Comparing implementation strategies for differentiable programming

Bachelor thesis by Daniel Stricker
Date of submission: March 9, 2022

1. Review: Prof. Dr.-Ing. Mira Mezini
2. Review: David Richter, M.Sc.
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Software Technology

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Daniel Stricker, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 9. März 2022

Daniel Stricker

Contents

1 Introduction

1.1 Abstract

Reverse mode differentiation builds the base for every machine learning algorithm. However, reverse mode differentiation is not trivial and therefore many ways to implement it have been and are still developed. Every implementation tries to be the best in some metric like for example easiness, speed or purity. Our main goal of this work is to show the differences and arguably more importantly the similarities between multiple implementations in written code, client API and runtime performance. For this we design our own implementations from scratch but also implement ideas developed by others and try to expand them. The comparison of code and API is done by incrementally developing each implementation by altering or referencing previous implementations to demonstrate differences/similarities on the way.

1.2 Motivation

Machine learning (ML) is a big research area in computer science. Much research on ML is done either directly or indirectly by using it to achieve some other goal/knowledge (e.g. for image recognition or natural language processing). Research on computer languages concerning problems specific to ML could facilitate the implementation of ML applications for many use cases. Gradient descent and therefore differentiable programming builds the base of ML which makes it the perfect target for research because better implementations could lead to improvements in many applications of ML. Having an overview of possible implementations would make it possible to decide which implementation is best suited to design a ML library, algorithm or even (domain specific) language.

Our goal for this work is to produce such an overview by implementing differentiation in multiple ways. These will be partly based on other's work which we (re-)implement and adjust to fit a similar implementation style so that we can easily compare all implementations. Furthermore, we will extend those implementations with new functionalities, change them from the ground up or implement our own designs from scratch. Secondarily

we will try to find use cases of the new language features introduced in Scala 3 to get an idea if this language has potential to aid implementing ML tasks.

1.3 Overview

We provide 10 distinct implementations of differentiation which are based on various (sometimes combined) concepts. 3 of those implement forward mode differentiation where we use simple operator overloading but also experiment with more advanced features like macros. Reverse mode differentiation has 4 implementations which use mutation and 3 which follow a purer approach. Ubiquitous concepts used for this are tapes to store operations, continuation passing style programming and monads.

In detail, we start off with implementing the normal well-known forward mode differentiation in section 2. For this we first give a quick view on how to use a naive but unorthodox way to implement it using macros to replace code in section 2.1. After that we introduce the more common dual numbers in section 2.2 which will be a key ingredient also for later (reverse mode) implementations. The next implementation gets more experimental again. In section 2.3 we use the newly introduced match types and value types of Scala 3 to implement a symbolic differentiation-like way to produce a derivative completely at type level in compile time.

The main focus of this work is reverse mode differentiation (section 3) which is much more relevant for ML tasks than forward mode. Here we begin by introducing the mathematical background of reverse mode by doing it once "by hand". This section is divided into two main parts. One where we use mutation (section 3.1) and one where we refrain from using it (section 3.2).

As using mutation and non-functional programming usually leads to an arguably more straight forward implementation we for now continue without restricting us to functional programming in section 3.1. We introduce continuation passing style (CPS) which at first seems like unnecessary overhead and in itself may be hard to wrap your head around at first. However, it pays off when it elegantly demystifies how to implement reverse mode differentiation in section 3.1.1. Using a tape to store operations on in section 3.1.2 is also a valid way to implement reverse mode in a similar way to CPS. From there we continue by wrapping CPS into a monad in section 3.1.3 to facilitate usage of CPS from the client side. We finish the mutation section by combining the monad and tape idea in section 3.1.4.

For the more puristic oriented in section 3.2 we translate the CPS and monad implementations in section 3.2.1 and section 3.2.2 respectively into a design which does use an accumulator which maps expressions to their adjoints instead of using mutation. The last

implementation (in section 3.2.3) is fully functional and does not even need a unique ID per expression which consequently means it only needs value equality to work. On top of that it is easily comprehensible because it comes without much overhead or complicated abstractions.

In the end we will compare the runtime performance of each implementation in section 4.

To conclude our introduction, we want to give a quick overview over which sections are based on which external work. We based the implementations of dual numbers (section 2.2), CPS (section 3.1.1, section 3.2.1) and tape (section 3.1.2) on the work of Fei Wang et al. [6]. All monad based implementations (section 3.1.3, section 3.1.4, section 3.2.2) are mainly based on our own work but do refer to or build on concepts introduced in the previously mentioned sections and therefore are indirectly based on the work of Fei Wang et al. They additionally use a running example function to showcase differentiation which is also used throughout our work. The work of Matthijs Vákár and Tom Smeding [5] is the base for our CHAD implementation (section 3.2.3). The forward mode implementations using macros (section 2.1) and match types (section 2.3) are implemented from scratch by us.

2 Forward Mode Differentiation

Forward mode differentiation by hand is straight forward and also translates well into code by sticking to our existing knowledge about symbolic differentiation (i.e. differentiation "by hand"). Remember the sum and product rule

$$\text{Sum rule} : (f + g)' = f' + g'$$
$$\text{Product rule} : (f \cdot g)' = f \cdot g' + g' \cdot f$$

and how to differentiate the variable and constants:

$$\text{Variable rule} : \frac{\mathrm{d}x}{\mathrm{d}x} = 1$$
$$\text{Constant rule} : \frac{\mathrm{d}c}{\mathrm{d}x} = 0 \ (c \neq x)$$

These four differentiation rules build the base for our forward mode implementations. Essentially we want to go through every expression recursively and replace it with its derivative.

2.1 Code replacement with macros

If you take the last sentence literally, you could now go the long and hard way (which we did), learn metaprogramming and implement it exactly by replacing expressions with their derivative by directly applying the aforementioned differentiation rules. This approach looks rather ugly at first sight but after ignoring the boilerplate (with the help of the comments on the right-hand side) one can see that it just boils down to recursive pattern matching of code which works pretty intuitively in Scala 3 [1]:

```
1 def d(t: Expr[Term])(using Quotes): Expr[Term] = t match
2     case '{ ($l: Term) + ($r: Term) } =>    // l + r
3         '{ ${ d(l) } + ${ d(r) } }          // d(l) + d(r)
4     case '{ ($l: Term) * ($r: Term) } =>    // l * r
5         '{ $l * ${ d(r) } + ${ d(l) } * $r }// l * d(r) + d(l) * r
6     case '{ X } => '{ 1 }                   // variable
7     case '{ $v: V } => '{ 0 }               // constant
```

Ignore all types of line 1 and just think of `t` as a (sub-)term (or expression for that matter) of a mathematical function we want to differentiate by calling `d`. Now take for example line 2. We match `t` to be a sum of two (sub-)terms (named `l` and `r`). In line 3 we have to define how `t` should be replaced to get the derivative. For that we use the sum rule and essentially return `d(l) + d(r)` which recursively applies the differentiation operator. The main challenge here is to ignore all those braces, dollar signs and apostrophes which are essentially a necessary evil to write macros but lets us intuitively pattern match over *type checked* code which is a really powerful tool. Line 4 and 5 do the same but use the product rule. Line 6 matches the term to be `X` which denotes our variable and is differentiated to 1. Analogously if a value is of type `V` like in line 7, it is a constant which differentiates to 0.

In fact, we would not even need macros and could just write our term with algebraic data types and recursively match them at runtime to implement this approach. This would reduce the boilerplate in comparison to this macro-implementation.

2.2 Operator overloading with dual numbers

Rewriting code that way has one problem. We loose the actual result of our formula and only get the derivative (or we would have to calculate both separately). Consider we have the following function:

```
1 def f(x: Double): Double = 2 * x + x * x * x
```

By writing `f(3)` we can now calculate the result. Our goal is to simultaneously calculate the result and derivative of each subexpression to ultimately accumulate it into the final result and derivative. For this, we could use a structure that consists of two values. Such a construct is called a ”dual number” and consequently has two members, one representing the value of an expression and the other representing the derivative of that expression:

```
1 case class Dual(v: Double, d: Double)
```

To implement operations on dual numbers we have to define the actual computation and additionally how to compute the derivative:

```
1 case class Dual(v: Double, d: Double):
2     def *(that: Dual) = Dual(
3       this.v * that.v,
4       this.v * that.d + this.d * that.v // product rule
5     )
6
7     def +(that: Dual) = Dual(
8       this.v + that.v,
9       this.d + that.d // sum rule
10    )
```

Listing 2.1: Dual number implementation

For multiplication this means that we can just multiply to get the actual result (line 3) and have to apply the product rule in line 4 by accessing `v` to get the actual value and `d` to get the derivative of the child expressions `this` and `that`. Addition works analogously but uses the sum rule. As we can see, this comes down to translating mathematic symbolic differentiation rules into code. How to define constants and the variable (i.e. x) also comes naturally from the differentiation rules as the variable differentiates to 1 and constants to 0:

```
1 def variable(v: Double) = Dual(v, 1)
2 def const(v: Double) = Dual(v, 0)
```

Differentiation of a function f at $x = 3$ could then look like this:

```
1 def differentiate(f: Dual => Dual)(x: Double): Double =
2     val result: Dual = f(variable(x))
3     result.d // result.v would be the actual result of f(x)
4
5 def f(x: Dual): Dual = const(2) * x + x * x * x
6
7 val derivative = differentiate(f)(3)
```

Listing 2.2: Differentiation of a dual number function

The `differentiate` function (line 1) essentially just applies a function to an argument for us (line 2) and then returns `d` (i.e. the derivative) of the result (line 3).

2.3 Match Types

To explain the following approach we first have to look at match types which are essentially functions but at type level. Given a type, it uses patter-matching over it to decide which type to return. It gets clear with the following example taken from the Scala 3 docs [4]:

```
1 type Elem[X] = X match
2     case String => Char
3     case Array[t] => t
4     case Iterable[t] => t
```

The match type `Elem` has one argument `X`. If `X` is `String`, it returns the type `Char` (line 2). If `X` is `Array[t]` or `Iterable[t]`, it returns the generic type parameter of them, that is `t`. In essence, `Elem` matches list-like types and returns their element type. For example, it could be used like this:

```
1 val i: Elem[Array[Int]] = 123
```

`X` is in this case `Array[Int]` and `t` is therefore `Int`. The type of `i` ultimately reduces to `Int`. By recursively matching types we can implement a differentiator match type `D` purely at type level:

```
1 type D[T <: Term] <: Term = T match
2     case l * r =>
3         l * D[r] + D[l] * r
4     case l + r =>
5         D[l] + D[r]
6     case X =>
7         V[1]
8     case V[_] =>
9         V[0]
```

`D` itself and its argument `T` are of type `Term` which is just the super type of all of our expressions. `T` is a type but encodes a full calculation. Consider line 2 where we match `T` to be `l * r`. `l` and `r` are child `Terms` (but are still types). In fact, the multiplication sign (`*`) is an infix type and not a method. It has two type parameters (in this case `l` and `r`) and is also a subtype of `Term`:

```
1 type *[L <: Term, R <: Term] <: Term
```

At this point you could ask yourself, which values those types can have. But we deliberately omit the runtime values of all types because the *complete* differentiation is done at type-level in compile time. The exact values are more or less irrelevant for the implementation logic. A type fully encodes a function and should be seen as some kind of value for now.

In line 3 we apply the product rule by using the matched subexpressions `l` and `r` and recursively using `D` on them. Addition works analogously but uses the sum rule.

Line 7 and 9 still look suspicious. What are integers doing as a type parameter? These are compile time value types which are now supported in Scala 3:

```
1 type V[C <: Double | Int] <: Term
```

`C` is a `Double` or `Int` singleton value type. By using `scala.compiletime.constValue[C]` we can extract the singleton value from a singleton type. With this we can encode (integer) numbers fully at type-level and even calculate with them, also at type-level in compile time. Using this and our differentiation rules for the variable we can match the variable (which is the type `X`) in line 6 and return a type which encodes the number 1. We do the same for constants where we match any constant (denoted by "_") and differentiate it to 0.

If we combine all functionalities described above, we gain the possibility to define a function and produce its derivative entirely at type level:

```
1 type F = X * X
2 type DF = D[F] // -> X * D[X] + D[X] * X -> X * V[1] + V[1] * X
3 // the "symbolic" differentiation is already completed here (at compile
      time)
4
5 // the following is necessary to compute an actual result from the
      derivative (at runtime)
6 val df: DF = initFromType[DF]
7 val result: Double = df(3)
```

The type `F` (line 1) is the function we want to differentiate with respect to `X`. As mentioned before it does *not* need a value. We use our match type `D` to produce the new type `DF` which is equivalent to `X * V[1] + V[1] * X`. That is, because it matches `F` to be multiplication and uses the product rule. At this point the differentiation is completed and `DF` fully represents the differentiated function.

Unfortunately computing a result at type level is impossible because we want to support decimal numbers. Type level calculation with integers on the other hand would be very possible and is included in Scala 3. To allow decimals we have to use a function `initFromType` which recursively matches the provided type and constructs a function value which computes the result later at runtime. Essentially this means that we still have to fall back to actual values at runtime if we want usable results from our implementation and ca not pull through a full implementation at type-level and in compile time.

Additionally, we had a major problem concerning compilation time. Multiple chained multiplications took minutes to compile. At first, we blamed our implementation because we shifted (almost) all of our logic into compile time. An update of the Scala compiler to the newest version (3.0.2 as of now) solved that problem. Compilation time now takes no longer than usual compilations.

3 Reverse mode differentiation

In contrast to forward mode differentiation which benefits highly from our intuition, reverse mode is not straight forward to implement and we even have to work constantly against our intuition.

At first let us introduce our running example function, called y:

$$\begin{aligned} y &= x_1 + x_1 x_2 \\ &= w_1 + w_1 w_2 \\ &= w_1 + w_3 \\ &= \quad w_4 \end{aligned}$$

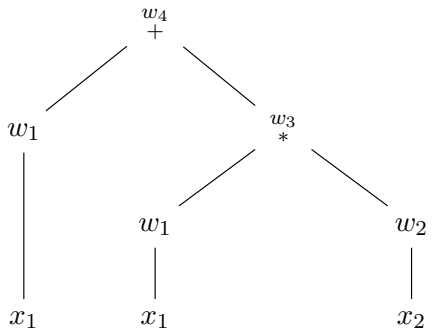
We gave every possible subexpression a name w_i:

$$\begin{aligned} w_1 &\coloneqq x_1 \\ w_2 &\coloneqq x_2 \\ w_3 &\coloneqq w_1 w_2 \\ w_4 &\coloneqq w_1 + w_3 \end{aligned}$$

Note that the outermost expression has the largest index and the innermost expressions have the smallest indices. Order of indices at the same level do not matter. Also notice that each occurrence of a x_i gets the same name as can be seen with x_1 which both got the name w_1. Other expressions which do share the same structure and are therefore equal but are not exactly a x_i would *not* have shared names. This is an important detail for later but does not concern us for now.

For our purpose each expression is either an operation acting on subexpressions (e.g. multiplication or addition) or a value (constant or variable). This can be conveniently visualized as an expression tree with operations as nodes and values as leaves. Both occurrences of w_1 have to have separate nodes to visualize our (later introduced) implementations better:

$$
\begin{aligned}
&= \quad w_4 \\
&= w_1 + w_3 \\
&= w_1 + w_1 w_2 \\
y = x_1 \; &+ x_1 x_2
\end{aligned}
$$



Forward and reverse mode differentiation are built on two different main formulas of concern:

$$\text{Derivative: } \dot{w}_i \coloneqq \frac{\mathrm{d}w_i}{\mathrm{d}x} \qquad\qquad \text{Adjoint: } \overline{w}_i \coloneqq \frac{\mathrm{d}y}{\mathrm{d}w_i}$$

In forward mode we compute the derivative from small i to largest, i.e. from the leaves to the root of the tree. For example if $\dot{w}_1 = \dot{x}_1$ and $\dot{w}_2 = \dot{x}_2$ are given, we can calculate (by using the product rule)

$$\dot{w}_3 = w_1 \dot{w}_2 + \dot{w}_1 w_2.$$

With that we can then find $\dot{w}_4$. Note that as stated above we need to know the initial value of $\dot{x}_1$ and $\dot{x}_2$. If we had one single variable, we would set it to $\dot{x} = \frac{\mathrm{d}x}{\mathrm{d}x} = 1$ and calculate our result. But as we have two variables we have to do two passes through the whole calculation, one with $\dot{x}_1 = 1, \dot{x}_2 = 0$ and one with $\dot{x}_1 = 0, \dot{x}_2 = 1$. Reverse mode does not have to do this. It only has to do one pass to calculate the same result which is the whole motivation to do reverse mode instead of forward mode. In fact for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ forward mode has to do n passes and reverse mode has to do m passes through the whole function. Usually in machine learning tasks you have functions with $n \gg m$ which are also very complex. You certainly want to do as least passes as possible.

For reverse mode we have to shift our focus from $\dot{w}_i$ to another main expression of concern, namely

$$\overline{w}_i \coloneqq \frac{\mathrm{d}y}{\mathrm{d}w_i}$$

also called the adjoint of w_i. Instead of calculating the derivative of a subexpression w_i with respect to x it expresses the derivative of y with respect to a particular subexpression w_i of y. Why this expression concerns us now gets clear after looking at the corresponding usage of the chain rule which both differentiation styles are based on. A quick reminder

on the general definition of the chain rule:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}z}\frac{\mathrm{d}z}{\mathrm{d}x}$$

Forward mode replaces all occurrences of $\dot{w}_i$ by using the chain rule which is the reason why that expression concerned us previously. The chain rule for backward mode is instead used to replace each occurrence of $\overline{w}_i$ recursively:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}w_1}\frac{\mathrm{d}w_1}{\mathrm{d}x} = \left(\frac{\mathrm{d}y}{\mathrm{d}w_2}\frac{\mathrm{d}w_2}{\mathrm{d}w_1}\right)\frac{\mathrm{d}w_1}{\mathrm{d}x} = \left(\left(\frac{\mathrm{d}y}{\mathrm{d}w_3}\frac{\mathrm{d}w_3}{\mathrm{d}w_2}\right)\frac{\mathrm{d}w_2}{\mathrm{d}w_1}\right)\frac{\mathrm{d}w_1}{\mathrm{d}x} = \dots$$

On first sight it might look like we traverse from $i = 1$ to 5. However as one calculates the expression in the innermost parentheses first one can easily see that the iteration actually goes from large to small index (i.e. root to leaves) opposed to forward mode.

This usage of the chain rule essentially dictates how we compute the reverse mode derivative. Our job is to calculate all $\overline{w}_i$ by applying the chain rule recursively until we have rewritten it into an expression including trivial subexpressions or $\overline{w}_j$ with $j > i$ (i.e. ancestor adjoints) which we would have already computed at that point. We use the same y as above and calculate the adjoints of subexpressions w_4 to w_2 by applying the chain rule which constantly introduces parent expressions into the formula:

$$\begin{aligned} y &= x_1 + x_1 x_2 \\ &= w_1 + w_1 w_2 \\ &= w_1 + w_3 \\ &= \quad w_4 \end{aligned}$$

$$\begin{aligned} \overline{w}_4 &= \frac{\mathrm{d}y}{\mathrm{d}w_4} = 1 \\ \overline{w}_3 &= \frac{\mathrm{d}y}{\mathrm{d}w_3} = \frac{\mathrm{d}y}{\mathrm{d}w_4}\frac{\mathrm{d}w_4}{\mathrm{d}w_3} = \overline{w}_4\frac{\mathrm{d}w_4}{\mathrm{d}w_3} = 1 \\ \overline{w}_2 &= \frac{\mathrm{d}y}{\mathrm{d}w_2} = \frac{\mathrm{d}y}{\mathrm{d}w_4}\frac{\mathrm{d}w_4}{\mathrm{d}w_2} = \frac{\mathrm{d}y}{\mathrm{d}w_4}\frac{\mathrm{d}w_4}{\mathrm{d}w_3}\frac{\mathrm{d}w_3}{\mathrm{d}w_2} = \overline{w}_4\frac{\mathrm{d}w_4}{\mathrm{d}w_3}\frac{\mathrm{d}w_3}{\mathrm{d}w_2} = \overline{w}_3\frac{\mathrm{d}w_3}{\mathrm{d}w_2} = w_1 \end{aligned}$$

These were straight forward after recognizing the general pattern. Calculating $\overline{w}_1$ is not as straight forward:

$$\overline{w}_1^a = \frac{\mathrm{d}y}{\mathrm{d}w_1} = \frac{\mathrm{d}y}{\mathrm{d}w_4}\frac{\mathrm{d}w_4}{\mathrm{d}w_1} = \overline{w}_4\frac{\mathrm{d}w_4}{\mathrm{d}w_1} = 1$$
$$\overline{w}_1^b = \frac{\mathrm{d}y}{\mathrm{d}w_1} = \frac{\mathrm{d}y}{\mathrm{d}w_4}\frac{\mathrm{d}w_4}{\mathrm{d}w_1} = \frac{\mathrm{d}y}{\mathrm{d}w_4}\frac{\mathrm{d}w_4}{\mathrm{d}w_3}\frac{\mathrm{d}w_3}{\mathrm{d}w_1} = \overline{w}_4\frac{\mathrm{d}w_4}{\mathrm{d}w_3}\frac{\mathrm{d}w_3}{\mathrm{d}w_1} = \overline{w}_3\frac{\mathrm{d}w_3}{\mathrm{d}w_1} = w_2$$
$$\overline{w}_1 = \overline{w}_1^a + \overline{w}_1^b$$

We had to realise that w_1 appears in two "calculation branches" and had to handle them separately. Lastly these partial adjoints (i.e. $\overline{w}_1^a$ and $\overline{w}_1^b$) of all branches then have to be summed (implying that if w_1 would occur n times, we had to sum n results) into the full adjoint $\overline{w}_1$.

After some consideration and breaking down every step this process is not very complicated. This is true for a human who can overlook the whole expression including all its subexpressions. A program on the other hand often only has a limited view of the whole expression. Consider this translation of our example into code:

```
1 def w3 = w1 * w2
2 def y = w1 + w3 // w4
```

At definition time of `w3` we do not have enough information to calculate a full andjoint because only after adding all context with the definition of `y` we know all branches where `w1` occurs. In fact `y` could also be just another subexpression of a bigger calculation. Usually when evaluating expressions you can start evaluating the innermost subexpression and use its result to evaluate its containing expression as we did for forward mode differentiation. This had the advantage that we could calculate the normal result values and the derivative of each subexpression simultaneously. Unfortunately this is not possible for reverse mode and directly using dual numbers as is does not suffice. We have to start with the top expression w_4 and have to work our way down to (all occurrences of) w_1 (and w_2). This is very unnatural to implement because the information flows in reverse order. On top of that when iterating from outer to inner expression we can no longer calculate the normal result. Naturally at some point we have to calculate these values too. The only option we have is to do a full *forward pass* (inner to outer) to calculate the result values and another full *backward pass* (outer to inner) to calculate the (partial) adjoints of every subexpression (also see figure 3.1). Fortunately the forward pass is trivial as it only has to calculate arithmetic results in usual recursive order.

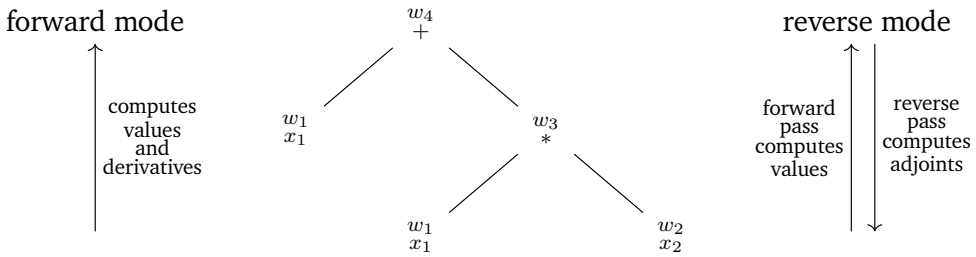


Figure 3.1: Information flow of forward and reverse mode

The main takeaway from this extensive example is an important pattern which we will be utilizing to implement the reverse pass. The second last term of every calculation of $\overline{w}_i^z$ (i.e. the partial adjoint of the z-th occurrence of w_i) with $z \in \{a, b, \dots\}$ has always the following pattern:

$$\overline{w}_i^z = \overline{w}_p \frac{\mathrm{d}w_p}{\mathrm{d}w_i^z}$$

for some p. This p (*parent index*) is not at all random. w_p is always the parent expression of that specific occurrence w_i^z of w_i. We use the superscript to distinguish specific occurrences. We also could have written y like (notice the added superscripts a and b)

$$\begin{aligned} y &= x_1 + x_1 x_2 \\ &= w_1^a + w_1^b w_2^a \\ &= w_1^a + w_3^a \\ &= \quad w_4^a \end{aligned}$$

to further emphasize distinct occurrences of expressions w_i. Usually we omit the superscript if that expression only occurs once. Actually w_i can only occur multiple times (i.e. have a "b" superscript) if $w_i = x_i$, i.e the expression is one of our variables. In other words: Equal expressions are not counted as multiple occurrences and for that matter only x_i count.

When boiling down

$$\overline{w}_i^z = \overline{w}_p \frac{\mathrm{d}w_p}{\mathrm{d}w_i^z}$$

further we realise that we have to compute the derivative of the parent expression with respect to w_i^z. This is fortunately comparatively easy. A parent expression will always be an atomic operation (e.g. (*), (+), (-), sin) and w_i^z is always a direct argument. Because we usually know the derivative of each of our atomic operations, we can simply handle

every case, e.g. if w_p is a multiplication expression (and it has some right-hand side w_k), we can simply compute:

$$\frac{\mathrm{d}w_p}{\mathrm{d}w_i^z} = \frac{\mathrm{d}(w_i^z \cdot w_k)}{\mathrm{d}w_i^z} = w_k$$

The remaining and main task is to find $\overline{w}_p$, the adjoint of the parent expression (i.e. the derivative of y with respect to w_p). This is a recursive problem but unfortunately in reverse order because information flows from outer expression to inner which is the reason why this is called the reverse pass as already illustrated in figure 3.1. Solving this "reversed flow of information" to calculate $\overline{w}_p$ elegantly, efficiently or easily to reason about is the main goal of the following implementations.

3.1 Using mutation

3.1.1 Continuation Passing Style (CPS)

Continuation is just an elaborate term for the frequently used callbacks (e.g. for frontend web development). Essentially you pass the "rest of the calculation" to the function instead of using its return value and manually applying the rest on that result. To make things clear consider chaining two arbitrary functions (with unspecified types `A, B, C, D`) as usual on the left-hand side and an equal implementation but in CPS on the right-hand side:

```
1 def first(x: A): B
2 def second(x: B): D
3
4 def chained(x: A): D =
5   val firstResult: B = first(a)
6   return second(firstResult)
7
8 val a: A = ???
9 val chainedResult: D =
10  chained(a)
```

Listing 3.1: Ordinary chaining

```
1 def first[R](x: A)(rest: B => R): R
2 def second[R](x: B)(rest: D => R): R
3
4 def chained[R](x: A)(ret: D => R): R =
5   first(a) { (firstResult: B) =>
6     second(firstResult) { ret }
7   }
8 val a: A = ???
9 val chainedResult: D =
10  chained(a) { identity }
```

Listing 3.2: CPS chaining

On the left-hand side we first define two functions which have unspecified implementations. The types are the important part. `first` takes `A` and returns `B`. `B` is in turn the input type of `second` which is important because this makes both functions chainable. `chained` (line 4) calls `first` (line 5) and passes its result to `second` (line 6) which is essentially the definition of function chaining. Lines 8 to 10 just visualize how `chained` is then used.

The right-hand side looks somewhat similar but has significant changes. Every function now has a second argument, i.e. the continuation which we usually call `rest`. Notice that the *input* type of `rest` in `first` of listing 3.2 is `B`. This matches the *result* type of `first` in listing 3.1 (and analogously for `D` and `second`). Also note that we had to introduce type parameter `R` to all functions. This is needed to support arbitrary `rest` functions even if they do not return exactly `D`. `chained` works similarly but looks very different. We also call `first` in line 5 but instead of creating a new named (constant) variable we have to pass a lambda where the parameter takes the role of the variable. After that we call `second` and pass `ret` as `rest` which in this case has an equal semantic to the `return` statement in line 6 of the left-hand side. In line 10 of listing 3.2 we pass `identity` to `chained` to mark the end of the calculation because it acts like a no-op. We could have passed another arbitrary operation instead, similarly to how we could have applied more arbitrary operations on `chainedResult` in listing 3.1. When following CPS strictly, every function takes a continuation and ordinary variables are never used. Lambdas with named parameters fulfill that role instead, as seen with `firstResult` in line 5.

Using CPS which is an at first glance rather obscure feature we can implement reverse mode using dual numbers:

```
1  case class Dual(x: Double, var adjoint: Double):
2      def *(that: Dual)(k: Dual => Dual): Dual =
3          // w_p
4          val localResult = Dual(this.x * that.x, 0)
5
6          val globalResult = k(localResult)
7
8          // i ∈ {this, that}
9          def addPartialAdjoint(
10             thisOrThat: Dual,
11             derivativeWrtThisOrThat: Double
12         ): Unit =
13             // ∂̄z_i = ∂̄w_p * dw_p/dz_i
14             val partialAdjoint =
15                 localResult.adjoint * derivativeWrtThisOrThat
16             // ∂̄w_i += ∂̄z_i
17             thisOrThat.adjoint += partialAdjoint
18         end addPartialAdjoint
19
20         addPartialAdjoint(this, that.x) // i = this
21         addPartialAdjoint(that, this.x) // i = that
22         globalResult
```

```
23      end *
24
25      // Analogous to (*)
26      def +(that: Dual)(k: Dual => Dual): Dual = ???
27  end Dual
```

Listing 3.3: Reverse mode CPS

Compared to forward dual numbers (listing 2.1) we changed the name of the second member of `Dual` to `adjoint` to reflect the shift in focus from $\dot{w}_i$ to $\overline{w}_i$. The comments add translations from code expressions into their mathematic notation from section 3. w_p in line 13 signifies the parent expression, e.g. for multiplication we would write it like this:

$$\begin{aligned} w_{\text{this}} &= \texttt{this} \\ w_{\text{that}} &= \texttt{that} \\ w_p &= w_{\text{this}} \cdot w_{\text{that}} \end{aligned}$$

The helper function `addPartialAdjoint` (line 9) essentially just executes the mathematical expressions in lines 13 and 16 but generalizes over for which subexpression ($\overline{w}^z_{\text{this}}$ or $\overline{w}^z_{\text{that}}$) to compute the partial adjoint. The `partialAdjoint` (line 14) is the adjoint of this specific occurrence of w^z_i. Remember that we have to sum the partial adjoints of all occurrence to get the full adjoint. Exactly that happens in line 17 by mutating `thisOrThat.adjoint`. Every expression is responsible to add the partial adjoints of their subexpressions. By translating every code piece into its corresponding mathematical notation, one can clearly see the close relationship between our implementation and the mathematical foundations of reverse mode differentiation seen in section 3.

The main difference is line 6 where we call the continuation `k`. It represents the rest of the computation as stated previously. In this context specifically, it represents all further operations that might use the passed `localResult`. Note that those further operations are ancestors and not children or in other words they are outer and not inner operations. The first job of the continuation is to do a *forward pass* through the rest of the operations. This is done by calculating the regular result (line 4) of the next operation and afterwards calling the next continuation. This recursive forward pass eventually finishes by calling a `k` which does not call another continuation. At this point we found the recursion anchor and have built a stack of calls as usual with recursive algorithms. This built-up stack now naturally tears down in *reverse order*. This is exactly our primary goal. We had to calculate the regular results in "normal" order (i.e. inner to outer expression) but the adjoint is naturally calculated in *reverse order* (as seen in figure 3.1). The built-up stack is visualized in figure 3.2 for our running example. It tears down from top to bottom.

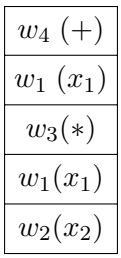


Figure 3.2: Expression stack after the forward pass

From here on therefore the *reverse pass* starts. On top of the stack now resides w_4. Lines 9 to 20 are executed to update the adjoint of its subexpressions (w_1 and w_3). After that, w_1 and then w_3 and its whole expression tree on the stack is handled. As you can see, after doing the forward pass by abusing continuations we now iterate through each expression in the same order as we would do when doing reverse differentiation by hand. Essentially we have linearized the expression tree to make the reverse pass and therefore adjoint accumulation easier. We use some kind of linearization in almost every reverse mode implementation.

In the end we define a `differentiate` operator and call it:

```
1  def differentiate(f: Dual => (Dual => Dual) => Dual)(x: Double): Double =
2      val xDual = Dual(x, 0)
3
4      // Use only side effects
5      f(xDual) { topExpression => {
6          // Manually set adjoint of top-most expression
7          topExpression.adjoint = 1 // ȳ = dy/dy
8          topExpression // y
9      }
10     }
11     xDual.adjoint // x̄ = dy/dx
12 end differentiate
13
14 def f(x: Dual)(k: Dual => Dual): Dual =
15     // 2 * x + x * x * x
16     (Dual(2, 0) * x) { y1 =>
17         (x * x) { y2 =>
18             (y2 * x) { y3 =>
19                 (y1 + y3) { k }
20             }
```

```
21          }
22      }
23  end f
24
25  val derivative: Double = differentiate(f)(3)
```

`differentiate` in line 1 takes the function `f` we want to differentiate as its first argument. Because we follow CPS `f` has two inputs, first the input for `x` of type `Dual` and second the continuation of type `(Dual => Dual)` which is called by `f` itself after calculating its result. The result of the continuation is the final result of type `Dual`. `differentiate` also gets the `Double` value passed at which we want to differentiate `f`. In line 2 we create a `Dual` from `x`. Its initial adjoint has to be zero because its partial adjoints will get added to it by mutation later. The continuation passed to `f` in line 5 is basically just an identity function to mark the top most expression and to act as the recursion anchor. We just have to additionally set the adjoint of the top expression to 1 because our program would not know which the top expression is. Another point to notice is that we do not directly use the result of `f` and instead read the mutated adjoint of x in line 11. This makes sense because `f` returns the result of the *top* expression. Its adjoint is trivially 1 (line 7) and therefore is not interesting while the adjoint of x (line 11) is exactly the derivative of `f`.

The biggest disadvantage of this CPS implementation is how one has to write the function `f` compared to for example our forward mode dual number implementation in listing 2.2. Reading CPS is not impossible because one "just" has to read every line from right to left. For example consider line 17. On the right-hand side is the variable name (`y2`) and on the left-hand side its "value" (`x * x`). We also have to give every subexpression a name which we would normally not need to and also often do not want to. Another problem is the deep nesting which occurs for more elaborate functions. A CPS function is therefore cumbersome to write and reading them needs some time getting used to. Those problems could be solved by using shift and reset operators which principally would make continuations implicit and would hide them completely from client code. Unfortunately there is currently no maintained implementation of them for Scala. Hence, we would like to refer to Fei Wang et al. [6] and their implementation of reverse mode differentiation using shift and reset operators without further going into it here.

3.1.2 Tape

The following implementation is very similar to CPS but instead of building a stack of calls implicitly by calling continuations we build that "call stack" manually. Remember that the only goal we achieved by using continuations was a two pass design which we used to do some operations (compute regular result) in normal order and some operations (compute

adjoint) in reverse order through the expression tree. Another way to achieve this is to do the forward pass as usual but on the way additionally save all operations which have to be done in the reverse pass for later. When we have collected every operation we just execute them in "reverse" order:

```
1  var tape: Unit => Unit = _ => ()
2
3  case class Dual(x: Double, var adjoint: Double):
4      def *(that: Dual): Dual =
5          val localResult = Dual(this.x * that.x, 0)
6
7          def addPartialAdjoint(
8              thisOrThat: Dual,
9              derivativeWrtThisOrThat: Double
10         ): Unit => Unit =
11             _ =>
12                 val partialAdjoint =
13                     localResult.adjoint * derivativeWrtThisOrThat
14                 thisOrThat.adjoint += partialAdjoint
15         end addPartialAdjoint
16
17         tape = addPartialAdjoint(this, that.x) andThen tape
18         tape = addPartialAdjoint(that, this.x) andThen tape
19
20         localResult
21     end *
22
23     def +(that: Dual): Dual = ???
24 end Dual
```

In line 1 we define a mutable `tape` which we use to store operations on. These operations can only produce side effects because the `tape` has type `Unit => Unit` which cannot take nor return anything meaningful. We initialize it with a no-op. The first part of the multiplication (lines 5 to 15) which includes `addPartialAdjoint` are in essence equal to the according lines in CPS, and therefore we will just highlight the differences. We also omitted the mathematical translations. They are still important to get the connection to the mathematical foundations but for them refer to listing 3.3 as they are very similar.

First thing to note is the altered return type of `addPartialAdjoint` (line 10). It now returns a function which in turn is just used for its side effects (`Unit => Unit`). This means that when we call `addPartialAdjoint` (lines 17 and 18) the adjoint is *not* directly updated opposed to CPS. Instead, we prepend that "operation" (calculating and updating the adjoint of `this` or `that`) to `tape`. We prepend (instead of appending) so that in the end we have a tape which executes each operation in reverse order of insertion.

Our `differentiate` function is again similar to CPS:

```
1 def differentiate(f: Dual => Dual)(x: Double): Double =
2     tape = _ => ()
3     val xDual: Dual = Dual(x, 0)
4     val topExpression = f(xDual)
5     topExpression.adjoint = 1
6     tape(())
7     xDual.adjoint
8 end differentiate
9
10 def f(x: Dual): Dual =
11     Dual(2, 0) * x + x * x * x
12
13 val derivative = differentiate(f)(3)
```

This time `differentiate` takes a simpler `f` as its first argument because we do not use continuations anymore. It now just has a `Dual` input and calculates a `Dual`. Because `tape` is a global variable we have to remember to reset it for every differentiation (line 2). We then call `f` (line 4) to do the forward pass and to populate the `tape`. Similar to CPS we have to manually set the adjoint of the top expression to $1 = \frac{\mathrm{d}y}{\mathrm{d}y}$ (line 5). At this point no differentiation has been done yet. We have to call `tape` to start it manually (as it takes a `Unit` we have to pass its only inhabitant, namely "`()`"). The definition of `f` (line 10 and 11) is possibly the most interesting change. We do not need any continuations and can omit variable names which makes it easier to read and write.

To make the definition of `f` even more regular we can define an implicit conversion which converts a constant into `Dual` automatically. For this we use `given` instances [3] of `Conversion` [2] which were introduced in Scala 3. They specifically describe the intent to convert a value. Previously `implicit` methods were used for this but their semantics were overloaded and for example have also been used to define extension methods. The first `given` instance (line 1) is not needed for this example but is included for completeness if one uses decimal numbers:

```
1 given Conversion[Double, Dual] = Dual(_, 0)
2 given Conversion[Int, Dual] = Dual(_, 0)
3
4 def f(x: Dual): Dual =
5     // 2 is implicitly converted into Dual(2, 0)
6     2 * x + x * x * x
```

So far we have only done reverse mode differentiation for one variable. As mentioned previously reverse mode differentiation shines when having multiple input variables. Therefore, it's apparent to make an example which supports that. Extending the tape implementation to take multiple variables is mostly trivial as we only have to change the `differentiate` function:

```
1  def differentiate(
2      f: List[Dual] => Dual,
3      xs: List[Double]
4  ): List[Double] =
5      tape = _ => ()
6      val xsDual: List[Dual] = xs map { Dual(_, 0) }
7      f(xsDual).adjoint = 1
8      tape(())
9      xsDual map { _.adjoint }
10 end differentiate
11
12 def f(xs: List[Dual]): Dual =
13     2 * xs(0) + xs(1) * xs(2) * xs(2)
14
15 val derivatives: List[Double] = differentiate(f, List(3.0, 5.0, 2.0))
```

We encode multiple variables as a single vector of type `List[Dual]`. Because `f` now takes a `List[Dual]`, `differentiate` has to reflect that by accepting a function `List[Dual] => Dual` and a vector of values to differentiate `f` at. At first, we have to reset the tape again (line 5). In line 6 we extract the `adjoint` of each variable which ultimately gives us a vector where every value is the derivative with respect to one variable. In other words we computed the gradient of `f`. The main takeaway here is that we computed the derivative of multiple variables in one go without having to call `differentiate` multiple times with different values. This is only possible with reverse mode and is its main advantage. Forward mode would have to do one full differentiation for each variable where all other variable are set to 0.

Extending other reverse mode implementations for multidimensional functions (in input or output) is done analogously. To allow better focus on the essential differences and keep the examples simple we mostly concentrate on single dimensional functions from here on.

3.1.3 Monad CPS

We already established that writing CPS-functions by hand is cumbersome and not easily readable. We do not want to write deeply nested lambdas only to represent simple

arithmetic calculations. Optimally we want to write arithmetic functions without a syntactic constraint, for example like this:

```
2 * x + x * x * x
```

Because this is not translatable into CPS easily, the next best thing would be a syntax which resembles common usage of Scala to utilize our inherent intuition instead of breaking it. We introduce a named value for every subexpression. This does not change any semantics but conveniently separates each subexpression into its own line and own syntactic construct:

```
val y1 = x * 2
val y2 = x * x
val y3 = y2 * x
val y4 = y1 + y3
```

Admittedly introducing a mandatory value name for every subexpression is not as elegant as directly writing down the expression. But value definitions are so ubiquitous that writing and reading them is at least very intuitive. The syntax should therefore look similar to this. Notice that we have only "defined" how we would like our code to look like and have not solved our problem yet as continuations are nowhere to be found yet. However we are not as far away from CPS as one could think. Remember how every continuation also has a mandatory named parameter which represents the result of the last calculation. By introducing mandatory named value definitions we have a somewhat similar situation at hand. Our goal is now to automatically rewrite these imperative value definitions into a CPS construct where each `val` is translated into a continuation parameter and each subexpression is nested into the continuation of the last one. Turns out Scala's for-comprehensions, if used in a specific way, can do exactly that. We mainly make use of the fact that a for-comprehension (without any guards) is entirely desugared into calls of multiple nested `flatMaps` and one concluding `map` for the `yield`. If we manage to implement those two methods for our dual numbers, which is essentially equivalent to implementing a monad, we can write a function like this:

```
def f(x: Dual): DualMonad
    for
        y1 <- x * 2
        y2 <- x * x
        y3 <- y2 * x
        y4 <- y1 + y3
    yield y4
```

`f` now returns a `Monad` which wraps a dual number. Except of changed syntax the code is essentially similar to the imperative code of the last listing. The compiler then desugars it into this:

```
def f(x: Dual): DualMonad =
    (x * 2).flatMap { y1 =>
        (x * x).flatMap { y2 =>
            (y2 * x).flatMap { y3 =>
                (y1 + y3).map { y4 =>
                    y4
                }
            }
        }
    }
end f
```

Listing 3.4: Desugared for-comprehension

At this point it should get clear why we wanted to use for-comprehensions to abstract over CPS. The compiler does the hard work for us and almost exactly translates a for-comprehension into CPS. Implementing `flatMap` and `map` (and thereby a monad) is the last (and main) task. The remaining code structure matches our previous CPS implementation.

Let us first look at the desired general signature for `flatMap` and `map` of a general monad:

```
trait Monad[A]:
    def flatMap[B](f: A => Monad[B]): Monad[B] = ???
    def map[B](f: A => B): Monad[B] = ???
```

`A` represents the value we are wrapping with the monad and `B` is an arbitrary new type (possibly same as `A`) which the value of `A` is converted to. Both methods return a new monad which now wraps `B`. In essence both methods model the mutation of the wrapped value and allow for the value type to change. The difference lies in the function passed

to them. While the passed function to `flatMap` returns a monad, the passed function to `map` only returns a new value and `map` has to wrap `B` itself to be ultimately able to return `Monad[B]`.

A monad defined like this is very versatile because all value types are parameterized. This is important when using advanced capabilities of monads and to understand the concept in itself. For our use case on the other hand this is clearly excessive and can be simplified. The only value type we work on is `Dual` and therefore we can replace all occurrences of `A` and `B` with simply `Dual`. `DualMonad` can be simply interpreted like an alias for `Monad[Dual]`. For our purposes this is enough and makes the code easier to read without sacrificing expressiveness:

```
trait DualMonad:
    def flatMap(k: Dual => DualMonad): DualMonad = ???
    def map(k: Dual => Dual): DualMonad = ???
```

We also renamed the passed functions into `k` because they clearly represent continuations like one can easily see when looking at the desugared for-comprehension in listing 3.4. Let us look at the full implementation of `Dual` and how we could implement `DualMonad` :

```
case class Dual(x: Double, var adjoint: Double):
    thisDual =>

    def *(thatDual: Dual): DualMonad = new DualMonad {
      override def flatMap(k: Dual => DualMonad): DualMonad =
        val parent = Dual(thisDual.x * thatDual.x, 0)
        val result = k(parent)
        thisDual.adjoint += thatDual.x * parent.adjoint
        thatDual.adjoint += thisDual.x * parent.adjoint
        result

      override def map(k: Dual => Dual): DualMonad =
        def wrap(dual: Dual): DualMonad = dual * 1
        flatMap(k andThen wrap)
    }

    def +(r: Dual): DualMonad = ???
end Dual
```

First notice that we do not implement `DualMonad` at top level and in fact only implement it ad hoc when calling an operation on `Dual`s. We use this to override `flatMap` with the

main operation logic which was found directly as part of `*` (or `+`) in previous reverse mode implementations. Because multiplication and addition have different logic we implement them ad hoc. When inspecting `flatMap` further we realize that it is almost equivalent to the code of `*` from the CPS implementation in listing 3.3. We calculate the parent result (line 6), call the continuation `k` to build the "call stack" (line 7) and then use the reverse pass to update the adjoints of `thisDual` and `thatDual` using the adjoint of the parent expression (lines 8-9). Note that we appended `-Dual` to the instance name in line 2 to prevent a name clash with identifier `this` when we are inside the ad hoc definition of `DualMonad`. The only but important difference is that we eventually return the `DualMonad` we got from the continuation in line 7. Returning a `DualMonad` allows us to chain multiple `flatMap` calls together which in turn means chaining multiple operations just like with normal CPS.

The next method we override is `map` (line 12). The important signature difference to `flatMap` is that it gets a function which returns a `Dual` directly instead of a wrapped `DualMonad`. Usually this is used to allow a last operation directly on `Dual` before ending the for-comprehension. In our case this is not needed because each of our operations on `Dual` return `DualMonad` and not `Dual`. Because of this the only `k` which can be passed is a function which returns its argument or another `Dual` captured by its closure. Essentially our job is to wrap the result of `k` into a monad without changing the semantics. Naively one would try to implement `DualMonad` ad hoc as we did with `flatMap`. One unfortunately quickly realizes that we would have to implement another `DualMonad` in the nested `map` which leads us into an infinite loop of implementations. A solution for this is to apply a trivial identity function like multiplying with 1 to wrap a `Dual` into a `DualMonad` (line 13). With the ability to wrap a `Dual` we can directly use the already implemented `flatMap`. We just pass `k` to it but wrap `k`'s result into a `DualMonad` to conform to `flatMap`'s signature.

We successfully implemented a Monad and now just need a `differentiate` function which sets the adjoint of the top expression to 1 just like in previous implementations. Additionally it has to wrap the top expression into a `DualMonad` manually by again using a no-op multiplication (line 11):

```
1 given Conversion[Double, Dual] = Dual(_, 0)
2 given Conversion[Int, Dual] = Dual(_, 0)
3
4 def differentiate(
5     f: Dual => (Dual => DualMonad) => DualMonad,
6     x: Double
7 ): Double =
8     val xDual = Dual(x, 0)
```

```
    f(xDual) { topExpression =>
        topExpression.adjoint = 1
        topExpression * 1
    }
    xDual.adjoint
end differentiate
```

Note that the expected function `f` again has a second argument, the continuation (`Dual => DualMonad`) which now returns a `DualMonad`. Finally, we can use a for-comprehension to define `f` to ultimately let the compiler rewrite our code into a CPS-like structure which was our goal:

```
def f(x: Dual)(k: Dual => DualMonad): DualMonad =
    for
        y1 <- x * 2
        y2 <- x * x
        y3 <- y2 * x
        y4 <- y1 + y3
        y5 <- k(y4)
    yield y5
end f

val derivative = differentiate(f, 3)
```

3.1.4 Combined Monad and Tape

Using for-comprehensions made the inside of a function more readable but when defining `f` we still needed a continuation `k` as its second parameter. This is not only ugly but makes it very hard to compose multiple functions. Instead, we want a simple way to define two functions `f` and `g` without seeing continuations at all and make chaining them as easy as writing `f andThen g` as we are used to in Scala. We try to solve this by coming back to our tape based approach and combining it with our monad based one.

At first, we need our tape where we save all adjoint updates which have to happen in reverse order. This is exactly same to our first tape based implementation:

```
var tape: Unit => Unit = _ => {}
```

Now we get back to monads. Previously we implemented `DualMonad` ad hoc to overcome the different requirements of multiplication and addition. Those differences can be abstracted over which leads to a much cleaner and more reusable design. Essentially

what we need to abstract over is the result of a parent calculation and how to update the adjoints of the child expressions using the parent adjoint. Those are both realized as members of `DualMonad` and are called `parent` and `adjointsUpdater` respectively (line 1):

```
1  class DualMonad(val parent: Dual, val adjointsUpdater: Dual => Unit):
2      def flatMap(k: Dual => DualMonad): DualMonad =
3          tape = ((_: Unit) => adjointsUpdater(parent)) andThen tape
4          k(parent)
5
6      def map(k: Dual => Dual): DualMonad =
7          flatMap(k andThen wrap)
8  end DualMonad
9
10 def wrap(dual: Dual): DualMonad = DualMonad(dual, identity)
```

Listing 3.5: Monad using tape

`flatMap` prepends the adjoint update operation to the tape (line 3) which is exactly what we did for our first tape implementation and then calls the continuation (line 4). `map` is very similar to our first monad approach as it also just uses `flatMap` after wrapping the result of `k` (line 7). The difference is that instead of doing a no-op operation on `Dual` we wrap it manually by passing a no-op identity function as the `adjointUpdater` in the `wrap` function (line 10).

We have done the hardest task and now only have to glue the pieces together:

```
case class Dual(x: Double, var adjoint: Double):
    def *(that: Dual): DualMonad =
        def addPartialAdjoint(
            thisOrThat: Dual,
            derivativeWrtThisOrThat: Double,
            parentAdjoint: Double
        ): Unit =
            val partialAdjoint = parentAdjoint * derivativeWrtThisOrThat
            thisOrThat.adjoint += partialAdjoint
        end addPartialAdjoint

        DualMonad(
            this.x * that.x,
            parent =>
                addPartialAdjoint(this, that.x, parent.adjoint)
                addPartialAdjoint(that, this.x, parent.adjoint)
        )
```

```
18      end *
19
20      def +(r: Dual): DualMonad = ???
21  end Dual
```

`addPartialAdjoint` (line 3) is equal to previous implementations. We pass the normal result (line 13) and as usual how to update the adjoints of `this` and `that` to the constructor of `DualMonad` (lines 14 to 16).

The biggest and most important change comes in the signature of `differentiate`. The expected function `f` uses no continuation and just expects and returns a `DualMonad` which is exactly what we wanted:

```
1 def differentiate(f: DualMonad => DualMonad)(x: Double): Double =
2     tape = _ => ()
3     val xDualMonad = wrap(Dual(x, 0))
4     f(xDualMonad).parent.adjoint = 1
5     tape(())
6     xDualMonad.parent.adjoint
```

The implementation of it on the other hand is nothing special. Again, we have to remember to reset the tape (line 2). After wrapping a `Double` into a `Dual` and then into a `DualMonad` (line 3) we have to call `f` with it to do the forward pass and fill the tape (line 4). After setting the adjoint of the top expression (also line 4) we can execute the tape (and start the reverse pass) (line 5) which sets the adjoint of our argument to `f`.

Because the expected function gets a `DualMonad` and also returns one we can now easily chain two functions using `andThen` (line 18) and still use for-comprehensions:

```
1  def f(xM: DualMonad): DualMonad =
2      for
3          x <- xM
4          y1 <- x * 2
5          y2 <- x * x
6          y3 <- y2 * x
7          y4 <- y1 + y3
8      yield y4
9  end f
10
11 def g(xM: DualMonad): DualMonad =
12     for
13         x <- xM
14         y <- x * x
15     yield y
```

```
16 end g
17
18 val derivative = differentiate(f andThen g)(3)
```

Note that one can interpret the first lines of the for-comprehensions (lines 3 and 13) as "unwrapping" the monad back into a `Dual`.

3.2 Without mutation

3.2.1 Continuation Passing Style (CPS)

To eliminate mutation we have to firstly detect where mutation even occurs. Looking at the implementation of mutable CPS (listing 3.3) we directly find the culprit in the first line. The second member of `Dual`, namely `adjoint`, is mutable. As a matter of fact that is the only mutable state we use and therefore we have to remove it. `Dual` is no longer a dual number when we remove its second member so we rename it to just `Num` accordingly. Simply changing `adjoint` to a `val` would not suffice because we cannot know the adjoint of a `Dual` at creation time because the adjoint has to be *accumulated* from multiple branches. Adding an accumulator which is recursively propagated is consequently the logical next step. For this step we have to remember the obvious but critical fact that the reverse pass is indeed *reverse* and we want to propagate the accumulator from parent to children. This is opposed to usual recursive algorithms where each child passes the accumulator to its parent until it reaches the top expression. The top expression uses the full accumulator to produce the full result. To achieve the reversed recursive pass through all expressions we still abuse continuations to build a stack which is naturally traversed in reverse order. The remaining task is to propagate the accumulator correctly. Let us look at the finished code and work along it to understand how one could implement that:

```
1  type Adjoints = Map[Num, Double]
2  type Continuation = Num => Adjoints
3
4  class Num(val x: Double):
5      def *(that: Num)(k: Continuation): Adjoints =
6          val parent = Num(this.x * that.x)
7
8          def addPartialAdjoint(
9              thisOrThat: Num,
10             derivativeWrtThisOrThat: Double,
11             adjoints: Adjoints
12         ): Adjoints =
13             val partialAdjoint =
14                 adjoints(parent) * derivativeWrtThisOrThat
15             val newAdjointThisOrThat =
16                 adjoints(thisOrThat) + partialAdjoint
17             adjoints + (thisOrThat -> newAdjointThisOrThat)
18         end addPartialAdjoint
19
20         val adjointsWithParent = k(parent)
21         val adjointsWithThis =
22             addPartialAdjoint(this, that.x, adjointsWithParent)
23         val adjointsWithThat =
24             addPartialAdjoint(that, this.x, adjointsWithThis)
25         adjointsWithThat
26     end *
27
28     def +(that: Num)(k: Continuation): Adjoint = ???
29 end Num
```

In line 1 and 2 we assign names to two important types so that we can reason more easily about this implementation. The accumulator we use is of type `Adjoints` and maps each expression to its currently accumulated adjoint. `Continuation` is exactly that, the continuation of our program as seen before but with one major change. In previous implementations continuations returned the actual calculated result hence the type `Dual => Dual` (listing 3.3). This seems somewhat convenient at first but we ignored the final result anyway (and could have used `Unit` instead). Remember that continuations represent the calculation of ancestor expressions and so is perfectly suited to pass the currently accumulated adjoints to its descendants. Therefore continuations now return `Adjoints` instead of `Num`.

The helper function `addPartialAdjoint` (line 8) is very similar to the mutable approach (listing 3.3) but because `Adjoints` is immutable it has to take the current adjoints as a parameter (line 11) and return an updated version with the added partial adjoint for

`thisOrThat` (line 17). Line 13 just computes the current partial adjoint using the adjoint of the parent expression which is exactly what happened in the mutable CPS implementation. Line 15 to 17 essentially just add `partialAdjoint` onto the current value in the `adjoints` accumulator. This translates exactly to the `+=` operation used in the mutable version (listing 3.3).

Line 20 calls the continuation to get the adjoint of the current parent (and all other accumulated ancestor adjoints, potentially including `this` or `that`). Remember that at this point the stack builds up and the following lines are executed from top expression to children expressions. `addPartialAdjoint` is called twice very similar to the mutable approach (lines 21 to 24). The most important part here is to always pass the latest updated adjoints to the next call. In the end we effectively return all adjoints of all ancestors we got from `k` and also the updated adjoints of `this` and `that`. After we return, the next descendant now gets those updated adjoints from its call to its `k`.

Finally we just have to define the usual `differentiate` function. Note that the expected `f` again expects a continuation and instead of returning the actual result of the calculation it now returns `Adjoints` which contains all accumulated adjoints (line 5). As always we have to manually set the adjoint of the top expression to 1. In this case this is done by initializing the `Adjoints` map with the `topExpression` as key and 1 as value (line 10). Setting the default value of the map to 0 (also line 10) is also important. Basically it initializes all adjoints of all expressions to 0 so that we can sum all partial adjoints without checking first if the key already exists in the map.

```
1  given Conversion[Double, Num] = Num(_)
2  given Conversion[Int, Num] = Num(_)
3
4  def differentiate(
5      f: Num => Continuation => Adjoints,
6      x: Double
7  ): Double =
8      val xNum = Num(x)
9      val allAdjoints = f(xNum) { (topExpression: Num) =>
10         Map(topExpression -> 1.0).withDefaultValue(0)
11     }
12     allAdjoints(xNum)
13
14 def f(x: Num)(k: Continuation): Adjoints =
15     (2 * x) { y1 =>
16         (x * x) { y2 =>
17             (y2 * x) { y3 =>
18                 (y1 + y3) { k }
```

```
19                 }
20             }
21         }
22
23 val derivative = differentiate(f, 3)
```

3.2.2 Monad

Next we want to take a look at our approach where we combined monad and tape and remove every mutation. Similar to CPS in the last section we have the mutable member `adjoint` of `Dual`. Our solution for it was a map which functions as an adjoint accumulator and replaces the `adjoint` member of `Dual`. This will prove to be useful also for this problem:

```
1 type Adjoints = Map[Num, Double]
```

A completely new obstacle is the tape itself which is also mutable. To convert it into something immutable let us first reflect which purpose it served. Principally it is a recursively accumulated list of operations which have to be executed later. Nothing speaks against passing a growing function (instead of a map for example) as an accumulator of our recursive pass through a calculation. In our case these operations always specifically acted with and on adjoints of some `Dual`. Alas, `Dual` has no `adjoint` member anymore (and therefore was renamed into `Num`). Its replacement is `Adjoints` which keeps track of adjoints of every `Num`. Consequently our "tape accumulator" has to act on `Adjoints`. More specifically it has to be of type `Adjoints => Adjoints` because you update the current (passed) adjoints by adding each child's partial adjoint and returning a new updated instance of `Adjoints`. Previously every `DualMonad` had a member `adjointsUpdater: Dual => Unit` which was prepended to the tape to ultimately collect every `adjointsUpdater`. We could instead use the member `adjointsUpdater` itself as an accumulator by changing its type. Previously `adjointsUpdater` just signified how to update the adjoints of one subexpression. Only on the tape they were chained. Now we directly chain each `adjointsUpdater` recursively on the fly and pass the chained `adjointsUpdater` to the outer `DualMonad`. Thus we are now skipping the tape entirely by moving it into an accumulator (line 3):

```
1 class DualMonad(
2     val parent: Num,
3     val adjointsUpdater: Adjoints => Adjoints
```

```
4 ):
5     def flatMap(k: Num => DualMonad): DualMonad =
6         val outerResult = k(parent)
7         DualMonad(
8             outerResult.parent,
9             outerResult.adjointsUpdater andThen this.adjointsUpdater
10        )
11    end flatMap
12
13    def map(k: Num => Num): DualMonad =
14        flatMap(k andThen wrap)
15 end DualMonad
16
17 def wrap(n: Num): DualMonad = DualMonad(n, identity)
```

`map` (line 13) follows the same logic as before in listing 3.5. In `flatMap` we first call the continuation to get the adjoints (and the normal result) of all outer expressions (line 6). `outerResult.adjointsUpdater` is then a tape-like construct which captures how to update all adjoints from the top expression up until the current expression. By appending `this.adjointsUpdater` in line 9 the operations are sorted from outer to inner which is the correct order we want to iterate through for the reverse pass of reverse mode differentiation.

Implementing `Num` is trivial and is very similar to previous implementations . We just have to remember to always use the updated adjoint map in line 25 and 26 because `addPartialAdjoint` returns a fresh object instead of mutating the old one:

```
1  class Num(val x: Double):
2      def *(that: Num): DualMonad =
3          val parent = Num(this.x * that.x)
4
5
6          def addPartialAdjoint(
7              thisOrThat: Num,
8              derivativeWrtThisOrThat: Double,
9              adjoints: Adjoints
10         ): Adjoints =
11             val partialAdjoint =
12                 adjoints(parent) * derivativeWrtThisOrThat
13             val newAdjointThisOrThat =
14                 adjoints(thisOrThat) + partialAdjoint
15             adjoints + (thisOrThat -> newAdjointThisOrThat)
16         end addPartialAdjoint
17
18
```

```
19          DualMonad(
20              parent,
21              adjoints =>
22                  val adjointsWithThis =
23                      addPartialAdjoint(this, that.x, adjoints)
24                  val adjointsWithThat =
25                      addPartialAdjoint(that, this.x, adjointsWithThis)
26                  adjointsWithThat
27          )
28      end *
29
30      def +(that: Num): DualMonad = ???
31  end Num
```

As always we have to implement `differentiate`:

```
1  def differentiate(f: DualMonad => DualMonad)(x: Double): Double =
2      val xM: DualMonad = wrap(Num(x))
3      val topMonad = f(xM)
4      val initialAdjoints =
5          Map.empty
6              .withDefaultValue(0.0)
7              .updated(topMonad.parent, 1.0)
8      topMonad.adjointsUpdater(initialAdjoints)(xM.parent)
9  end differentiate
10
11 def f(xM: DualMonad): DualMonad =
12     for
13         x <- xM
14         y1 <- x * 2
15         y2 <- x * x
16         y3 <- y2 * x
17         y4 <- y1 + y3
18     yield y4
19 end f
20
21 def g(xM: DualMonad): DualMonad =
22     for
23         x <- xM
24         y <- x * x
25     yield y
26 end g
27
28 val derivative = differentiate(f andThen g)(3)
```

At first we wrap the input `Double` into a `DualMonad` (line 1). When executing `f` in line 3

we do not get a meaningful result yet. We only have the monad of the top expression of the calculation. But this monad contains an `adjointsUpdater` which accumulated all operations needed to update an `Adjoints` map from scratch into our full result. We create our initial `Adjoints` map with default value 0, and we set the top expression to 1 (lines 4 to 7) just like for CPS. Finally, we can call `topMonad.adjointsUpdater` on our `initialAdjoints` and ultimately get the adjoint of `xM` (line 8)

As one can see we did not have to sacrifice much to reach an implementation without mutation. Writing and chaining functions (lines 11 to 28) are exactly the same to the previous monad implementation. The implementation itself is arguably better structured because we do not have to implement `DualMonad` ad hoc for every operation. Removing mutation and the global tape variable should also make the code easier to reason about. We also prevented at least one potential bug source because one had to remember to reset the tape for every calculation which is not a problem anymore. Furthermore, we could also argue that parallel execution using a global tape could lead to a multitude of problems which we wo not cover in more detail.

3.2.3 Combinatory Homomorphic Automatic Differentiation (CHAD)

Until now we relied on a map to accumulate all adjoints of all expressions by mapping `Num` to `Double` to get rid of mutation. From a functional programming standpoint this is still a compromise. We mapped specific instances of `Num` which means we have to generate a unique ID for every object to get a functioning map. In contrast imagine we would build our map by strictly comparing values (i.e. instances of `Num` with the same numeric value are treated as indistinguishable). Sub-expressions of our calculation with the same result value would share one adjoint which is clearly flawed. To have a purely functional implementation we have to get rid of instance comparison which means to drop the map. We also take one step back from monads and return to dual numbers:

```
1  case class Dual(v: Double, variableAdjoint: Double => Double):
2      def *(that: Dual): Dual =
3          def variableAdjointBothSides(
4              partialAdjointThis: Double,
5              partialAdjointThat: Double
6          ) =
7              this.variableAdjoint(partialAdjointThis)
8                  + that.variableAdjoint(partialAdjointThat)
9
10         def partialAdjointThis(parentAdjoint: Double) =
11             parentAdjoint * that.v
```

```
12
13          def partialAdjointThat(parentAdjoint: Double) =
14              parentAdjoint * this.v
15
16          Dual(
17              this.v * that.v,
18              parentAdjoint =>
19                  variableAdjointBothSides(
20                      partialAdjointThis(parentAdjoint),
21                      partialAdjointThat(parentAdjoint)
22                  )
23          )
24      end *
25
26      def +(that: Dual): Dual = ???
```

First thing to notice is that `Dual` now has a member named `variableAdjoint: Double => Double` (line 1). It calculates an adjoint using the parent adjoint like already familiar. But the important difference (and the reason for the prefix "variable-") is that it does *not* compute the partial adjoint of the current expression. It instead accumulates the sum of all partial adjoints of our variable (i.e. x). It ignores all partial adjoints of expressions not including x. We do not need them anyway.

Lines 10 to 14 look very familiar because it just computes the partial adjoints of `this` and `that`. From line 15 on we create a new `Dual` with a new `variableAdjoint`. We calculate both partial adjoints (lines 20 and 21) and then pass it to `variableAdjointBothSides`. `variableAdjointBothSides`'s job is to sum the partial adjoints of x from both branches but has to ensure that only relevant partial adjoints are included and for example constants are ignored. The function does in fact just sum both variable adjoints together (lines 7 and 8). So where do we ensure that only relevant partial adjoints are summed? The answer lies in how constants and the variable are instantiated/defined:

```
1 def variable(v: Double): Dual = Dual(v, identity)
2 def const(v: Double): Dual = Dual(v, _ => 0.0)
```

The partial adjoint of x is just directly its partial adjoint without further ado. This translates to the variable having `identity` as its variable adjoint function (line 1). The variable adjoint of constants are always zero (line 2). This makes sense because constants have no meaningful partial adjoint for our variable. Effectively when `variableAdjointBothSides` calls `this.variableAdjoint(partialAdjointThis)` and `this` is the variable x then it evaluates to `partialAdjointThis`. If `this` is a constant, the call evaluates to zero. If `this` is neither (e.g. a multiplication expression) then the previous two rules are implicitly applied

for all branches of that expression recursively. This, after some consideration, rather simple design is sufficient to implement reverse mode differentiation fully functionally and without building some kind of virtual stack/tape as we did previously.

Another advantage of this approach is its simplicity for the user:

```
1 given Conversion[Double, Dual] = const(_)
2 given Conversion[Int, Dual] = const(_)
3
4 def f(xDouble: Double): Dual =
5     val x: Dual = variable(xDouble)
6     2 * x + x * x * x
7 end f
8
9 val derivative = f(3).variableAdjoint(1)
```

After declaring the variable explicitly (line 5) (and using given conversions (lines 1 and 2) for constants) we can write `f` simply like a usual arithmetic expression in Scala (line 6). We do not even need a `differentiate` function because we accumulate the correct adjoint of x on the fly while calculating the result of `f`. In previous reverse mode implementations we called `adjoint` on x to get its adjoint which makes sense from a mathematical standpoint because we are interested in the adjoint of our variable and nothing else. In this case however we have to call `variableAdjoint` on the result of `f`. Ultimately we obviously still get the adjoint of x but the semantics of `variableAdjoint` are different (as the name suggests). It can be most easily interpreted as the sum of all partial adjoints of x contained in all branches of the current expression. Therefore, we must call it on the outermost expression to include all partial adjoints of x in our whole calculation. When doing this we have to pass 1 to `variableAdjoint` because it expects the adjoint of the current expression (i.e. the top expression) and similarly to previous implementations we know trivially that the adjoint of the top expression is 1.

4 Evaluation

Even though the main focus of this work were the implementations and their design we still want to do a brief runtime performance measurement of our implementations. For this we usually use the following definition for the test function `f`:

```
def f[A](x: A): A =
    0.1 * x * x * x + 2 * x * (x + 0.3 + x * 1) + 31 * x + x + -1 * x
```

Dependent on the tested reverse mode implementation the input and return type of `f` differs (e.g. `Monad`, `Dual`) and we use `A` as a placeholder. The implementation of `f` can also look different but the implemented mathematical function is always the same. For example think about CPS where we could not just write `f` as an one liner like above. `f` has multiplication, addition and uses `x` and constants multiple times and therefore is not trivial, but it is not really complex either. Clearly we have to know how our implementations perform on more complex functions which resemble deeply nested functions used in machine learning tasks. To achieve this we reuse the previous function and nest it into itself. Mathematically this is simple function composition $f \circ f \circ \cdots \circ f$. In Scala we use `andThen` to compose functions:

```
f andThen f andThen ??? andThen f
```

We want to compare how the nesting depth impacts runtime performance hence we need a way to dynamically produce a nested function with a specific depth `d`:

```
(0 until d) foldLeft(f) { (nestingAcc, _) =>
    nestedAcc andThen f
}
```

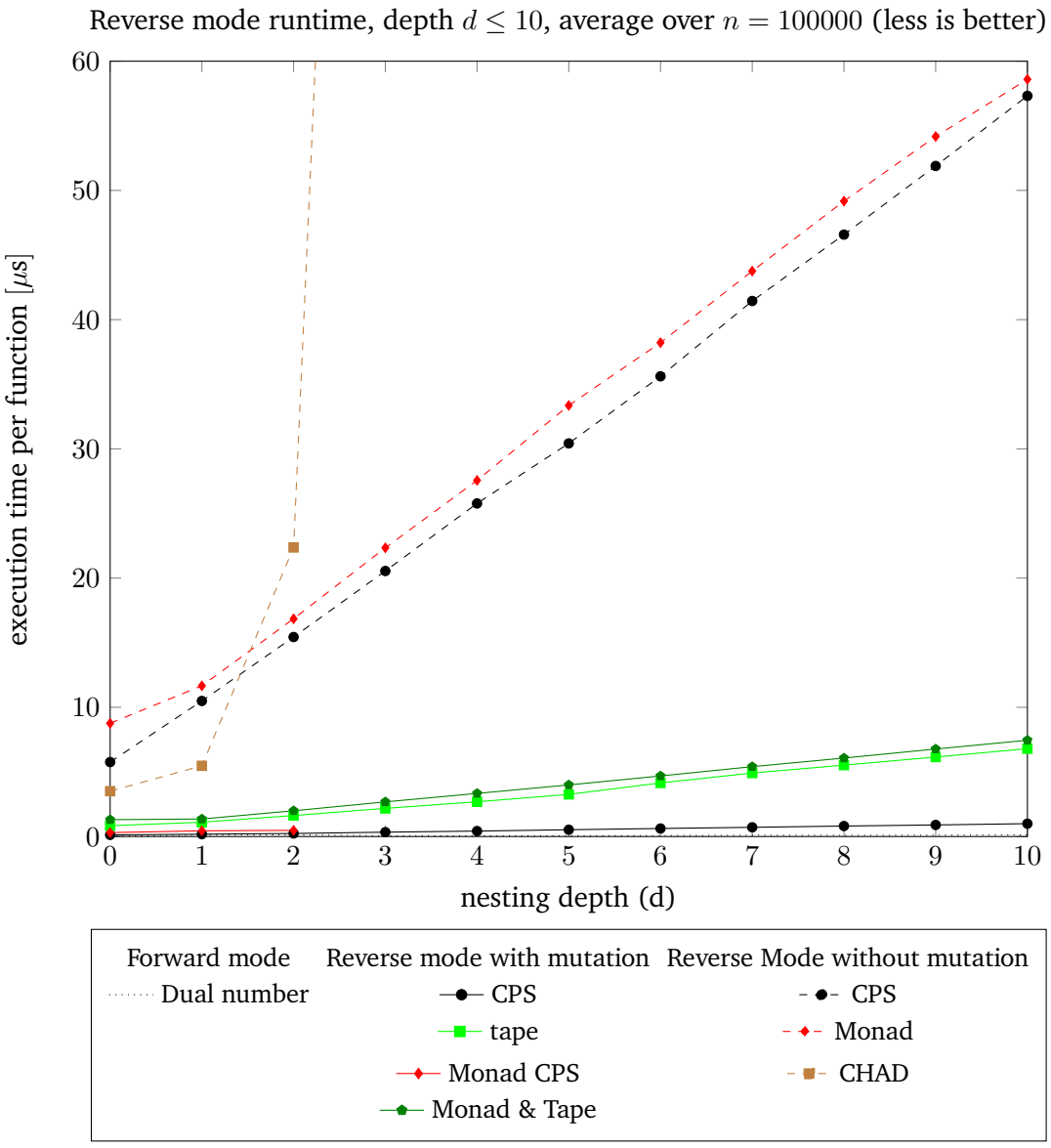
We create a range (or a list for that matter) with `d` items by using `(0 until d)`. We fold over it and use `f` as the base value. The lambda has two parameters. `nestingAcc` accumulates the nested `f`s. We ignore the second parameter because it is the value of

the integer range we called `foldLeft` on. The lambda essentially appends another `f` to the currently accumulated nested `f`s by using `andThen`. This is done `d` times because the original range has length `d`. If `d` is 0, its result is just `f`. If `d` is 1 then it produces `f andThen f` and so forth.

Each measurement is done `n = 100000` times and the average is taken to reduce measurement errors. The full measurement takes place like this:

- For each reverse mode implementation:
  - For each depth `d` (≥ 0):
    1. Produce the `d` times nested function
    2. Start timer
    3. Call the nested function `n` times
    4. Stop timer
    5. Divide measured time by `n` to get the time per function call

Let us look at the results for depth $d \leq 10$:



Focus on the legend to get a good overview over the many plots in that graph. It is divided into three columns. Forward mode has only one plot "Dual number" and is deliberately chosen to be rather unobtrusive. It is mainly included to serve as a kind of baseline to give the reverse mode measurements a context. Dual number is arguably the most straight forward and simple implementation of differentiation there is and because of this fills in

that role perfectly. More on forward mode implementations later.

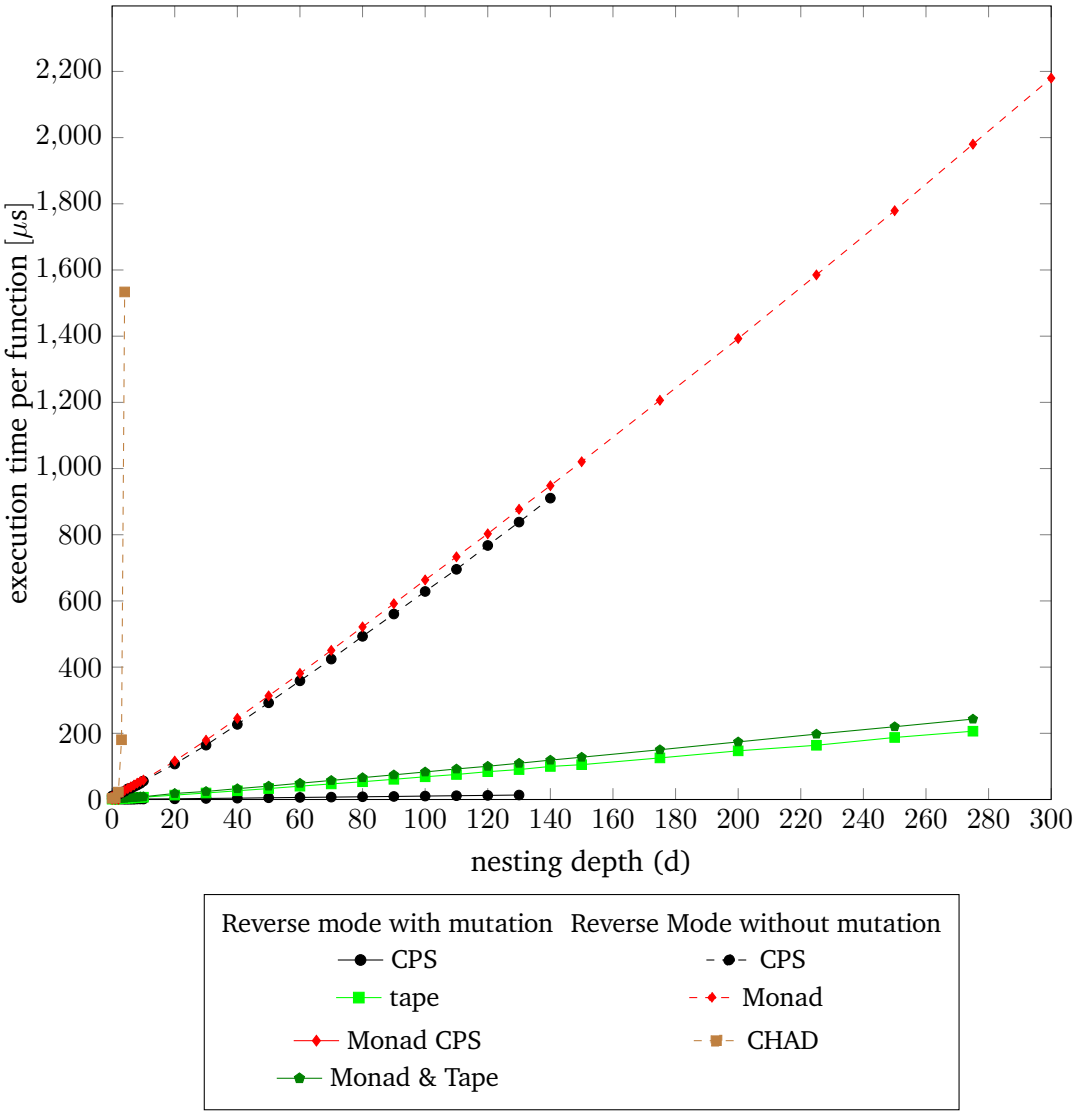
The second column of the legend includes our reverse mode implementations with mutation of section 3.1. These plots have solid lines. In contrast, all plots of the third column which represent implementations without mutation of section 3.2 have dashed plot lines. Note that the implementation of "Monad CPS" does not allow us to nest functions and therefore we had to manually write a function by hand for each nesting depth. As this is not feasible for deep nesting we could only measure function up until a nesting depth of 2.

The first observation we can quickly make is that implementations using mutation consistently perform better. This matches our intuitive prediction. We also notice that CPS without mutation and monad without mutation (white dashed and red dashed) perform similarly. This makes sense when we recall how similar the implementations of both approaches are. Both implementations essentially used a map as an accumulator for adjoints which is propagated through the whole calculation. Even though the resulting API (for-comprehensions vs. CPS) and the motivation for their implementations are different, in their cores they are alike which manifests as equal runtime performances. Another occurrence of this behaviour can be seen with tape and "Monad & Tape". The original motivation was to improve our original monad implementation by adding a tape. Turns out the finished monad can be interpreted more or less as a wrapper around the tape (instead of the other way around). The monad alters how we interact with our implementation from the outside (defining `f` with for-comprehensions) but on the inside it creates (almost) the same tape as our original tape implementation.

After discussing similarities we have to face the big disparity in our measurements. Every implementation seems to have a linear plot while CHAD is the only outlier. Note that linear increase is what we would expect and not exponential growth. Each time we chain our test function into itself the result normally only has to be calculated once and is reused whenever needed. Therefore, the most interesting insight out of this measurement would mostly be the gradient for each implementation in relation to the nesting depth but CHAD performs a magnitude worse for deep nesting. Note that for shallow/no nesting CHAD actually performs best among all implementations without mutation. So at first sight CHAD might have seemed like the silver bullet as it seemingly combined purity with comparatively great runtime performance but the main application of reverse mode differentiation is machine learning where we have to expect deeply nested structures for which CHAD evidently is not suitable. Opposed to our other implementations CHAD can not reuse the result of the nested function at multiple occurrences. Every occurrence of every variable has an implicit trace of the whole preceding calculation which has to be calculated. This is on the one hand the reason for its purity and elegance but on the other hand leads to this massive runtime performance hit.

If we zoom out from this graph, i.e. look at even deeper nesting depths, we can clearly see the linearity of each implementation except of CHAD:

Reverse mode runtime, depth $d \leq 300$, average over $n = 100000$ (less is better)



Moreover it can give us another interesting insight about our implementations. Each implementation has a soft limit at which it ca not run anymore. This could be an indicator

for how good the implementation supports very complex real life machine learning tasks. Usually the program crashes with a stack overflow or similar exceptions. We can see that CPS with and without mutation keep up similarly long and can handle a nesting depth of about 130 to 140. As explained earlier the end product of tape and combined monad with tape are similar which also shows in this metric. Probably because of its exponential growth CHAD only manages to handle a depth of 4. Outlier on the other extreme is monad without mutation. It can handle a depth of about 7000 which is far out of reach of this graph. The reason for this is that this is the only implementation where we do not build a (possibly implicit) stack of calculations which is run later (usually in reverse order). For the monad implementation we instead used a map which saves only the adjoint results and not the calculations themselves.

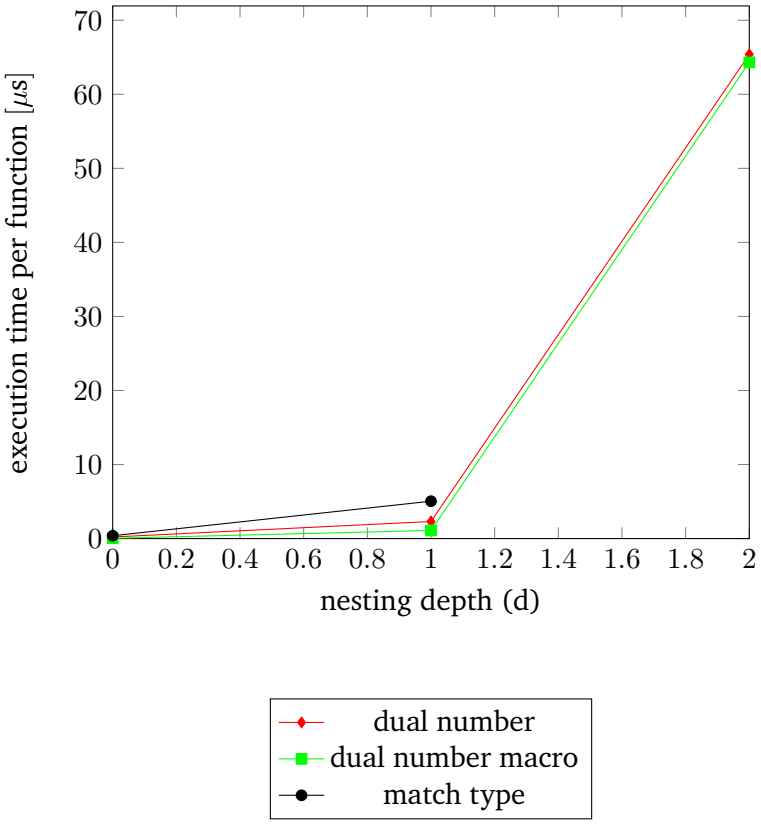
For our reverse mode implementations we can conclude that CHAD is not a good choice for real machine learning tasks which is particularly unfortunate because it was our purest implementation and had by far the best API among the implementations without mutation (no CPS nor for-comprehensions). While the slowest of the rest, Monad without mutation has the advantage of being extremely robust with respect to function complexity. This could theoretically prove useful for very complex real life tasks but will probably be seldom a deciding factor because a nesting depth of 130 is not small either. But still, CPS without mutation is at a somewhat bad spot. It does not have a nice API, is one of the slowest and has not the robustness of monad either. It still has the advantage of not using mutation and is on par performance wise with monad.

If purity is not of concern, the tape implementation could serve as the perfect sweet spot. It has the best API (even with CHAD), i.e. every expression is written like in vanilla Scala, has still a massively better performance than our implementations without mutation and supports very complex functions. Monad & tape performs similarly but you have to use for-comprehensions to define functions and is therefore not the best decision. Lastly CPS with mutation is the implementation of choice if runtime performance is of essence which is probably a major concern for machine learning tasks. Bad performing differentiations could have a massive impact on the total performance of a machine learning algorithm. Apart of CHAD it crashes at the lowest nesting depth but probably will not affect usual machine learning tasks. The major disadvantage is the API. Understanding the concept in itself, writing and presumably most importantly reading CPS is messy and could lead to bugs by misinterpreting or mistyping.

Even though reverse mode is most important we still want to have a quick look at our measurements of forward mode differentiation. Because dual number macro and match type are based on macros every expression must be available at compile time. Therefore we can not dynamically nest functions for this measurement. Because of technical limitations we also can not manually write functions which are nested deeply. Such a function would

be too long and causes “method too long” exceptions. Hence, we have to get along with a maximum nesting depth of 2 and only 1 for match type:

Forward mode runtime, depth $d \leq 2$, average over $n = 1000000$ (less is better)



We can see that dual number and dual number macro perform similarly which is no surprise because essentially the macro builds an object hierarchy at compile time which in turn executes the same operations the normal dual number implementation would do. Even though we technically have everything for our differentiation determined at compile time when using match types, we still have to do the whole calculation at runtime and unfortunately ca not perform better than the other two implementations. In fact the overhead even makes it a little slower.

5 Conclusion

In this work we have implemented forward and reverse mode differentiation in multiple ways. By starting with the mathematical foundations of reverse mode differentiation we have developed each implementation incrementally. By doing that we have seen that the core design of each implementation is similar but each design decision also has consequences. Implementations which are purer tend to perform worse which goes so far that we would say our purest implementation (CHAD) is not usable in real life tasks. In terms of API usability and functionality the result are mixed. There seems to be no real correlation between API and other metrics. But, at least under our implementations, the fastest one (CPS) has an unpleasant API and the slowest one (CHAD) has the purest implementation. We also learned that our implementations have different nesting limits which could be an indicator for how complex a function supported by our implementation can be.

Bibliography

[1] *Scala 3 Language Guide: Macros*. URL: https://docs.scala-lang.org/scala3/guides/macros/macros.html (visited on 03/08/2021).

[2] *Scala 3 Language Reference: Conversions*. URL: https://docs.scala-lang.org/scala3/reference/contextual/conversions.html (visited on 10/03/2021).

[3] *Scala 3 Language Reference: Givens*. URL: https://docs.scala-lang.org/scala3/reference/contextual/givens.html (visited on 10/03/2021).

[4] *Scala 3 Language Reference: Match Types*. URL: https://docs.scala-lang.org/scala3/reference/new-types/match-types.html (visited on 10/03/2021).

[5] Matthijs Vákár and Tom Smeding. "CHAD: Combinatory Homomorphic Automatic Differentiation". In: (2021).

[6] Fei Wang et al. "Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator". In: (2019).