:

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

With this in mind we must now deduce a categorical instance for $\mathcal{D}$ such that $\hat{\mathcal{D}}$ is a valid functor.

(1) $\hat{\mathcal{D}}$ functor is equivalent to, for all f and g morphisms of appropriate type,

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

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

(2) From corollaries 2.3 and 2.1 we know that

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

Replacing what we have in (1) using what we've determined in (2) we can conclude that

$id = \mathcal{D}\ (\lambda a \to (id\ a, id))$

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

From this we can trivially obtain the identity definition for our instance, but morphism composition needs further work.

In order to use the morphism composition we'll first generalise the condition derived above:

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

As we can see, this condition can be used directly in our new instance, and as such we've arrived at our new instance.

3.1.2 Deduced Instance

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

where *linearD* is equivalent to the definition of $\hat{\mathcal{D}}$ for linear functions:

$linearD :: (a \to b) \to \mathcal{D}\ a\ b$
$linearD\ f = \mathcal{D}\ (\lambda a \to (f\ a, f))$

3.2 Instance proof

In order to prove that the instance we've arrived at is well constructed we must prove that it follows laws (C.1) and (C.2).

In order to simplify these proofs we assume that $\mathcal{D}^+$ is a functor. In order to do this we must make a concession: all morphisms we're considering arise from $\mathcal{D}^+$, i.e., we only use $f' :: \mathcal{D}\ a\ b$ where $f' = \hat{\mathcal{D}}\ f$ for some $f :: a \to b$ morphism of $\mathcal{D}$.

This becomes possible by making $\mathcal{D}$ a b an abstract type.

With this in mind we can now prove that our instance obeys the rules:

3.2.1 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 }
$= \hat{\mathcal{D}}\ f$

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

As a final note these proofs required nothing from $\mathcal{D}$ and $\hat{\mathcal{D}}$ besides the functor laws, meaning we can avoid presenting this proof for all other categorical instances arising from a functor.

3.3 Monoidal category and respective functor

As noted before we want our algorithm to have parallel composition in it. To do this we must use a new type of category: the monoidal category.

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

We also define the monoidal functor in the same manner as the previous category type, noting that it must obey the following rules:

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

3.3.1 Instance deduction

Deducing a instance of our new category type is done using the same method as before:

(1) $\hat{\mathcal{D}}$ monoidal functor is equivalent to, for all f and g morphisms of appropriate type,

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

(2) From corollary 2.1 we know that

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

Replacing what we have in (1) using what we've determined in (2) we can conclude that

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

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

and we can use this directly in our new instance.

3.3.2 Deduced Instance

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

3.4 Cartesian and Cocartesian category and respective functors

Monoidal category gives us the ability to combine functions, but not to split data, nor does it give us a way to duplicate or remove it.

In order to add these functionalities we must use 2 new types of categories: cartesian and cocartesian categories, noting that these 2 types of categories are dual to one another.

We also take note that for this paper co-products coincide with categorical products. As such, we cannot instantiate for $(\rightarrow)$ in the cocartesian category type as we've done before.

With that noted we define our categories

class *Monoidal* $k \Rightarrow$ *Cartesian* k **where**
  $exl :: (a, b)\ `k`\ a$
  $exr :: (a, b)\ `k`\ b$
  $dup :: a\ `k`\ (a, a)$

instance *Cartesian* $(\rightarrow)$ **where**
  $exl = \lambda(a, b) \rightarrow a$
  $exr = \lambda(a, b) \rightarrow b$
  $dup = \lambda a \rightarrow (a, a)$

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

and note the following properties must be followed by the respective functors:

3.4.1 Cartesian functor properties

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

3.4.2 Cocartesian functor properties

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

We note that due to the duality of the 2 categories we'll only show the deduction for the cartesian category.

3.4.3 Instance deduction

We follow the same process as before:

(1) $\hat{\mathcal{D}}$ cartesian functor is equivalent to

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

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

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

(2) From corollary 3.1 and exl, exr and dup's linearity we deduce that

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

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

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

After replacing what we have in (1) using what we've determined in (2), we can take the result and directly use it in our new instance.

3.4.4 Deduced Instance

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

4 Fork and Join

We can derive this two useful operations with just the operators defined earlier.

$\Delta :: Cartesian\ k \Rightarrow (a\ `k`\ c) \to (a\ `k`\ d) \to (a\ `k`\ (c \times d))$

$f\ \Delta\ g = (f \times g) \circ dup$

$\nabla :: Cartesian\ k \Rightarrow (c\ `k`\ a) \to (d\ `k`\ a) \to ((c \times d)\ `k`\ a)$

$f\ \nabla\ g = jam \circ (f \times g)$

5 Instance of $\rightarrow^+$

We can now describe Additive Functions: functions that transform something that is Additive to something that is also Additive, and using these we can derive instances for every categorical type discussed before for $(\rightarrow^+)$

```
newtype a →+ b = AddFun (a → b)

instance Category (→+) where
  type Obj (→+) = Additive
  id = AddFun id
  AddFun g ∘ AddFun f = AddFun (g ∘ f)

instance Monoidal (→+) where
  AddFun f × AddFun g = AddFun (f × g)

instance Cartesian (→+) where
  exl = AddFun exl
  exr = AddFun exr
  dup = AddFun dup

instance Cocartesian (→+) where
  inl = AddFun inlF
  inr = AddFun inrF
  jam = AddFun jamF

inlF :: Additive b ⇒ a → a × b
inrF :: Additive a ⇒ b → a × b
jamF :: Additive a ⇒ a × a → a

inlF = λa → (a, 0)
inrF = λb → (0, b)
jamF = λ(a, b) → a + b
```

6 Numeric operations

To describe numerical operations in a k category we can use the definitions above, noting that the same process can be done for Floating types.

```
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 (∗)
  ...
```

When we observe the mathematical definition for differentiation it can be seen that neither u or v is defined in a concrete way, and they can mean different things in different contexts, like functions, numbers, tuples, etc. This requires us to define each one of them separately.

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

To make this definition more uniform and simple we can differentiate not the expression but the operation itself. For example the *negate* and the $+$ can be seen as linear because they don't require the inputs from the expression, but the $*$ operation cannot, and has such requires an alternative definition such as:

$$\mathcal{D}\ mulC\ (a, b) = scale\ b\ \nabla\ scale\ a$$

```
class Scalable k a where
  scale :: a → (a `k` a)
instance Num a ⇒ Scalable (→+) a where
  scale a = AddFun (λda → a ∗ da)
instance NumCat D where
  negateC = linearD negateC
  addC = linearD addC
  mulC = D (λ(a, b) → (a ∗ b, scale b ∇ scale a))
instance FloatingCat D where
  sinC = D (λa → (sin a, scale (cos a)))
  cosC = D (λa → (cos a, scale (−sin a)))
  expC = D (λa → let e = exp a in (e, scale e))
  ...
```

Some examples of functions:

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

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

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

With a compiler plugin referenced in the article we can convert the above expressions to our expressions instanciated earlier:

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

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

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

7 Generalising Automatic Differentiation

In the other sections we've used the definition of $\mathcal{D}$ with the $\multimap$, but how do we define a linear map? To be more expressive we can define $\mathcal{D}$ for any given "linear" category k.

$$\textbf{newtype}\ D_k\ a\ b = D\ (a \to b \times (a\ `k`\ b))$$
$$linearD :: (a \to b) \to (a\ `k`\ b) \to D_k\ a\ b$$
$$linearD\ f\ f' = D\ (\lambda a \to (f\ a, f'))$$

$$\textbf{instance}\ Category\ k \Rightarrow Category\ D_k\ \textbf{where}$$
$$\quad \textbf{type}\ Obj\ D_k = Additive \wedge Obj\ k$$
$$\quad id = linearD\ id\ id$$
$$\quad \mathcal{D}\ g \circ \mathcal{D}\ f = \mathcal{D}\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))$$

$$\textbf{instance}\ Monoidal\ Category\ k \Rightarrow Category\ D_k\ \textbf{where}$$
$$\quad \mathcal{D}\ f \times \mathcal{D}\ g = \mathcal{D}\ (\lambda(a, b) \to \textbf{let}\ \{(c, f') = f\ a; (d, g') = g\ b\}\ \textbf{in}\ ((c, d), f' \times g'))$$

$$\textbf{instance}\ Cartesian\ Category\ k \Rightarrow Category\ D_k\ \textbf{where}$$
$$\quad exl = linearD\ exl\ exl$$
$$\quad exr = linearD\ exr\ exr$$
$$\quad dup = linearD\ dup\ dup$$

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

$$\textbf{instance}\ NumCat\ D\ \textbf{where}$$
$$\quad negateC = linearD\ negateC\ negateC$$
$$\quad addC = linearD\ addC\ addC$$
$$\quad mulC = D\ (\lambda(a, b) \to (a * b, scale\ b \nabla scale\ a))$$

8 Matrices

Let us now consider matrices. There are three non-exclusive possibilities for a non-zero W matrix:

- 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.

These operations that have been mentioned before,

$scale :: a \to (a \text{ `}k\text{` } a)$

$(\nabla) :: (c \text{ `}k\text{` } a) \to (d \text{ `}k\text{` } a) \to ((c \times d) \text{ `}k\text{` } a)$

$(\Delta) :: (a \text{ `}k\text{` } c) \to (a \text{ `}k\text{` } d) \to (a \text{ `}k\text{` } (c \times d))$

correspond exactly to the three possibilities seen above, where in linear maps, the domain and the co-domain are determined, respectively, by the width and height of the matrix together with the type of matrix elements. This happens if we use the convention of matrix on the left multiplied by a column vector on the right.

8.1 Extracting a Data Representation

In addition to what we have used so far, we will also need a data representation for the linear map.

For example, in machine learning gradient-based optimisation works by searching for local minima in the domain of a differentiable function f. 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. If U has dimension m (for example $U = \mathbb{R}^n$), this sampling requires m passes.

In the case where m is a relatively small number, then the method of Jacobian matrix extraction is reasonably efficient. The problem is that it worsens significantly as dimension grows. In particular, many useful problems involve gradient-based optimization over very high-dimensional spaces, which makes this technique inefficient. The next section gives us an alternative.

8.2 Generalised Matrices

Rather than representing derivatives as functions and then extracting a (Jacobian) matrix, a more conventional alternative is to construct and combine matrices in the first place.These matrices are usually rectangular arrays that represent $\mathbb{R}^m \multimap \mathbb{R}^n$. The problem with this alternative is that interferes with the composability we get from organising around binary cartesian products.

There is, however, an especially convenient perspective on linear algebra: free vector spaces. Given a scalar field s, any 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 only finitely many p values can have non-zero images). Scaling a vector $v :: p \to s$ or adding two such vectors is defined in the usual way for functions.

Thus, rather than directly using functions as representations, we can alternatively use any representation isomorphic to such a function. In particular, we can represent vector spaces over a given field as a representable functor, i.e., a functor F such that

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

This method is convenient in a richly typed functional language like Haskell, which comes with libraries of functor-level building blocks. Four such building blocks are functor product, functor composition and their corresponding identities, which are the unit functor (containing no elements) and the identity functor (containing one element).

All of these functors give data representations of functions that save recomputation over a native function representation. They also provide a composable, type-safe alternative to the more commonly used multi-dimensional arrays (often called 'tensors') in machine learning libraries.

9 Generalised RAD algorithm

Earlier in this document we derived and generalised an AD (automatic differentiation) algorithm, and now we desire a generalisation for an RAD (reverse-mode AD) and FAD (forward mode AD) algorithm derived from this more generic one. Let's start with RAD.

In order to achieve an RAD algorithm from our generalised AD we must have it so all compositions of morfisms are left-associated.

To achieve this we start by converting the way we write morfisms in a category k:

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

If we build a category based around this concept we'll arrive at our algorithm.

To do that we start by defining a data type and constructing a functor that uses it:

newtype $Cont^r_k\ a\ b = Cont\ ((b \text{ `}k\text{` } r) \to (a \text{ `}k\text{` } r))$

$cont :: Category\ k \Rightarrow (a \text{ `}k\text{` } b) \to Cont^r_k\ a\ b$

$cont\ f = Cont\ (\circ f)$

From these we do the same we've done before for $\mathcal{D}$ and $\hat{\mathcal{D}}$, and we derive all the various instances for our new data type of the numerous categorical types.

9.1 Deduced Instances

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**
$scale\ s = Cont\ (scale\ s)$

With these instances derived we have created a generalised RAD algorithm.

10 Gradients and duality

Due to its widespread use in machine learning we'll talk about a specific case of an AD algorithm: computing gradients (derivatives of functions with scalar co-domains).

A vector space A over a scalar field s has A $\multimap$ s as its dual, and the dual space of A is not only a vector space but also isomorphic to A if it has finite dimension.

As such we can represent each linear map in A $\multimap$ s using the form dot u for some u :: A where

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

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

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

10.1 Duality

Using the construction $Cont_k^r$ from the previous section and taking r to be the scalar field s, we observe that the internal representation of $Cont_{\multimap}^s$ a b is (b $\multimap$ s) $\to$ (a $\multimap$ s) , which is isomorphic to (a $\to$ b).

To this representation we'll call the dual of k:

$$\textbf{newtype } \mathit{Dual}_k\ a\ b = \mathit{Dual}\ (b\ `k`\ a)$$

With this definition we can achieve the dual representations of generalised linear maps by converting them from $Cont_k^S$ to $Dual_k$ using a functor:

$$\mathit{asDual} :: (\mathit{HasDot}\ (S)\ a, \mathit{HasDot}\ (S)\ b) \Rightarrow ((b\ \multimap\ s) \to (a\ \multimap\ s)) \to (b\ \multimap\ a)$$
$$\mathit{asDual}\ (\mathit{Cont}\ f) = \mathit{Dual}\ (\mathit{onDot}\ f)$$

where

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

From the new data type and the functor derived from it we can now do the same process as before and determine the instances for our new category:

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**
$scale\ s = Dual\ (scale\ s)$

This new instances dualize a computation exactly.

As final notes to this chapter we have that (∇) and (Δ) mutually dualize and that by that using the first definition of matrices that was derived in this paper, dualizing a matrix is equivalent to transposing it, with the advantage of not requiring any matrix computations unless k requires them.

11 Generalised FAD algorithm

To derive a generalised FAD algorithm from the generalised AD algorithm and calculate gradients from it all that needs to be done has already been show in the previous 2 sections, taking into account we can simply consider

newtype $Begin^r_k\ a\ b = Begin\ ((r\ `k`\ a) \to (r\ `k`\ b))$
$begin :: Category\ k \Rightarrow (a\ `k`\ b) \to Begin^r_k\ a\ b$
$begin\ f = Begin\ (f\circ)$

for the instance deduction and that we can choose r to be the scalar field s knowing that $s \multimap a$ is isomorphic to a for the gradient calculation.

12 Scaling Up

A practical application of differentiation often involves high-dimensional spaces (we can see this in Artificial Neural Networks).
With this in mind, we easily observe that binary products are a very unwieldy and inefficient way of encoding high-dimensional spaces.
A practical alternative is to consider n-ary product as representable functors.

```
class Category k ⇒ MonoidalI k h where
  crossI :: h (a ‘ k ‘ b) → (h a ‘ k ‘ h b)

instance Zip h ⇒ MonoidalI (→) h where
  crossI = 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

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 inl
  jamI = linearD sum jamI
```

13 Implementation

In order to implement several ideas of the article we've had to use a few language extensions.

The majority of the *pseudo-code* in the article was already valid in Haskell with the aforementioned language extensions, but there were still things we had to add to the code.

1. The first thing we thought was to use the already made Category in *Control.Category*, but that proved to not be enough because in the article the author indicates the object types of a Category.

In the already made Category, we could do something with Kinds to achieve this, but it would be strange to specify a Kind Additive with Data Kinds. So we decided to create a new Category, named CategoryDom, with kind $(* \to \mathit{Constraint}) \to (* \to * \to *) \to \mathit{Constraint}$, that receives a class which restricts the objects of a Category and the Category itself. Furthermore, the *id* and $(\circ)$ are restricted by the first argument.

2. The Additive also needed to have an instance with pair of Additive objects, so we could utilise dup without problems.

3. We've had to add Constraints in the context of some instances and change some code to fix certain subtle errors.

The code for the implementation can be found in the git project([2])

14 Future Work

- Generalise the idea to calculate second derivatives.
- Implement the more efficient way that was introduced.

References

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

[2] grupo 114. git project. `https://github.com/Ezequiel-Moreira/LEI_2019_Uminho`, 2019.

[3] Spivak, M. *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus*. Addison-Wesley, 1965.