Beautiful Differentiation
(extended version)

Conal M. Elliott
LambdaPix
conal@conal.net

Abstract
Automatic differentiation (AD) is a precise, efficient, and convenient method for computing derivatives of functions. Its forward-mode implementation can be quite simple even when extended to compute all of the higher-order derivatives as well. The higher-dimensional case has also been tackled, though with extra complexity. This paper develops an implementation of higher-dimensional, higher-order, forward-mode AD in the extremely general and elegant setting of calculus on manifolds and derives that implementation from a simple and precise specification.

In order to motivate and discover the implementation, the paper poses the question “What does AD mean, independently of implementation?” An answer arises in the form of naturality of sampling a function and its derivative. Automatic differentiation flows out of this naturality condition, together with the chain rule. Graduating from first-order to higher-order AD corresponds to sampling all derivatives instead of just one. Next, the setting is expanded to arbitrary vector spaces, in which derivative values are linear maps. The specification of AD adapts to this elegant and very general setting, which even simplifies the development.

Categories and Subject Descriptors G.1.4 [Mathematics of Computing]: Numerical Analysis—Quadrature and Numerical Differentiation

General Terms Algorithms, Design, Theory

Keywords Automatic differentiation, program derivation

1. Introduction
Derivatives are useful in a variety of application areas, including root-finding, optimization, curve and surface tessellation, and computation of surface normals for 3D rendering. Considering the usefulness of derivatives, it is worthwhile to find software methods that are

• simple to implement,
• simple to prove correct,
• convenient,
• accurate,
• efficient, and
• general.

One differentiation method is numeric approximation, using simple finite differences. This method is based on the definition of (scalar) derivative:

\[ dfx \equiv \lim_{h \to 0} \frac{f(x + h) - fx}{h} \]  

(1)

\[ d(f + g) \equiv df + dg \]
\[ d(fg) \equiv df \cdot g + f \cdot dg \]
\[ d(f \circ g) \equiv df \cdot g' \]
\[ d(1/f) \equiv -df / f^2 \]
\[ d(e^f) \equiv e^f \cdot df \]
\[ d(a^f) \equiv a^f \log a \cdot df \]
\[ d(\log f) \equiv df / f \]
\[ d(1/f) \equiv -df / f^2 \]
\[ d(x^n) \equiv n x^{n-1} \]
\[ d(\sin f) \equiv \cos f \cdot df \]
\[ d(\cos f) \equiv -\sin f \cdot df \]
\[ d(\tan f) \equiv 1 / \cos^2 f \cdot df \]
\[ d(\cot f) \equiv -1 / \sin^2 f \cdot df \]
\[ d(\sec f) \equiv \sec f \tan f \cdot df \]
\[ d(\csc f) \equiv -\csc f \cot f \cdot df \]
\[ d(\arcsin f) \equiv 1 / \sqrt{1 - f^2} \cdot df \]
\[ d(\arccos f) \equiv -1 / \sqrt{1 - f^2} \cdot df \]
\[ d(\arctan f) \equiv 1 / (1 + f^2) \cdot df \]

Figure 1. Some rules for symbolic differentiation

(The left-hand side reads “the derivative of f at x”.) The approximation method uses

\[ dfx \approx \frac{f(x + h) - fx}{h} \]

for a small value of h.

While very simple, this method is often inaccurate, due to choosing either too large or too small a value for h. (Small values of h lead to rounding errors.) More sophisticated variations improve accuracy while sacrificing simplicity.

A second method is symbolic differentiation. Instead of using the limit-based definition directly, the symbolic method uses a collection of rules, such as those in Figure 1.

There are two main drawbacks to the symbolic approach to differentiation. One is simply the inconvenience of symbolic methods, requiring access to and transformation of the source code of computation, and placing restrictions on that source code. A second drawback is that implementations tend to be quite expensive and in particular perform redundant computation.

A third method is the topic of this paper (and many others), namely automatic differentiation (also called “algorithmic differentiation”), or “AD” (Wengert 1964). There are forward and reverse variations (“modes”) of AD, as well as mixtures of the two. This paper considers only the forward-mode. The idea of AD is to simultaneously manipulate values and derivatives. Overloading of the standard numerical operations makes this combined manipulation as convenient and elegant as manipulating values without derivatives. Moreover, the implementation of AD can be quite simple as well. For instance, Figure 2 gives a simple, functional (forward-mode)
**Figure 2.** First-order, scalar, functional automatic differentiation

AD implementation, packaged as a data type \( D \) and a collection of numeric type class instances. Every operation acts on a regular value and a derivative value in tandem. (The derivatives for \( \cdot \) and \( \text{signum} \) need more care at 0.)

As an example, define

\[
\begin{align*}
f_1 &::= \text{Floating} \ a \Rightarrow a \to a \\
f_1 \ x &= \text{sqr} \ (3 \times \sin \ x)
\end{align*}
\]

and try it out in GHCi:

```
*Main> f1 2
1.6516332160855343 (-0.3779412091869595)
```

To test correctness, here is a symbolically differentiated version:

\[
\begin{align*}
f_2 &::= \text{Floating} \ a \Rightarrow a \to D \ a \\
f_2 \ x &= D \ (f_1 \ x) \ (3 \times \cos \ x / (2 \times \text{sqr} \ (3 \times \sin \ x)))
\end{align*}
\]

Try it out in GHCi:

```
*Main> f2 2
1.6516332160855343 (-0.3779412091869595)
```

This AD implementation satisfies most of our criteria very well:

- It is simple to implement. The code matches the familiar laws given in Figure 1. There are, however, some stylistic improvements to be made in Section 4.
- It is simple to verify informally, because of its similarity to the differentiation laws.
- It is convenient to use, as shown with \( f_1 \) above.
- It is accurate, as shown above, producing exactly the same result as the symbolic differentiated code, \( f_2 \).
- It is efficient, involving no iteration or redundant computation.
- The formulation in Figure 2 does less well with generality:
  - It computes only first derivatives.
  - It applies (correctly) only to functions over a scalar (one-dimensional) domain.

Moreover, proving correctness is hampered by lack of a precise specification. Later sections will address these shortcomings.

This paper’s technical contributions include the following.

- A prettier formulation of first-order and higher-order forward-mode AD using function-based overloading (Sections 2, 3 and 4).
- A simple formal specification for AD (Section 5).
- A systematic derivation of first-order and higher-order forward-mode AD from the specification (Sections 5.1 and 6).
- Reformulation of AD to general vector spaces including (but not limited to) \( \mathbb{R}^m \to \mathbb{R}^n \), from the perspective of *calculus on manifolds* (CoM) (Spivak 1971), and adaptation of the AD derivation to this new setting (Section 10).
- General and efficient formulations of linear maps and bases of vector spaces (using associated types and memo tries), since the notion of linear map is at the heart of CoM (Appendix A).

## 2. Friendly and precise

To start, let’s make some cosmetic improvements, which will be carried forward to the more general formulations as well. Figure 1 has an informality that is typical of working math notation, but we can state these properties more precisely. For now, give differentiation the following higher-order type:

\[
d :: (a \to a) \to (a \to a) \to \text{first attempt}
\]

Then Figure 1 can be made more precise. For instance, the sum rule is short-hand for

\[
d \ (\lambda x \to u + v) \equiv \lambda x \to d \ u + d \ v \ x
\]

and the log rule means

\[
d \ (\lambda x \to \log \ (u \ x)) \equiv \lambda x \to d \ u \ x / u \ x
\]

These more precise formulations are tedious to write and read. Fortunately, there is an alternative to replacing Figure 1 with more precise but less human-friendly forms. We can instead make the human-friendly form become machine-friendly. The trick is to add numeric overloading for *functions*, so that numeric operations apply point-wise. For instance,

\[
u + v \equiv \lambda x \to u \ x + v \ x
\]

\[
\log \ u \equiv \lambda x \to \log \ (u \ x)
\]

Then the “informal” laws in Figure 1 turn out to be well-defined and exactly equivalent to the “more precise” long-hand versions above. The *Functor* and *Applicative* (McBride and Paterson 2008) instances of functions (shown in Figure 3) come in quite handy.

Figure 4 shows the instances needed to make Figure 1 well-defined and correct exactly as stated, by exploiting the *Functor* and *Applicative* instances in Figure 3. In fact, these instances work for any applicative functor—a point that will become important in Section 10.

We’ll soon see how to exploit this simple, precise notation to improve the style of the definitions from Figure 2.
3. A scalar chain rule

Many of the laws in Figure 1 look similar: \( d (f \circ u) = d u \ast f' \begin{operator} u \end{operator} \) for some function \( f' \). The \( f' \) is not just some function; it is the derivative of \( f \). The reason for this pattern is that these laws follow from the scalar chain rule for derivatives.

\[
\begin{align*}
\frac{d}{dx} (g \circ f) &= \frac{dg}{df} \times \frac{df}{dx} \\
\end{align*}
\]

Using the \( (\ast) \) overloading in Figure 4, the chain rule can also be written as follows:

\[
\begin{align*}
d (g \circ f) &= (d g \circ f) \ast d f \\
\end{align*}
\]

All but the first two rules in Figure 1 then follow from the chain rule. For instance, \( d (\sin u) \equiv \{ \sin \text{ on functions} \} \) for some function \( \sin \).

The first two rules cannot be explained in terms of the scalar chain rule, but can be explained via the generalized chain rule in Section 10.

We can implement the scalar chain rule simply, via a new infix operator, \( \circ \), whose arguments are a function and its derivative.

\[
\begin{align*}
\text{infix} 0 \circ \\
\text{\( \circ \):} &\quad \text{Num } a \Rightarrow (a \rightarrow a) \rightarrow (a \rightarrow a) \rightarrow (D a \rightarrow D a) \\
&\quad (f \circ f') (D a \circ a') = D (f (a)) (a' \ast f') \\
\end{align*}
\]

This chain rule removes repetition from our instances. For instance,

\[
\begin{align*}
\text{instance} \quad \text{Floating } x \Rightarrow \text{Floating } (D x) \text{ where} \\
&\quad \pi \equiv D \pi 0 \\
&\quad \exp \equiv \exp \circ \exp \\
&\quad \log \equiv \log \circ \recip \\
&\quad \sqrt{\circ} = \sqrt{\circ} (x) \Rightarrow \sqrt{\circ} (2^{\ast} \sqrt{\circ} x) \\
&\quad \sin \equiv \sin \circ \cos \\
&\quad \cos \equiv \cos \circ \sin \\
&\quad \recip \Rightarrow \recip (\sqrt{\circ} (1 - \sqrt{\circ} x)) \\
&\quad \ast \Rightarrow \ast \\
\end{align*}
\]

4. Prettier derivatives via function overloading

Section 2 gave numeric overloading for functions in order to make the derivative laws in Figure 1 precise, while retaining their simplicity. We can use these overloading to make the derivative implementation simpler as well.

With the help of \( \circ \) and the overloading in Figure 4, the code in Figure 2 can be simplified to that in Figure 5.

5. What is automatic differentiation, really?

The preceding sections describe what AD is, informally, and they present plausible implementations. Let’s now take a deeper look at AD, in terms of three questions:

1. \textit{What} does it mean, independently of implementation?
2. \textit{How} do the implementation and its correctness flow gracefully from that meaning?
3. \textit{Where} else might we go, guided by answers to the first two questions?

5.1 A model for automatic differentiation

How do we know whether this AD implementation is correct? We can’t even begin to address this question until we first answer a more fundamental one: what exactly does its correctness mean? In other words, what specification must our implementation obey?

AD has something to do with calculating a function’s values and derivative values simultaneously, so let’s start there.

\[
\begin{align*}
\text{toD} :: (a \rightarrow a) \rightarrow (a \rightarrow D a) \\
\text{toD } f = \lambda x \rightarrow D (f x) (d f x) \\
\end{align*}
\]

Or, in point-free form,

\[
\begin{align*}
\text{toD } f = \text{liftA} \circ D d f \\
\end{align*}
\]

thanks to the \textit{Applicative} instance in Figure 3.

We have no implementation of \( d \), so this definition of \( \text{toD} \) will serve as a specification, not an implementation.

Since AD is structured as type class instances, one way to specify its semantics is by relating it to a parallel set of standard instances, by a principle of \textit{type class morphisms}, as described in (Elliot 2009b,a), which is to say that the interpretation preserves the structure of every method application. For AD, the interpretation...
instance Num a ⇒ Num (D a) where
  fromInteger = constD o fromInteger
  D x0 x' + D y0 y' = D (x0 + y0) (x' + y')
  x * D (D x0 x') + y * D (D y0 y') = D (x0 + y0) (x' + y + x*y')

 negate = negate ≡ λ v → v ≡ λ u → negate u
 signum = signum ≡ λ v → signum v

instance Fractional a ⇒ Fractional (D a) where
  fromRational = constD o fromRational
  recip = recip ≡ λ v → recip (2 * sqrt u)
  sin = sin ≡ λ v → cos
  cos = cos ≡ λ v → −sin
  asin = asin ≡ recip (sqrt (1 − sqrt u))
  acos = acos ≡ recip (−sqrt (1 − sqrt u))
  sinh = sinh ≡ cos
  cosh = cosh ≡ sinh
  asinh = asinh ≡ recip (sqrt (1 + sqrt u))
  acosh = acosh ≡ recip (−sqrt (sqrt u))
  atanh = atanh ≡ recip (1 − sqrt u)

instance Floating a ⇒ Floating (D a) where
  π = π ≡ λ v → cos
  exp = exp ≡ exp
  log = log ≡ recip
  sqrt = sqrt ≡ recip (2 * sqrt)
  sin = sin ≡ cos
  cos = cos ≡ −sin
  asin = asin ≡ recip (sqrt (1 − sqrt))
  acos = acos ≡ recip (−sqrt (1 − sqrt))
  sinh = sinh ≡ cos
  cosh = cosh ≡ sinh
  asinh = asinh ≡ recip (sqrt (1 + sqrt))
  acosh = acosh ≡ recip (−sqrt (sqrt u − 1))
  atanh = atanh ≡ recip (1 − sqrt u)

Figure 5. Simplified derivatives using the scalar chain rule and
function overloading.

function is toD. The Num, Fractional, and Floating morphisms
provide the specifications of the instances:

toD (u + v) = toD u + toD v

toD (u * v) = toD u * toD v

toD (negate u) = negate (toD u)

toD (sin u) = sin (toD u)

toD (cos u) = cos (toD u)

Note here that the numeric operations are applied to values of type
a → a on the left, and to values of type D a on the right.

These (morphism) properties exactly define correctness of any
implementation of AD, answering the first question:

What does it mean, independently of implementation?

5.2 Deriving an AD implementation

Equipped with a simple, formal specification of AD (numeric type
class morphisms), we can try to prove that the implementation
above satisfies the specification. Better yet, let’s do the reverse, using
the morphism properties to derive (discover) the implementation,
and prove it correct in the process. The derivations will then
provide a starting point for more ambitious forms of AD.

5.2.1 Constants and identity function

Value/derivative pairs for constant functions and the identity function
are specified as such:

\[ \text{constD} :: \text{Num} \rightarrow a \rightarrow D a \]
\[ \text{idD} :: \text{Num} \rightarrow a \rightarrow D a \]

5.2.2 Addition

Specify addition on D by requiring that toD preserves its structure:

\[ \text{toD} (u + v) \equiv \text{toD} u + \text{toD} v \]

Expand toD, and simplify both sides, starting on the left:

\[ \text{toD} (u + v) \equiv \{ \text{definition of toD} \} \]
\[ \text{toD} (\text{liftA}_2 D (u + v)) \equiv \{ \text{d (u + v)} \text{from Figure 1} \} \]
\[ \text{toD} (\text{liftA}_2 D (u + v)) \equiv \{ \text{functions from Figure 3} \} \]
\[ \lambda x \rightarrow D ((u + v) x) ((d u + d v) x) \equiv \{ (+) \text{from Figure 4} \} \]
\[ \lambda x \rightarrow D (u x + v x) (d u x + d v x) \]

Then start over with the right-hand-side:

\[ \text{toD} u + \text{toD} v \equiv \{ (+) \text{from Figures 3 and 4} \} \]
\[ \lambda x \rightarrow D u x + v x \equiv \{ \text{definition of toD} \} \]
\[ \lambda x \rightarrow D (u x) (d u x + d v x) \]

We need a (+) on D that makes these two final forms equal, i.e.,

\[ \lambda x \rightarrow D (u x + v x) (d u x + d v x) \equiv \lambda x \rightarrow D (u x) (d u x) + D (v x) (d v x) \]

An easy choice is

\[ D a a' + D b b' = D (a + b) (a' + b') \]

This definition provides the missing link, completing the proof that

\[ \text{toD} (u + v) \equiv \text{toD} u + \text{toD} v \]

5.2.3 Multiplication

The specification:

\[ \text{toD} (u * v) \equiv \text{toD} u * \text{toD} v \]

Reason similarly to the addition case. Begin with the left hand side:
Then start over with the right-hand side:

toD (u * v)
≡ { definition of toD }
liftA₂ D (u * v) (d (u * v))
≡ { d (u * v) from Figure 1 }
liftA₂ D (u * v) (d u * v + d v * u)
≡ { liftA₂ on functions from Figure 3 }
λx → D ((u * v) x) ((d u * v + d v * u) x)
≡ { liftA₂ on functions from Figure 4 }
λx → D (u x + v x) (d u x + v x x + d v x * u x)

Then start over with the right-hand side:

toD u * toD v
≡ { (+) on functions }
λx → toD u x * toD v x
≡ { definition of toD }
λx → D (u x) (d u x) * D (v x) (d v x)

Sufficient definition:
D a a' * D b b' = D (a * b) (a' * b + b' * a)

5.2.4 Sine

The specification:

\( \sin u \equiv \frac{\sin x}{x} \) for small \( x \)

The whole implementation can be derived in exactly this style, answering the second question:

How does the implementation and its correctness flow gracefully from that meaning?

6. Higher-order derivatives

Let’s now turn to the third question:

Where else might we go, guided by answers to the first two questions?

Our next destination will be higher-order derivatives, followed in Section 10 by derivatives over higher-dimensional domains.

Jerzy Karczmarczuk (2001) extended the \( D \) representation above to an infinite “lazy tower of derivatives”.

\[
\text{data } D a = D a \; (D a) \\
\text{The } toD \text{ function easily adapts to this new } D \text{ type: }
\]

\[
\text{toD } (a \to a) \to (a \to D a)
\]

\[
toD f x = D (f x) \; (toD \; (d f) \; x)
\]

or

\[
toD f = liftA_2 \; D \; f \; (toD \; (d f))
\]

The definition of \( toD \) comes from simplicity and type-correctness. Similarly, let’s adapt the previous derivations and see what arises.

6.1 Addition

Specification:

\[
toD (u + v) \equiv toD u + toD v
\]

Simplify the left-hand side:

\[
toD (u + v)
\]

\[
≡ { \text{definition of } toD }
\]

\[
liftA₂ D (u + v) (toD (d (u + v)))
\]

\[
≡ { d (u + v) }
\]

\[
liftA₂ D (u + v) (toD (d u + d v))
\]

\[
≡ { \text{induction for } toD \; (+) }
\]

\[
liftA₂ D (u + v) (toD (d u) + toD (d v))
\]

\[
≡ { \text{definition of } liftA₂ \text{ and } (+) \text{ on functions } }
\]

\[
λx → D (u x + v x) (toD (d u) x + toD (d v) x)
\]

and then the right:

\[
toD u + toD v
\]

\[
≡ { (+) on functions }
\]

\[
λx → toD u x + toD v x
\]

\[
≡ { \text{definition of } toD }
\]

\[
λx → D (u x) (toD (d u x) + D (v x) (toD (d v x))
\]

Again, we need a definition of \((+)\) on \( D \) that makes the LHS and RHS final forms equal, i.e.,

\[
λx → D (u x + v x) (toD (d u x) + toD (d v) x)
\]

≡

\[
λx → D (u x) (toD (d u x) + D (v x) (toD (d v x))
\]

Again, an easy choice is

\[
D a₀ a' + D b₀ b' = D (a₀ + b₀) (a' + b')
\]

The “induction” step above can be made more precise in terms of fixed-point introduction or the generic approximation lemma (Hutton and Gibbons 2001). Crucially, the morphism properties are assumed more deeply inside of the representation.

6.2 Multiplication

Simplifying on the left:

\[
toD (u * v)
\]

\[
≡ { \text{definition of } toD }
\]

\[
liftA₂ D (u * v) (toD (d (u * v)))
\]

\[
≡ { d (u * v) }
\]

\[
liftA₂ D (u * v) (toD (d u * v + d v * u))
\]

\[
≡ { \text{induction for } toD \; (+) }
\]

\[
liftA₂ D (u * v) (toD (d u * v) + toD (d v * u))
\]

\[
≡ { \text{induction for } toD \; (\times) }
\]

\[
liftA₂ D (u * v) (toD (d u) * toD v + toD (d v) * toD u)
\]

\[
≡ { liftA₂, \times, (+) on functions }
\]

\[
λx → liftA₂ D (u x * v x) (toD (d u) x * toD v x + toD (d v) x * toD u x)
\]
and then on the right:

\[ \text{toD } u \ast \text{toD } v \]
\[ = \{ \text{definition of toD } \} \]
\[ \text{liftA}_2 \ D \ u \ (\text{toD} (d \ u)) \ast \text{liftA}_2 \ D \ v \ (\text{toD} (d \ v)) \]
\[ = \{ \text{liftA}_2 \ and \ (\ast) \ on \ functions \} \]
\[ \lambda x \to D \ (u \ x) \ (\text{toD} (d \ u) \ x) \ast D \ (v \ x) \ (\text{toD} (d \ v) \ x) \]

A sufficient definition is

\[ a \circ (D \ a \ a') \ast b \circ (D \ b \ b') = D \ (a_0 + b_0) \ (a' \ast b + b' \ast a) \]

because

\[ \text{toD } u \ x \equiv D \ (u \ x) \ (\text{toD} (d \ u) \ x) \]
\[ \text{toD } v \ x \equiv D \ (v \ x) \ (\text{toD} (d \ v) \ x) \]

Note the new element here. The entire \( D \) value (tower) is used in building the derivative.

### 6.3 Sine

As usual \( \sin \) shows a common pattern that applies to other unary functions. Simplifying on the left-hand side:

\[ \text{toD} \ (\sin u) \]
\[ = \{ \text{definition of toD } \} \]
\[ \text{liftA}_2 \ D \ (\sin u) \ (\text{toD} (d \ (\sin u))) \]
\[ = \{ d \ (\sin u) \} \]
\[ \text{liftA}_2 \ D \ (\sin u) \ (\text{toD} (d \ u \ast \cos u)) \]
\[ = \{ \text{induction for toD} / (\ast) \} \]
\[ \text{liftA}_2 \ D \ (\sin u) \ (\text{toD} (d \ u) \ast \text{toD} \ (\cos u)) \]
\[ = \{ \text{induction for toD} / \cos \} \]
\[ \text{liftA}_2 \ D \ (\sin u) \ (\text{toD} (d \ u) \ast \cos \ (\text{toD} \ u \ x)) \]
\[ = \{ \text{liftA}_2 \ \sin, \ \cos \ and \ (\ast) \ on \ functions \} \]
\[ \lambda x \to D \ (\sin (u \ x)) \ (\text{toD} (d \ u) \ x \ast \cos \ (\text{toD} \ u \ x)) \]

and then the right:

\[ \sin \ (\text{toD} \ u) \]
\[ = \{ \text{definition of toD } \} \]
\[ \sin \ (\text{liftA}_2 \ D \ u \ (\text{toD} \ (d \ u))) \]
\[ = \{ \text{liftA}_2 \ and \ sin \ on \ functions \} \]
\[ \lambda x \to \sin \ (D \ (u \ x) \ (\text{toD} \ (d \ u) \ x)) \]

To make the left and right final forms equal, define

\[ \sin \ a \circ (D \ a \ a') \equiv D \ (\sin \ a_0) \ (a' \ast \cos \ a) \]

### 6.4 A higher-order, scalar chain rule

The derivation above for \( \sin \) shows the form of a chain rule for scalar derivative towers. It is very similar to the formulation in Section 3. The only difference are that the second argument (the derivative) gets applied to the whole tower instead of a regular value, and so has type \( D \ a \to D \ a \) instead of \( a \to a \).

```haskell
infix 0 ∞
(∞) :: (Num a) ⇒ (a → a) → (D a → D a) → (D a → D a)
(f ∞ f') a@(D a0 a') = D (f a0) (a' ∗ f' a)
```

With this new definition of \( ∞ \), all of the chain-rule-based definitions in Figure 5 (first-order derivatives) carry over without change to compute infinite derivative towers. For instance, \n
**instance** `Floating a ⇒ Floating (D a) where`

\[ \text{exp} = \exp \ ∞ \ \text{exp} \]
\[ \text{log} = \log \ ∞ \ \text{recip} \]
\[ \text{sqrt} = \sqrt{\_} \ ∞ \ \text{recip} \ (2 \ast \sqrt{\_}) \]
\[ \sin = \sin \ ∞ \ \text{cos} \]
\[ \cos = \cos \ ∞ \ -\sin \]

Now the operators and literals on the right of \( (\circ) \) are overloaded for the type \( D \ a \to D \ a \). For instance, in the definition of \( \text{sqrt} \),

\[ 2 :: D a \to D a \]
\[ \text{recip} :: (D a \to D a) \to (D a \to D a) \]
\[ (+) :: (D a \to D a) \to (D a \to D a) \to (D a \to D a) \]
\[ → (D a \to D a) \]

### 7. Optimizing zeros

The derivative implementations above are simple and powerful, but have an efficiency problem. For polynomial functions (constant, linear, quadratic, etc), all but a few derivatives are zero. Considerable wasted effort can go into multiplying and adding zero derivatives.

Consider, for instance, multiplying a constant by a \( \sin \) function.

Let’s look first at the situation with first-order derivatives:

\[ 2 \ast \sin \ (idD \ x) \]
\[ = D \ 2 \ast \sin \ (D \ x \ 1) \]
\[ = D \ 2 \ast D \ (\sin x) \ (1 \ast \cos x) \]
\[ = D \ (2 \ast \sin x) \ (0 \ast \sin x + (1 \ast \cos x) \ast 2) \]

In this case there is a wasted \( \sin \ x \), multiplication by zero, and addition to zero. The situation is much worse with higher derivatives:

\[ 2 \ast \sin \ (idD \ x) \]
\[ = D \ 2 \ast \sin \ (D \ x \ 1) \]
\[ = D \ 2 \ast D \ (\sin x) \ (1 \ast \cos (idD \ x)) \]
\[ = D \ (2 \ast \sin x) \ (0 \ast \sin (idD \ x) + (1 \ast \cos (idD \ x)) \ast 2) \]

In the derivative part of this last expression, every factor is an infinite derivative towers: \( 0, 1, 2, \sin \ (idD \ x) \) and \( \cos \ (idD \ x) \).

Jerzy Karczmarczuk (2001) and others suggest using two constructors, one for constants and one for non-constants. In other words, one for zero derivative and one for non-zero derivative. Instead, let’s have only one \( D \) constructor but have the derivative part be optional.

For simplicity, consider the first-order case. Replace

**data** `D a = D a a` -- first-order, non-optimized

with

**data** `D a = D a (Maybe a)` -- first-order, optimized

The constant and identity cases change accordingly:

\[ \text{constD} :: \text{Num a} ⇒ a → D a \]
\[ \text{constD} x = D x \ Nothing \]
\[ idD :: \text{Num a} ⇒ a → D a \]
\[ idD x = D x \ (Just 1) \]

In order to keep the numeric instances simple, define versions of addition and multiplication on these optional derivatives.

**infixl 6 +**

\( (+) :: \text{Num a} ⇒ \text{ Maybe a} → \text{Maybe a} → \text{Maybe a} \)
\[ \text{Nothing} + b' = b' \]
\[ a' + \text{Nothing} = a' \]
\[ \text{Just} \ a' + \text{Just} \ b' = \text{Just} \ (a' + b') \]

**infixl 7 ∗**

\( (∗) :: \text{Num a} ⇒ \text{Maybe a} → a → \text{Maybe a} \)
\[ \text{Nothing} ∗ _ = \text{Nothing} \]
\[ Just \ a' ∗ \_ = Just \ (a' ∗ \_ b) \]

The \( (+) \) definition mirrors the \textit{Monoid} instance of \textit{Maybe (Sum a)}; while the \( (∗) \) definition is a \textit{fmap} using the \textit{Functor} instance of \textit{Maybe}. 

Beautiful differentiation
The changes required for AD are minimal: simply replace addition of derivatives and multiplication by derivatives with \((\ast)\) and \((\ast+\)\) in the chain rule

\[
\text{infix } 0 \ast
\]

\[
(\infix) : (\text{Num } a) \Rightarrow (a \rightarrow a) ightarrow (a \rightarrow a) \rightarrow (D a \rightarrow D a)
\]

\[
(f \infix f') (D a a') = D (f a) (a' * f' a)
\]

and in the derivative of sums and products

\[
D a a' + D b b' = D (a + b) (a' + b')
\]

\[
D a a' * D b b' = D (a * b) (a' * b + b' * a)
\]

Optimizing higher-order differentiation is just as easy, using these same replacements. The new representation:

\[
data D a = D a (\text{Maybe } (D a)) \quad -- \text{higher-order, optimized}
\]

8. What is a derivative, really?

Section 6 showed how easily and beautifully one can construct an infinite tower of derivative values in Haskell programs, while computing plain old values. The trick (from Karczmarek 2001) was to overload numeric operators to operate on the following (co)recursive type:

\[
data D b = D b (D b)
\]

This representation, however, works only when differentiating functions from a one-dimensional domain. The reason for this limitation is that only in those cases can the type of derivative values be identified with the type of regular values.

Consider a function \(f : \mathbb{R}^2 \rightarrow \mathbb{R}\). The value of \(f\) at a domain value \((x, y)\) has type \(\mathbb{R}\), but the derivative of \(f\) consists of two partial derivatives. Moreover, the second derivative consists of four partial second-order derivatives (or three, depending how you count). A function \(f : \mathbb{R}^2 \rightarrow \mathbb{R}^2\) also has two partial derivatives at each point \((x, y)\), each of which is a triple. That pair of triples is commonly written as a three-by-two matrix.

Each of these situations has its own derivative shape and its own chain rule (for the derivative of function compositions), using plain-old multiplication, scalar-times-vector, vector-dot-vector, matrix-times-vector, or matrix-times-matrix. Second derivatives are more complex and varied.

How many forms of derivatives and chain rules are enough? Are we doomed to work with a plethora of increasingly complex types of derivatives, as well as the diverse chain rules needed to accommodate all compatible pairs of derivatives? Fortunately, not. There is a single, simple, unifying generalization. By reconsidering what we mean by a derivative value, we can see that these various forms are all representations of a single notion, and all the chain rules mean the same thing on the meanings of the representations.

Let’s now look at unifying view of derivatives, which is taken from *calculus on manifolds* (Spivak 1971). To get an intuitive sense of what’s going on with derivatives in general, we’ll look at some examples.

8.1 One dimension

Start with a simple function on real numbers:

\[
f_1 : \mathbb{R} \rightarrow \mathbb{R}
\]

\[
f_1 x = x^2 + 3 * x + 1
\]

Writing the derivative of a function \(f\) as \(d f\), let’s now consider the question: what is \(d f_1\)? We might say that

\[
d f_1 x = 2 * x + 3
\]

so e.g., \(d f_1 5 = 13\). In other words, \(f_1\) is changing 13 times as fast as its argument, when its argument is passing 5.

Rephrased yet again, if \(dx\) is a very tiny number, then \(f_1 (5 + dx) - f_1 5\) is very nearly \(13 * dx\). If \(f_1\) maps seconds to meters, then \(d f_1\) is 13 meters per second. So already, we can see that the range of \(f\) (meters) and the range of \(d f\) (meters/second) disagree.

8.2 Two dimensions in and one dimension out

As a second example, consider a two-dimensional domain:

\[
f_2 : \mathbb{R}^2 \rightarrow \mathbb{R}
\]

\[
f_2 (x, y) = 2 * x * y + 3 * x + 5 * y + 7
\]

Again, let’s consider some units, to get a guess of what kind of thing \(d f_2 (x, y)\) really is. Suppose that \(f_2\) measures altitude of terrain above a plane, as a function of the position in the plane. (So \(f_2\) is a height field.) You can guess that \(d f (x, y)\) is going to have something to do with how fast the altitude is changing, i.e. the slope, at \((x, y)\). But there is no single slope. Instead, there’s a slope for every possible compass direction (a hiker’s degrees of freedom).

Now consider the conventional answer to what is \(d f_2 (x, y)\). Since the domain of \(f_2\) is \(\mathbb{R}^2\), it has two partial derivatives:

\[
d f_2 (x, y) = (2 * y + 3, 2 * x + 5)
\]

In our example, these two pieces of information correspond to two of the possible slopes. The first is the slope if heading directly east, and the second if directly north (increasing \(x\) and increasing \(y\), respectively).

What good does it do our hiker to be told just two of the infinitude of possible slopes at a point? The answer is perhaps magical: for well-behaved terrains, these two pieces of information suffice to calculate all (infinitely many) slopes, with just a bit of math. Every direction can be described as partly east and partly north (negatively for westish and southish directions). Given a direction angle \(\vartheta\) (where east is zero and north is 90 degrees), the east and north components are \(\cos \vartheta\) and \(\sin \vartheta\), respectively. When heading in the direction \(\vartheta\), the slope will be a weighted sum of the north-going slope and the east-going slope, where the weights are these north and south components.

Instead of angles, our hiker may prefer thinking directly about the north and east components of a tiny step from the position \((x, y)\). If the step is small enough and lands \(dx\) to the east and \(dy\) to the north, then the change in altitude, \(f_2 (x + dx, y + dy) - f_2 (x, y)\) is very nearly equal to \((2 * y + 3) * dx + (2 * x + 5) * dy\). If we use \((\infix)\) to mean dot (inner) product, then this change in altitude is \(d f_2 (x, y) \infix (dx, dy)\).

From this second example, we can see that the derivative value is not a range value, but also not a rate-of-change of range values. It’s a pair of such rates plus the know-how to use those rates to determine output changes.

8.3 Two dimensions in and three dimensions out

Next, imagine moving around on a surface in space, say a torus, and suppose that the surface has grid marks to define a two-dimensional parameter space. As our hiker travels around in the 2D parameter space, his position in 3D space changes accordingly, more flexibly than just an altitude. The hiker’s type is then

\[
f_3 : \mathbb{R}^2 \rightarrow \mathbb{R}^3
\]

At any position \((s, t)\) in the parameter space, and for every choice of direction through parameter space, each of the coordinates of the position in 3D space has a rate of change. Again, if the function is mathematically well-behaved (differentiable), then all of these rates of change can be summarized in two partial derivatives. This time, however, each partial derivative has components in \(X, Y,\) and \(Z\), so it takes six numbers to describe the 3D velocities for all possible directions in parameter space. These numbers are usually written as a 3-by-2 matrix \(m\) (the Jacobian of \(f_3\)). Given a small
8

Beautiful differentiation

parameter step \((dx, dy)\), the resulting change in 3D position is equal to the product of the derivative matrix and the difference vector, i.e., \(m \cdot \text{timesVec}' (dx, dy)\).

### 8.4 A unifying perspective

The examples above use different representations for derivatives: scalar numbers, a vector (pair of numbers), and a matrix. Common to all of these representations is the ability to perform a small step in the function’s domain into a resulting step in the range.

- In \(f_1\), the (scalar) derivative \(c\) means \((c\cdot)\), i.e., multiply by \(c\).
- In \(f_2\), the (vector) derivative \(v\) means \((v \cdot \text{vec})\).
- In \(f_3\), the (matrix) derivative \(m\) means \((m \cdot \text{timesVec}')\).

So, the common meaning of these derivative representations is a function, and not just any function, but a linear function—often called a “linear map” or “linear transformation”.

Now, what about the different chain rules, saying to combine derivative values via various kinds of products (scalar/scalar, scalar/vector, vector/vector dot, matrix/vector)? Each of these products implements the same abstract notion, which is composition of linear maps.

### 9. The general setting: vector spaces

Linear maps (transformations) lie at the heart of generalized differentiation. Talking about linearity requires a few simple operations, which are encapsulated in the the abstract interface known from math as a vector space.

Vector spaces specialize the more general notion of a group which as an associative and commutative binary operator, an identity, and inverses. For convenience, we’ll specialize to an additive group which provides addition-friendly names:

```huge
class AdditiveGroup v where
  0 :: v
  (+) :: v \to v \to v
  negate :: v \to v
```

Next, given a field \(s\), a vector space over \(s\) adds a scaling operation:

```huge
class AdditiveGroup v \Rightarrow Vector s v where
  (\cdot) :: s \to v \to v
```

In many cases, we’ll want to add inner (dot) products as well, to form an inner product space

```huge
class Vector s v \Rightarrow InnerSpace s v where
  (\cdot\cdot) :: v \to v \to s
```

Instances include Float, Double, and Complex, as well as tuples of vectors, and functions with vector ranges. (By “vector” here, I mean any instance of Vector, recursively.) For instance, here are instances for functions:

```huge
instance AdditiveGroup v \Rightarrow AdditiveGroup (a \to v) where
  0 = pure 0
  (+) = liftA2 (+)
  negate = fmap negate
```

```huge
instance Vector s v \Rightarrow Vector s (a \to v) where
  (\cdot) s = fmap (s\cdot)
```

These method definitions have a form that can be used with any applicative functor.

Other useful operations can be defined in terms of these methods, e.g., subtraction for additive groups, and linear interpolation for vector spaces.

Several familiar types are vector spaces:

- Trivially, the unit type is an additive group and is a vector space over every field.
- Scalar types are vector spaces over themselves, with \(t\equiv(\cdot)\).
- Tuples add and scale component-wise.
- Functions add and scale point-wise, i.e., on their range.

Appendix A gives an efficient representation of linear maps via an associated type (Chakravarty et al. 2005) of bases of vector spaces. Without regard to efficiency, we could instead represent linear maps as a simple wrapper around functions, with the invariant that the contained function is indeed linear:

```huge
newtype u \sim v = LMap (u \to v) -- simple & inefficient deriving (AdditiveGroup, Vector)
```

Assume the following abstract interface, where linear and lapply convert between linear functions and linear maps, and \(id_L\) and \((\cdot)\) are identity and composition.

```huge
linear :: (Vector s u, Vector s v) \Rightarrow (u \to v) \to (u \sim v)
lapply :: (Vector s u, Vector s v) \Rightarrow (u \sim v) \to (u \to v)
```

```huge
id_L :: (Vector s u) \Rightarrow u \sim u
(\cdot) :: (Vector s u, Vector s v) \Rightarrow (v \sim u) \to (u \sim v)
```

Another operation plays the role of dot products, as used in combining partial derivatives.

```huge
(\cdot\cdot) :: (Vector s u, Vector s v, Vector s w) \Rightarrow (v \sim w) \to ((u, v) \sim w)
```

Semantically,

\[(s', t') \cdot\cdot (da, db) \equiv s' \cdot da + t' \cdot db\]

which is linear in \((da, db)\). Compare with the usual definition of dot products:

\[(s', t') \cdot\cdot (da, db) \equiv s' \cdot da + t' \cdot db\]

Dually to \((\cdot)\), another way to form linear maps is by “zipping”:

```huge
(\cdot) :: (w \sim u) \to (w \sim v) \to (w \sim (u, v))
```

which will reappear in generalized form in Section 10.3.

### 10. Generalized derivatives

We’ve seen what AD means and how and why it works for a specialized case of the derivative of \(a \to a\) for a one-dimensional (scalar) type \(a\). Now we’re ready to tackle the specification and derivation of AD in the much broader setting of vector spaces.

Generalized differentiation introduces linear maps:

```huge
d :: (Vector s u, Vector s v) \Rightarrow (a \to v) \to (u \sim (a \sim v))
```

In this setting, there is a single, universal chain rule (Spivak 1971):

\[d (g \circ f) x \equiv d g (f x) \odot d f x\]

where \((\circ)\) is composition of linear maps. More succinctly,

\[d (g \circ f) \equiv (d g \circ f) \odot d f\]

using lifted composition:

\[(\cdot\cdot) \equiv \text{liftA}2\ (\cdot)\]

#### 10.1 First-order generalized derivatives

The new type of value/derivative pairs has two type parameters:

```huge
data a \sim b = D b (a \sim b)
```
As in Section 5.1, the AD specification centers on a function, $toD$, that samples a function and its derivative at a point. This time, it will be easier to swap the parameters of $toD$:

$$toD :: (\text{Vector } s \ u, \text{Vector } s \ v) \Rightarrow u \to (u \to v) \to u \triangleright v$$

$$toD_x f = D (f \ x) (d \ f \ x)$$

In Sections 5 and 6, AD algorithms were derived by saying that $toD$ is a morphism over numeric types. The definitions of these morphisms and their proofs involved one property for each method. In the generalized setting, we can instead specify and prove three simple morphisms, from which all of the others follow effortlessly.

We already saw in Figure 4 that the numeric methods for functions have a simple, systematic form. They’re all defined using $\text{fmap}$, pure, or $\text{liftA}_2$ in a simple, regular pattern, e.g.,

$$\text{fromInteger} = \text{pure} \circ \text{fromInteger}$$

$$(*) = \text{liftA}_2 (*)$$

$${\sin} = \text{fmap} \sin$$

$$\ldots$$

Numeric instances for many other applicative functors can be given exactly the same method definitions. For instance, $\text{Maybe} \ a, \text{Either} \ a \ b, \text{a} \to \text{b}, \text{tries}$, and syntactic expressions (Elliott 2009a).

Could these same definitions work on $a \triangleright \cdot$, as an implementation of AD?

Consider one example:

$$\sin = \text{fmap} \sin$$

For now, assume this definition and look at the corresponding numeric morphism property, i.e.,

$$toD_x (\sin u) = \sin (toD_x u)$$

Expand the definitions of $\sin$ on each side, remembering that the left $\sin$ is on functions, as given in Figure 4.

$$toD_x (\text{fmap} \sin u) = \text{fmap} \sin (toD_x u)$$

which is a special case of the functor morphism property for $toD_x$. Therefore, proving the functor morphism property will cover all of the definitions that use $\text{fmap}$.

The other two definition styles (using pure and $\text{liftA}_2$) work out similarly. For example, if $toD_x$ is an applicative morphism, then

$$toD_x (\text{fromInteger} n) \equiv \{ \text{fromInteger for } a \triangleright b \}$$

$$toD_x (\text{pure} (\text{fromInteger} n)) \equiv \{ toD_x \text{ is an applicative morphism } \}$$

pure \equiv \{ \text{fromInteger for functions } \}

fromInteger n

$$toD_x (u \triangleright v) \equiv \{ (\ast) \text{ for } a \triangleright b \}$$

$$toD_x (\text{liftA}_2 (\ast) \ u \ v) \equiv \{ toD_x \text{ is an applicative morphism } \}$$

$$\text{liftA}_2 (\ast) \ (toD_x u) \ (toD_x v) \equiv \{ (\ast) \text{ on functions } \}$$

$$toD_x u \triangleright toD_x v$$

Now we can see why these definitions succeed so often: For applicative functors $F$ and $G$, and function $\mu :: F \ a \to G \ a$, if $\mu$ is a morphism on $\text{Functor}$ and $\text{Applicative}$, and the numeric instances for both $F$ and $G$ are defined as in Figure 4, then $\mu$ is a numeric morphism.

Thus, we have only to come up with $\text{Functor}$ and $\text{Applicative}$ instances for $\triangleright \cdot$ $u$ such that $toD_x$ is a $\text{Functor}$ and $\text{Applicative}$ morphism.

### 10.2 Functor

First look at $\text{Functor}$. The morphism condition (naturality), $\eta$-expanded, is

$$\text{fmap} \ g \ ((toD_a f) \equiv toD_x \ (\text{fmap} \ g \ f))$$

Using the definition of $toD$ on the left gives

$$\text{fmap} \ g \ ((D \ (f \ x) \ (d \ f \ x)))$$

Simplifying the RHS,

$$\begin{align*}
&toD_x \ (\text{fmap} \ g \ f) \\
\equiv &\{ \text{definition of } toD \} \\
&\text{D} ((\text{fmap} \ g \ f) \ x) (d \ (\text{fmap} \ g \ f) \ x) \\
\equiv &\{ \text{definition of } \text{fmap} \text{ for functions} \} \\
&\text{D} ((g \triangleright f) \ x) (d \ (g \triangleright f) \ x) \\
\equiv &\{ \text{generalized chain rule} \} \\
&D \ (g \ (f \ x)) (d \ g \ (f \ x) \circ d \ f \ x)
\end{align*}$$

So the morphism condition is equivalent to

$$\text{fmap} \ g \ ((D \ (f \ x) \ (d \ f \ x))) \equiv D \ (g \ (f \ x)) (d \ g \ (f \ x) \circ d \ f \ x)$$

Now it’s easy to find a sufficient definition:

$$\text{fmap} \ g \ (D \ (fx \ dfx)) = D \ (g \ fx) (d \ g \ fx \circ dfx)$$

This definition is not executable, however, since $d$ is not. Fortunately, all uses of $\text{fmap}$ in the numeric instances involve functions $g$ whose derivatives are known statically and so can be statically substituted for applications of $d$. To make the static substitution more apparent, refactor the $\text{fmap}$ definition, as in Section 3.

\[\begin{align*}
\text{instance } \text{Functor} \ ((\triangleright\triangleright g)) \text{ where} &\text{ fmap} \ g = g \triangleright\triangleright d \ g \\
(\triangleright\triangleright) :: \text{Vector } s \ u, \text{Vector } s \ v, \text{Vector } s \ w \Rightarrow & (v \to w) \to (v \to (v \to w)) \to (u \triangleright v) \to (u \triangleright w) \\
& (g \triangleright\triangleright dg) (D \ fx \ dfx) = D \ (g \ fx) (dg \ fx \circ dfx)
\end{align*}\]

This new definition makes it easy to transform the $\text{fmap}$-based definitions systematically into effective versions. After inlining this definition of $\text{fmap}$, the $\text{fmap}$-based definitions look like

\[\begin{align*}
&\sin = \sin \triangleright\triangleright d \ sin \\
&\text{sqrt} = \text{sqrt} \triangleright\triangleright \text{recip} \ (2 * \text{sqrt}) \\
&\ldots
\end{align*}\]

Every remaining use of $d$ is applied to a function whose derivative is known, so we can replace each use.

\[\begin{align*}
&\sin = \sin \triangleright\triangleright \cos \\
&\text{sqrt} = \text{sqrt} \triangleright\triangleright \text{recip} \ (2 * \text{sqrt}) \\
&\ldots
\end{align*}\]

Now we have an executable implementation again.

### 10.3 Applicative/Monoidal

$\text{Functor}$ is handled, which leaves just $\text{Applicative}$ (McBride and Paterson 2008):

\[\begin{align*}
\text{class } \text{Functor} \ f \Rightarrow \text{Applicative} \ f \text{ where} &\text{ pure :: } a \to f \ a \\
(\otimes) :: f \ (a \to b) \to f \ a \to f \ b
\end{align*}\]

The morphism properties will be easier to demonstrate in terms of a type class for (strong) lax monoidal functors:

\[\begin{align*}
\text{class } \text{Monoidal} \ f \text{ where} &\text{ unit :: } f () \\
(\ast) :: f \ a \to f \ b \to f \ (a, b)
\end{align*}\]

For instance, the function instance is
The Applicative class is equivalent to \textit{Functor+Monoidal} (McBride and Paterson 2008, Section 7). To get from \textit{Functor} and \textit{Monoidal} to Applicative, define

\[
pure \ a = \text{fmap} (\text{const} \ a) \ \text{unit} \\
fs \otimes \xs = \text{fmap} \ \text{app} \ (fs \ast xs)
\]

where
\[
\text{app} :: (a \to b, a) \to b
\]

The reason for switching from the \textit{Application} is very simple with the latter:

\[
(f \ x) \equiv \text{fmap} \ f \ x
\]

I’ve kept \textit{Monoidal} independent of \textit{Functor}, unlike (McBride and Paterson 2008), because the linear map type has \textit{unit} and \textit{(\ast)} but is not a functor. (Only \textit{linear} functions can be mapped over a linear map.)

The shift from \textit{Application} to \textit{Monoidal} makes the specification of \textit{toD} simpler, again as a morphism:

\[
\begin{align*}
\text{unit} & \equiv \text{toD}_x \ \text{unit} \\
\text{toD}_x f \ast \text{toD}_x g & \equiv \text{toD}_x (f \ast g)
\end{align*}
\]

Filling in the definition of \textit{toD},

\[
\begin{align*}
\text{unit} & \equiv \text{D} \ (\text{unit} \ x) \\
\text{D} \ (f \ x) \ (d \ \text{unit} \ x) & \equiv \text{D} \ ((f \ast g) \ x) \ (d \ (f \ast g) \ x)
\end{align*}
\]

The reason for switching from \textit{Application} to \textit{Monoidal} is that differentiation is very simple with the latter:

\[
\begin{align*}
d \text{unit} & \equiv \text{id} \\
d (f \ast g) & \equiv d f \ast d g
\end{align*}
\]

The \((\ast)\) on the right is a variant of \((\ast)\), lifted to work on functions (or other applicative functors) that yield linear maps:

\[
(\ast) = \text{liftA}_2 \ (\ast)
\]

We cannot simply pair linear maps to get a linear map. Instead, \((\ast)\) pairs linear maps point-wise.

Now simplify the morphism properties, using \textit{unit} and \textit{(\ast)} for functions, and their derivatives:

\[
\begin{align*}
\text{unit} & \equiv \text{D} \ (0) \\
\text{D} \ (f \ x) \ (d \ f \ x) \ast \text{D} \ (g \ x) \ (d \ g \ x) & \equiv \text{D} \ (f \ x, g \ x) \ (d \ f \ x \ast d g \ x)
\end{align*}
\]

So the following simple definitions suffice:

\[
\begin{align*}
\text{unit} & \equiv \text{D} \ (0) \\

\text{D} \ fx \ dfx \ast \text{D} \ gx \ dgx & \equiv \text{D} \ (fx, gx) \ (dfx \ast dgx)
\end{align*}
\]

The switch from \textit{Application} to \textit{Monoidal} introduced \textit{fmap app} (in the definition of \((\ast)\)). Because of the meaning of \textit{fmap} on \textit{\texttt{on}}\textit{v}, we will need a derivative for \textit{app}. Fortunately, \textit{app} is fairly easy to differentiate. Allowing only \(x\) to vary (while holding \(f\) constant), \(f x\) changes just as \(f\) changes at \(x\), so the second partial derivative of \textit{app} at \((f, x)\) is \(d f \ x\). Allowing only \(f\) to vary, \(f x\) is \textit{linear} in \(f\), so it (considered as a linear map) is its own derivative. That is, using \((\ast)\) as \textit{infix} function application,

\[
\begin{align*}
\text{d} \ (\$ \ x) \ f & \equiv \text{linear} \ (\$ \ x) \\
\text{d} \ (f \ \$) \ x & \equiv d f \ x
\end{align*}
\]

As mentioned in Section 9, \((\otimes)\) takes the place of \textit{dot product} for combining contributions from partial derivatives, so

\[
\text{d app} \ (f, x) = \text{linear} \ (\$ \ x) \ast d f \ x
\]

There is an alternative to using \textit{app}. Recall that \textit{app} was introduced in interpreting \((\ast)\) from the \textit{Application} interface in terms of \textit{fmap} and \((\ast)\). The numeric instances only use \((\ast)\) indirectly, via \textit{liftA}$_2$. We can instead define \textit{liftA}$_2$ more directly:

\[
\text{liftA}$_2$ \ h as bs = \text{fmap} \ (\text{uncurry} \ h) \ (as \ast bs)
\]

Now we need to differentiate \textit{uncurry} \(h\) instead of \textit{app}, which is equivalent to taking the two partial derivatives of \(h\). For instance,

\[
\begin{align*}
\text{d} \ (\text{uncurry} \ (+)) & \equiv \text{dAdd} \\
\text{d} \ (\text{uncurry} \ (\ast)) & \equiv \text{dMul}
\end{align*}
\]

where

\[
\begin{align*}
\text{dAdd} \ (x, y) & \equiv \text{idL} \circ \text{idL} \\
\text{dMul} \ (x, y) & \equiv \text{linear} \ (\ast y) \circ \text{linear} \ (x\ast)
\end{align*}
\]

The usual derivative rules for \((+)\) and \((\ast)\) follow from the \textit{liftA}$_2$-based definitions, and so needn’t be programmed explicitly.

\[
\begin{align*}
\text{D} \ x \ x' + D \ y \ y' & \equiv \text{liftA} \ (+) \ (D x x') \ (D y y') \\
\text{fmap} \ (\text{uncurry} \ (+)) & \equiv fmap \ (\text{uncurry} \ (+)) \ (D x x' \ast D y y') \\
\text{fmap} \ (\text{uncurry} \ (+)) & \equiv fmap \ (\text{uncurry} \ (+)) \ (D (x, y) \ (x' \ast y')) \\
\text{D} \ (\text{uncurry} \ (+)) \ (x, y) & \equiv (\text{fmap} \ (\text{uncurry} \ (+))) \ (x, y) \circ(x' \ast y')) \\
\text{D}(x + y) & \equiv d \ (\text{uncurry} \ (+)) \ (x, y) \circ(x' \ast y')) \\
\text{D} \ (x + y) \ ((\text{idL} \circ \text{idL}) \ (x' \ast y')) & \equiv D \ (x + y) \ (x' + y')
\end{align*}
\]

10.4 Fun with rules

Let’s back up to our more elegant method definitions:

\[
\begin{align*}
(\ast) & \equiv \text{liftA} \ (\ast) \\
\text{sin} & \equiv \text{fmap} \ \text{sin} \\
\text{sqrt} & \equiv \text{fmap} \ \text{sqrt} \\
\end{align*}
\]

...
{- RULES

d (uncurry (+)) = dAdd

d (uncurry (⋆)) = dMul

d negate = = − 1

d ⌲ | ⌱ = signum

d signum = = 0

d recip = = − sqr recip

d exp = = exp

d log = = recip

d sqrt = = recip (2 * sqrt)

d sin = = cos

d cos = = − sin

d asin = = recip (sqrt (1 − sqr))

d acos = = recip (− sqrt (1 − sqr))

d atan = = recip (1 + sqr)

d sinh = = sinh

d cosh = = cosh

d asinh = = recip (sqrt (1 + sqr))

d acosh = = recip (− sqrt (sqr − 1))

d atanh = = recip (1 − sqr)

-}

10.5 A problem with type constraints

There is a problem with the Functor and Monoidal (and hence Applicative) instances derived in above. In each case, the method definitions type-check only for type parameters that are vector spaces. The standard definitions of Functor and Applicative in Haskell do not allow for such constraints. The problem is not in the categorical notions, but in their specialization (often adequate and convenient) in the standard Haskell library. For this reason, we’ll need a variation on these standard classes, either specific for use with vector spaces or generalized. The definitions above work with the following variations, parameterized over a scalar type s:

```haskell
class Functor s f where
    fmap :: (Vector s u, Vector s v) ⇒ (u ⌲ v) → (f u → f v)

class Functor s f ⇒ Monoidal s f where
    unit :: f ()
    (⋆) :: (Vector s u, Vector s v) ⇒ f u ⌲ f v → f (u ⌲ v)
```

While we are altering the definition of Functor, we can make another change. Rather than working with any function, limit the class to accepting only differentiable functions. A simple representation of a differentiable function is a function and its derivative:

```haskell
data u ⌲ v = FD (u ⌲ v) (u → (u → v))
```

This representation allows a simple and effective implementation of d:

```haskell
d :: (Vector s u, Vector s v) ⇒ (u ⌲ v) → (u → (u → v))

d (FD f f') = f'
```

With these definitions, the simple numeric method definitions (via fmap and liftA2) are again executable, provided that the functions passed to fmap are replaced by differentiable versions.

10.6 Generalized derivative towers

To compute infinitely many derivatives, begin with a derivative tower type and a theoretical means of constructing towers from a function:

```haskell
data u ⌲ v = D v (u ⌲ v (u → v))

toD :: (Vector s u, Vector s v) ⇒ u → (u → v) → u ⌲ v
toD_s f = D (f x) (toD_s (d f))
```

The naturality (functor morphism) property is

```haskell
fmap g ⌲ toD_s ≡ toD_s ⌲ fmap g
```

As before, let’s massage this specification into a form that is easy to satisfy. First, η-expand, and fill in the definition of toD:

```haskell
fmap g (D (f x)) (toD_s (d f))
≡ D ((fmap g f) x) (toD_s (d (f g)))
```

Next, simplify the right side, inductively assuming the Functor and Applicative morphisms inside the derivative part of D.

```haskell
D ((g o f) x) (toD_s (d (g o f)))
≡ \{ Generalized chain rule \}
D (g (f x)) (toD_s ((d g o f) ⌲ d f))
≡ \{ liftA2 morphism, (∝) \equiv liftA2 (∥) \}
D (g (f x)) (toD_s (d g o f) ⌲ toD_s (d f))
≡ \{ fmap ≡ (∝) on functions \}
D (g (f x)) (toD_s (fmap (d g) f) ⌲ toD_s (d f))
≡ \{ fmap morphism \}
D (g (f x)) (fmap (d g) (toD_s f) ⌲ toD_s (d f))
```

Summarizing, toD_s is a Functor morphism iff

```haskell
fmap g (D (f x)) (toD_s (d f))
≡ D (g (f x)) (fmap (d g) (toD_s f) ⌲ toD_s (d f))
```

Given this form of the morphism condition, and recalling the definition of toD, it’s easy to find a fmap definition for (⋆):

```haskell
fmap g f xs (D f x dfx) = D (g f x) (fmap (d g) f xs ⌲ dfx)
```

The only change from (▷) is (∝) in place of (∥).

Again, this definition can be refactored, followed by replacing the non-effective applications of d with known derivatives. Alternatively, replace arbitrary functions with differentiable functions (u → v), as in Section 10.5, so as to make this definition executable as is.

The Monoidal derivation goes through as before, adding an induction step. The instance definition is exactly as with (▷) above. The only difference is using (⋆) in place of (▷).

```haskell
instance Vector s u ⇒ Monoidal s ((u ⌲) u) where
    unit ⇒ D ()
    D u u' ⌲ D v v' = D (u, v) (u' ⊸ v')
```

To optimize out zeros in either u ⊸ v or u ⌲ v, add a Maybe around the derivative part of the representation, as described in Section 7. The zero-optimizations are entirely localized to the definitions of fmap and (⋆). To handle the Nothing vs Just, add an extra fmap in the fmap definition, and add another liftA2 in the (⋆) definition. For instance,

```haskell
data u ⌲ v = D v (Maybe (u ⌲ v (u → v)))

fmap g f xs (D f x dfx) =
D (g f x) (fmap (d g) f xs ⌲ dfx)
```

11. Related work

Jerzy Karczmarczuk (2001) first demonstrated the idea of an infinite “lazy tower of derivatives”, giving a lazy functional implementation. His first-order warm-up was similar to Figure 2, with the
higher-order (tower) version somewhat more complicated by the introduction of streams of derivatives. Building on Jerzy’s work, this paper implements the higher-order case with the visual simplicity of the first-order formulation (Figure 2). It also improves on that simplicity by means of numeric instances for functions, leading to Figure 5. Another improvement is optimizing zeros without clattering the implementation (Section 7). In contrast, (Karczmarczuk 2001) and others had twice as many cases to handle for unary methods, and four times as many for binary.

Jerzy’s AD implementation was limited to scalar types, although he hinted at a vector extension in (Karczmarczuk 1999), using an explicit list of partial derivatives.

These hints were later fleshed out for the higher-order case in (Foutz 2008), replacing lists with (nested) sparse arrays (represented as fast integer maps). The constant-optimizations there complicated matters but had an advantage over the version in this paper. In addition to having constant vs non-constant constructors (and hence many more cases to define), each sparse array can have any subset of its partial derivatives missing, avoiding still more multiplications and additions with zero. To get the same benefit, one might use a linear map representation based on partial functions.

Pearlmutter and Siskind (2007) also extended higher-order forward-mode AD to the multi-dimensional case. They remarked:

The crucial additional insight here, both for developing the extension and for demonstrating its correctness, involves reformulating Karczmarczuk’s method using Taylor expansions instead of the chain rule.

The expansions involve introducing non-standard “ε” values, which come from dual numbers. Each ε must be generated, managed, and carefully distinguished from others, so as to avoid problems of nested use described and addressed in (Siskind and Pearlmutter 2005, 2008). In contrast, the method in this paper is based on the chain rule, while still handling multi-dimensional AD. I don’t know whether the tricky nesting problem arises with the formulation in this paper (based on linear maps).

Henrik Nilsson (2003) extended higher-order AD to work on a generalized notion of functions that includes Dirac impulses, allowing for more elegant functional specification of behaviors involving instantaneous velocity changes. These derivatives were for functions over a scalar domain (time).

Doug McIlroy (2001) demonstrated some beautiful code for manipulating infinite power series. He gave two forms, Horner and MacLaurin, and their supporting operations. The MacLaurin form is especially similar, under the skin, to working with lazy derivative towers. Doug also examined efficiency and warns that “the product operators for MacLaurin and Horner forms respectively take O(2^n) and O(n^5) coefficient-domain operations to compute n terms.” He goes on to suggest computing products by conversion to and from the Horner representation. I think the exponential complexity can apply in the formulations in (Karczmarczuk 2001) and in this paper.

I am not aware of work on AD for general vector spaces, nor on deriving AD from a specification.

12. Future work

Reverse and mixed mode AD. Forward-mode AD uses the chain rule in a particular way: in compositions g ◦ f, g is always a primitive function, while f may be complex. Reverse-mode AD uses the opposite decomposition, with f being primitive, while mixed-mode combines styles. Can the specification in this paper be applied, as is, to these other AD modes? Can the derivations be successfully adapted to yield general, efficient, and elegant implementations of reverse and mixed mode AD, particularly in the general setting of vector spaces?

Richer manifold structure. Calculus on vector spaces is the foundation of calculus on rich manifold structures stitched together out of simpler pieces (ultimately vector spaces). Explore differentiation in the setting of these rich structures.

Efficiency analysis. Forward-mode AD for \( \mathbb{R}^n \rightarrow \mathbb{R} \) is described as requiring \( n \) passes and therefore inefficient. The method in this paper makes only one pass. That pass manipulates linear maps instead of scalars, which could be as expensive as \( n \) passes, but it might not need to be.

13. Acknowledgments

I’m grateful for comments from Anonymous, Barak Pearlmutter, Mark Rafter, and Paul Liu.

References


A. Efficient linear maps

A.1 Basis types

A basis of a vector space \( V \) is a subset \( B \) of \( V \), such that the elements of \( B \) span \( V \) and are linearly independent. That is to say, every element (vector) of \( V \) is a linear combination of elements of \( B \), and no element of \( B \) is a linear combination of the other elements of \( B \). Moreover, every basis determines a unique decomposition of any member of \( V \) into coordinates relative to \( B \).

Since Haskell doesn’t have subtyping, we can’t represent a basis type directly as a subset. Instead, for an arbitrary vector space \( V \), represent a distinguished basis as an associated type (Chakravarty et al. 2005), \( \text{Basis } v \), and a function that interprets a basis representation as a vector. Another method extracts coordinates (coefficients) for a vector with respect to basis elements.

```haskell
class (Vector s v, Enumerable (Basis v)) => HasBasis s v where
type Basis v :: *
basisValue :: Basis v → v
coord :: v → (Basis v → s)
```

The \( \text{Enumerable} \) constraint enables enumerating basis elements for application of linear maps (Section A.2). It has one method that enumerates all of the elements in a type:

```haskell
class Enumerable a where enumerate :: [a]
```

There are instances for \( () \), scalars, and sums of enumerable types.

A.1.1 Primitive bases

Since \( () \) is zero-dimensional, its basis is the \( \text{Void} \) type. The distinguished basis of a one-dimensional space has only one element, which can be represented with no information. Its corresponding value is 1.

```haskell
instance HasBasis Double Double where
type Basis Double = ()
basisValue () = 1
coord s = const s
```

A.1.2 Composing bases

Given vector spaces \( u \) and \( v \), a basis element for \( (u, v) \) will be one basis representation or the other, tagged with \( \text{Left} \) or \( \text{Right} \). The vectors corresponding to these basis elements are \( (u, 0) \) or \( (0, v) \), where \( u \) corresponds to a basis element for \( u \), and \( v \) for \( v \). As expected then, the dimensionality of the cross product is the sum of the dimensions. The decomposition of a vector \((u, v)\) contains left-tagged decompositions of \( u \) and right-tagged decompositions of \( v \).

```haskell
instance (HasBasis s u, HasBasis s v) ⇒ HasBasis s (u, v) where
type Basis (u, v) = Basis u ‘Either’ Basis v
basisValue (Left a) = (basisValue a, 0)
basisValue (Right b) = (0, basisValue b)
coord (u, v) = coord u ‘either’ coord v
```

A simple trick fixes this efficiency problem: \( \text{memoize} \) the linear map. We could do the memoization privately, e.g.,

```
linear f = memo (f ∘ basisValue)
```

A.2 Linear maps

Semantically, a linear map is a function \( f :: a → b \) such that, for all scalar values \( s \) and “vectors” \( u, v :: a \), the following properties hold:

\[
f \left( s \cdot u \right) = s \cdot f \left( u \right)
f \left( u + v \right) = f \left( u \right) + f \left( v \right)
\]

By repeated application of these properties,

\[
f \left( s_1 \cdot u_1 + \ldots + s_n \cdot u_n \right) = s_1 \cdot f \left( u_1 \right) + \ldots + s_n \cdot f \left( u_n \right)
\]

Taking the \( u_i \) as basis vectors, this form implies that a linear function is determined by its behavior on any basis of its domain type.

Therefore, a linear function can be represented simply as a function from a basis, using the representation described in Section A.1.

```
type u → v = Basis u → v
```

The semantic function converts from \((u → v)\) to \((u → v)\). It decomposes a source vector into its coordinates, applies the basis function to basis representations, and linearly combines the results.

```
lapply :: (Vector s u, Vector s v) ⇒ (u → v) → (u → v)
lapply uv v = sumV [coord u e ∙ v e | e ← enumerate]
```

or

```
lapply lm = linearCombo ∘ fmap (first lm) ∘ decompose
```

The inverse function is easier. Convert a function \( f \), presumed linear, to a linear map representation:

```
linear :: (Vector s u, Vector s v, HasBasis u) ⇒ (u → v) → (u → v)
```

It suffices to apply \( f \) to basis values:

```
linear f = f ∘ basisValue
```

The \( \text{coord} \) method can be changed to return \( v → s \), which is the dual of \( v \).

A.2.1 Memoization

The idea of the linear map representation is to reconstruct an entire (linear) function out of just a few samples. In other words, we can make a very small sampling of function’s domain, and re-use those values in order to compute the function’s value at all domain values. As implemented above, however, this trick makes function application more expensive, not less. If \( lm = linear f \), then each use of \( \text{lapply } lm \) can apply \( f \) to the value of every basis element, and then linearly combine results.

A simple trick fixes this efficiency problem: \( \text{memoize} \) the linear map. We could do the memoization privately, e.g.,

```
linear f = memo (f ∘ basisValue)
```

If \( lm = linear f \), then no matter how many times \( \text{lapply } lm \) is applied, the function \( f \) can only get applied as many times as the dimension of the domain of \( f \).

However, there are several other ways to make linear maps, and it would be easy to forget to memoize each combining form. So, instead of the function representation above, ensure that the function be memoized by representing it as a memo trie (Hinze 2000; Elliott 2009a).

```
type u → v = Basis u → v
```
The conversion functions \textit{linear} and \textit{lapply} need just a little tweaking. Split \textit{memo} into its definition \textit{untrie} \circ \textit{trie}, and then move \textit{untrie} into \textit{lapply}. We’ll also have to add \textit{HasTrie} constraints:

\begin{verbatim}
linear :: (Vector s u, Vector s v, HasBasis s u, HasTrie (Basis u)) \Rightarrow (u \rightarrow v) \rightarrow (u \circ v)
linear f = trie (f \circ basisValue)
lapply :: (Vector s u, Vector s v, HasBasis s u, HasTrie (Basis u)) \Rightarrow (u \circ v) \rightarrow (u \rightarrow v)
lapply lm = linearCombo \circ fmap (first (untrie lm)) \circ decompose
\end{verbatim}

Now we can build up linear maps conveniently and efficiently by using the \textit{Functor} and \textit{Applicative} instances for memo tries (Elliott 2009a). For instance, suppose that \( h \) is a linear function of two arguments (linear in both, not it each) and \( m \) and \( n \) are two linear maps. Then \( \text{liftA}_2 \ h \ m \ n \) is the linear map that applies \( h \) to the results of \( m \) and \( n \).

\begin{verbatim}
lapply (liftA_2 h m n) a = h (lapply m a) (lapply n a)
\end{verbatim}

Exploiting the applicative functor instance for functions, we get another formulation:

\begin{verbatim}
lapply (liftA_2 h m n) = liftA_2 h (lapply m) (lapply n)
\end{verbatim}

In other words, the meaning of a \( \text{liftA}_2 \) is the \( \text{liftA}_2 \) of the meanings.