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:

$$d\,f\,x \equiv \lim_{h\to 0} \frac{f(x+h) - f\,x}{h} \qquad (1)$$

$$\begin{aligned}
d\,(u+v) &\equiv d\,u + d\,v \\
d\,(u\cdot v) &\equiv d\,v\cdot u + d\,u\cdot v \\
d\,(-u) &\equiv -d\,u \\
d\,(e^u) &\equiv d\,u\cdot e^u \\
d\,(\log u) &\equiv d\,u/u \\
d\,(\sqrt{u}) &\equiv d\,u/(2\cdot\sqrt{u}) \\
d\,(\sin u) &\equiv d\,u\cdot\cos u \\
d\,(\cos u) &\equiv d\,u\cdot(-\sin u) \\
d\,(\sin^{-1} u) &\equiv d\,u/\sqrt{1-u^2} \\
d\,(\cos^{-1} u) &\equiv -d\,u/\sqrt{1-u^2} \\
d\,(\tan^{-1} u) &\equiv d\,u/(u^2+1) \\
d\,(\sinh u) &\equiv d\,u\cdot\cosh u \\
d\,(\cosh u) &\equiv d\,u\cdot\sinh u \\
d\,(\sinh^{-1} u) &\equiv d\,u/\sqrt{u^2+1} \\
d\,(\cosh^{-1} u) &\equiv -d\,u/\sqrt{u^2-1} \\
d\,(\tanh^{-1} u) &\equiv d\,u/(1-u^2)
\end{aligned}$$

Figure 1. Some rules for symbolic differentiation

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

$$d\,f\,x \approx \frac{f(x+h) - f\,x}{h}$$

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 inconvience 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 (foward-mode)

```
data D a = D a a deriving (Eq, Show)

constD :: Num a ⇒ a → D a
constD x = D x 0

idD :: Num a ⇒ a → D a
idD x = D x 1

instance Num a ⇒ Num (D a) where
  fromInteger x     = constD (fromInteger x)
  D x x' + D y y'   = D (x + y) (x' + y')
  D x x' * D y y'   = D (x * y) (y' * x + x' * y)
  negate  (D x x')  = D (negate x) (negate x')
  signum  (D x _)   = D (signum x) 0
  abs     (D x x')  = D (abs x) (x' * signum x)

instance Fractional x ⇒ Fractional (D x) where
  fromRational x = constD (fromRational x)
  recip (D x x')  = D (recip x) (−x' / sqr x)

sqr :: Num a ⇒ a → a
sqr x = x * x

instance Floating x ⇒ Floating (D x) where
  π              = constD π
  exp  (D x x') = D (exp  x) (x' * exp x)
  log  (D x x') = D (log  x) (x' / x)
  sqrt (D x x') = D (sqrt x) (x' / (2 * sqrt x))
  sin  (D x x') = D (sin  x) (x' * cos x)
  cos  (D x x') = D (cos  x) (x' * (−sin x))
  asin (D x x') = D (asin x) (x' / sqrt (1 − sqr x))
  acos (D x x') = D (acos x) (x' / (−sqrt (1 − sqr x)))
                  ...
```

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 *abs* and *signum* need more care at 0.)

As an example, define

```
f₁ :: Floating a ⇒ a → a
f₁ z = sqrt (3 * sin z)
```

and try it out in GHCi:

```
*Main> f1 (D 2 1)
D 1.6516332160855343 (-0.3779412091869595)
```

To test correctness, here is a symbolically differentiated version:

```
f₂ :: Floating a ⇒ a → D a
f₂ x = D (f₁ x) (3 * cos x / (2 * sqrt (3 * sin x)))
```

Try it out in GHCi:

```
*Main> f2 2
D 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 → a) → (a → a)   -- first attempt
```

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

```
d (λx → u x + v x) ≡ λx → d u x + d v x
```

and the *log* rule means

```
d (λx → log (u x)) ≡ λx → 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 overloadings for *functions*, so that numeric operations apply *point-wise*. For instance,

```
u + v ≡ λx → u x + v x
log u ≡ λx → 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.

```
instance Functor ((→) t) where
  fmap f g = f ∘ g
instance Applicative ((→) t) where
  pure     = const
  f ⊛ g    = λt → (f t) (g t)
```

Consequently,

```
liftA₂ h u v    ≡ λx → h (u x) (v x)
liftA₃ h u v w  ≡ λx → h (u x) (v x) (w x)
...
```

Figure 3. Standard *Functor* and *Applicative* instances for functions

```
instance Num b ⇒ Num (a → b) where
  fromInteger  = pure ∘ fromInteger
  negate       = fmap negate
  (+)          = liftA₂ (+)
  (∗)          = liftA₂ (∗)
  abs          = fmap abs
  signum       = fmap signum
instance Fractional b ⇒ Fractional (a → b) where
  fromRational = pure ∘ fromRational
  recip        = fmap recip
instance Floating b ⇒ Floating (a → b) where
  π            = pure π
  sqrt         = fmap sqrt
  exp          = fmap exp
  log          = fmap log
  sin          = fmap sin
               ...
```

Figure 4. Numeric overloadings for function types

3. A scalar chain rule

Many of the laws in Figure 1 look similar: $d\ (f\ u) = d\ u * f'\ u$ 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.

$$d\ (g \circ f)\ x \equiv d\ g\ (f\ x) * d\ f\ x$$

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

$$d\ (g \circ f) \equiv (d\ g \circ f) * d\ f$$

All but the first two rules in Figure 1 then follow from the chain rule. For instance,

```
  d (sin u)
≡   { sin on functions }
  d (sin ∘ u)
≡   { reformulated chain rule }
  (d sin ∘ u) ∗ d u
≡   { d sin ≡ cos }
  (cos ∘ u) ∗ d u
≡   { cos on functions }
  cos u ∗ d u
```

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, $(\bowtie)$, whose arguments are a function and its derivative.

```
infix 0 ⋈
(⋈) :: Num a ⇒ (a → a) → (a → a) → (D a → D a)
(f ⋈ f′) (D a a′) = D (f a) (a′ ∗ f′ a)
```

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

```
instance Floating x ⇒ Floating (D x) where
  π    = D π 0
  exp  = exp  ⋈ exp
  log  = log  ⋈ recip
  sqrt = sqrt ⋈ λx →  recip (2 ∗ sqrt x)
  sin  = sin  ⋈ cos
  cos  = cos  ⋈ λx → −sin x
  asin = asin ⋈ λx →  recip (sqrt (1 − sqr x))
  acos = acos ⋈ λx → −recip (sqrt (1 − sqr x))
       ...
```

4. Prettier derivatives via function overloading

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

With the help of $(\bowtie)$ and the overloadings 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. *What* does it mean, independently of implementation?
2. *How* do the implementation and its correctness flow gracefully from that meaning?
3. *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.

```
toD :: (a → a) → (a → D a)
toD f = λx → D (f x) (d f x)
```

Or, in point-free form,

```
toD f = liftA₂ D f (d f)
```

thanks to the *Applicative* instance in Figure 3.

We have no implementation of d, so this definition of 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 *type class morphisms*, as described in (Elliott 2009b,a), which is to say that the interpretation preserves the structure of every method application. For AD, the interpretation

$$\begin{array}{l}
\textbf{instance } \mathit{Num}\ a \Rightarrow \mathit{Num}\ (D\ a)\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{fromInteger} & = \mathit{constD} \circ \mathit{fromInteger} \\
D\ x_0\ x' + D\ y_0\ y' & = D\ (x_0 + y_0)\ (x' + y') \\
x@(D\ x_0\ x') * y@(D\ y_0\ y') & = D\ (x_0 * y_0)\ \ (x' * y + x * y')
\end{array} \\
\quad \begin{array}{lll}
\mathit{negate} & = \mathit{negate} & \bowtie -1 \\
\mathit{abs} & = \mathit{abs} & \bowtie \mathit{signum} \\
\mathit{signum} & = \mathit{signum} & \bowtie 0
\end{array} \\
\textbf{instance } \mathit{Fractional}\ a \Rightarrow \mathit{Fractional}\ (D\ a)\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{fromRational} & = \mathit{constD} \circ \mathit{fromRational} \\
\mathit{recip} & = \mathit{recip} \bowtie -\mathit{sqr}\ \mathit{recip}
\end{array} \\
\textbf{instance } \mathit{Floating}\ a \Rightarrow \mathit{Floating}\ (D\ a)\ \textbf{where} \\
\quad \begin{array}{lll}
\pi & = \mathit{constD}\ \pi & \\
\mathit{exp} & = \mathit{exp} & \bowtie \mathit{exp} \\
\mathit{log} & = \mathit{log} & \bowtie \mathit{recip} \\
\mathit{sqrt} & = \mathit{sqrt} & \bowtie \mathit{recip}\ (2 * \mathit{sqrt}) \\
\mathit{sin} & = \mathit{sin} & \bowtie \mathit{cos} \\
\mathit{cos} & = \mathit{cos} & \bowtie -\mathit{sin} \\
\mathit{asin} & = \mathit{asin} & \bowtie \mathit{recip}\ (\mathit{sqrt}\ (1 - \mathit{sqr})) \\
\mathit{acos} & = \mathit{acos} & \bowtie \mathit{recip}\ (-\mathit{sqrt}\ (1 - \mathit{sqr})) \\
\mathit{atan} & = \mathit{atan} & \bowtie \mathit{recip}\ (1 + \mathit{sqr}) \\
\mathit{sinh} & = \mathit{sinh} & \bowtie \mathit{cosh} \\
\mathit{cosh} & = \mathit{cosh} & \bowtie \mathit{sinh} \\
\mathit{asinh} & = \mathit{asinh} & \bowtie \mathit{recip}\ (\mathit{sqrt}\ (1 + \mathit{sqr})) \\
\mathit{acosh} & = \mathit{acosh} & \bowtie \mathit{recip}\ (-\mathit{sqrt}\ (\mathit{sqr} - 1)) \\
\mathit{atanh} & = \mathit{atanh} & \bowtie \mathit{recip}\ (1 - \mathit{sqr})
\end{array}
\end{array}$$

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

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

$$\begin{array}{ll}
\mathit{toD}\ (u + v) & \equiv \mathit{toD}\ u + \mathit{toD}\ v \\
\mathit{toD}\ (u * v) & \equiv \mathit{toD}\ u * \mathit{toD}\ v \\
\mathit{toD}\ (\mathit{negate}\ u) & \equiv \mathit{negate}\ (\mathit{toD}\ u) \\
\mathit{toD}\ (\mathit{sin}\ u) & \equiv \mathit{sin}\ (\mathit{toD}\ u) \\
\mathit{toD}\ (\mathit{cos}\ u) & \equiv \mathit{cos}\ (\mathit{toD}\ u) \\
 & \ldots
\end{array}$$

Note here that the numeric operations are applied to values of type $a \to a$ on the left, and to values of type $a \to D\ a$ 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:

$$\begin{array}{ll}
\mathit{constD} & :: \mathit{Num}\ a \Rightarrow a \to D\ a \\
\mathit{idD} & :: \mathit{Num}\ a \Rightarrow a \to D\ a
\end{array}$$

$$\begin{array}{ll}
\mathit{constD}\ x & \equiv \mathit{toD}\ (\mathit{const}\ x)\ \bot \\
\mathit{idD} & \equiv \mathit{toD}\ \mathit{id}
\end{array}$$

To derive implementations, expand toD and simplify.

$$\begin{array}{l}
\quad \mathit{constD}\ x \\
\equiv \quad \{ \text{specification} \} \\
\quad \mathit{toD}\ (\mathit{const}\ x)\ \bot \\
\equiv \quad \{ \text{definition of } \mathit{toD} \} \\
\quad D\ (\mathit{const}\ x\ \bot)\ (d\ (\mathit{const}\ x)\ \bot) \\
\equiv \quad \{ \text{definition of } \mathit{const} \text{ and its derivative} \} \\
\quad D\ x\ 0
\end{array}$$

$$\begin{array}{l}
\quad \mathit{idD}\ x \\
\equiv \quad \{ \text{specification} \} \\
\quad \mathit{toD}\ (\mathit{id}\ x) \\
\equiv \quad \{ \text{definition of } \mathit{toD} \} \\
\quad D\ (\mathit{id}\ x)\ (d\ \mathit{id}\ x) \\
\equiv \quad \{ \text{definition of } \mathit{id} \text{ and its derivative} \} \\
\quad D\ x\ 1
\end{array}$$

In (Karczmarczuk 2001) and elsewhere, idD is called $dVar$ and is sometimes referred to as the "variable" of differentiation, a term more suited to symbolic differentiation than to AD.

5.2.2 Addition

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

$$\mathit{toD}\ (u + v) \equiv \mathit{toD}\ u + \mathit{toD}\ v$$

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

$$\begin{array}{l}
\quad \mathit{toD}\ (u + v) \\
\equiv \quad \{ \text{definition of } \mathit{toD} \} \\
\quad \mathit{liftA}_2\ D\ (u + v)\ (d\ (u + v)) \\
\equiv \quad \{ d\ (u + v) \text{ from Figure 1} \} \\
\quad \mathit{liftA}_2\ D\ (u + v)\ (d\ u + d\ v) \\
\equiv \quad \{ \mathit{liftA}_2 \text{ on functions from Figure 3} \} \\
\quad \lambda x \to D\ ((u + v)\ x)\ ((d\ u + d\ v)\ x) \\
\equiv \quad \{ (+) \text{ on functions from Figure 4} \} \\
\quad \lambda x \to D\ (u\ x + v\ x)\ (d\ u\ x + d\ v\ x)
\end{array}$$

Then start over with the right-hand side:

$$\begin{array}{l}
\quad \mathit{toD}\ u + \mathit{toD}\ v \\
\equiv \quad \{ (+) \text{ on functions from Figures 3 and 4} \} \\
\quad \lambda x \to \mathit{toD}\ u\ x + \mathit{toD}\ v\ x \\
\equiv \quad \{ \text{definition of } \mathit{toD} \} \\
\quad \lambda x \to D\ (u\ x)\ (d\ u\ x) + D\ (v\ x)\ (d\ v\ x)
\end{array}$$

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

$$\begin{array}{l}
\quad \lambda x \to D\ (u\ x + v\ x)\ (d\ u\ x + d\ v\ x) \\
\equiv \\
\quad \lambda x \to D\ (u\ x)\ (d\ u\ x) + D\ (v\ x)\ (d\ v\ x)
\end{array}$$

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

$$\mathit{toD}\ (u + v) \equiv \mathit{toD}\ u + \mathit{toD}\ v$$

5.2.3 Multiplication

The specification:

$$\mathit{toD}\ (u * v) \equiv \mathit{toD}\ u * \mathit{toD}\ v$$

Reason similarly to the addition case. Begin with the left hand side:

$$
\begin{aligned}
& toD\ (u * v) \\
\equiv\ & \{\ \text{definition of } toD\ \} \\
& liftA_2\ D\ (u * v)\ (d\ (u * v)) \\
\equiv\ & \{\ d\ (u * v) \text{ from Figure 1 }\} \\
& liftA_2\ D\ (u * v)\ (d\ u * v + d\ v * u) \\
\equiv\ & \{\ liftA_2 \text{ on functions from Figure 3 }\} \\
& \lambda x \to D\ ((u * v)\ x)\ ((d\ u * v + d\ v * u)\ x) \\
\equiv\ & \{\ (*) \text{ and } (+) \text{ on functions from Figure 4 }\} \\
& \lambda x \to D\ (u\ x * v\ x)\ (d\ u\ x * v\ x + d\ v\ x * u\ x)
\end{aligned}
$$

Then start over with the right-hand side:

$$
\begin{aligned}
& toD\ u * toD\ v \\
\equiv\ & \{\ (*) \text{ on functions }\} \\
& \lambda x \to toD\ u\ x * toD\ v\ x \\
\equiv\ & \{\ \text{definition of } toD\ \} \\
& \lambda x \to D\ (u\ x)\ (d\ u\ x) * D\ (v\ x)\ (d\ v\ x)
\end{aligned}
$$

Sufficient definition:

$$D\ a\ a' * D\ b\ b' = D\ (a + b)\ (a' * b + b' * a)$$

5.2.4 Sine

The specification:

$$toD\ (sin\ u) \equiv sin\ (toD\ u)$$

Simplify the left-hand side:

$$
\begin{aligned}
& toD\ (sin\ u) \\
\equiv\ & \{\ \text{definition of } toD\ \} \\
& liftA_2\ D\ (sin\ u)\ (d\ (sin\ u)) \\
\equiv\ & \{\ d\ (sin\ u)\ \} \\
& liftA_2\ D\ (sin\ u)\ (d\ u * cos\ u) \\
\equiv\ & \{\ liftA_2 \text{ on functions }\} \\
& \lambda x \to D\ ((sin\ u)\ x)\ ((d\ u * cos\ u)\ x) \\
\equiv\ & \{\ sin, (*) \text{ and } cos \text{ on functions }\} \\
& \lambda x \to D\ (sin\ (u\ x))\ (d\ u\ x * cos\ (u\ x))
\end{aligned}
$$

and then the right:

$$
\begin{aligned}
& sin\ (toD\ u) \\
\equiv\ & \{\ sin \text{ on functions }\} \\
& \lambda x \to sin\ (toD\ u\ x) \\
\equiv\ & \{\ \text{definition of } toD\ \} \\
& \lambda x \to sin\ (D\ (u\ x)\ (d\ u\ x))
\end{aligned}
$$

So a sufficient definition is

$$sin\ (D\ a\ a') = D\ (sin\ a)\ (a' * cos\ a)$$

Or, using the chain rule operator,

$$sin = sin \bowtie cos$$

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".

$$\mathbf{data}\ D\ a = D\ a\ (D\ a)$$

The toD function easily adapts to this new D type:

$$
\begin{aligned}
& toD :: (a \to a) \to (a \to D\ a) \\
& toD\ f\ x = D\ (f\ x)\ (toD\ (d\ f)\ x)
\end{aligned}
$$

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:

$$
\begin{aligned}
& toD\ (u + v) \\
\equiv\ & \{\ \text{definition of } toD\ \} \\
& liftA_2\ D\ (u + v)\ (toD\ (d\ (u + v))) \\
\equiv\ & \{\ d\ (u + v)\ \} \\
& liftA_2\ D\ (u + v)\ (toD\ (d\ u + d\ v)) \\
\equiv\ & \{\ \text{induction for } toD\ /\ (+)\ \} \\
& liftA_2\ D\ (u + v)\ (toD\ (d\ u) + toD\ (d\ v)) \\
\equiv\ & \{\ \text{definition of } liftA_2 \text{ and } (+) \text{ on functions }\} \\
& \lambda x \to D\ (u\ x + v\ x)\ (toD\ (d\ u)\ x + toD\ (d\ v)\ x)
\end{aligned}
$$

and then the right:

$$
\begin{aligned}
& toD\ u + toD\ v \\
\equiv\ & \{\ (+) \text{ on functions }\} \\
& \lambda x \to toD\ u\ x + toD\ v\ x \\
\equiv\ & \{\ \text{definition of } toD\ \} \\
& \lambda x \to D\ (u\ x)\ (toD\ (d\ u\ x)) + D\ (v\ x)\ (toD\ (d\ v\ x))
\end{aligned}
$$

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

$$
\begin{aligned}
& \lambda x \to D\ (u\ x + v\ x)\ (toD\ (d\ u)\ x + toD\ (d\ v)\ x) \\
\equiv\ & \\
& \lambda x \to D\ (u\ x)\ (toD\ (d\ u)\ x) + D\ (v\ x)\ (toD\ (d\ v)\ x)
\end{aligned}
$$

Again, an easy choice is

$$D\ a_0\ a' + D\ b_0\ b' = D\ (a_0 + b_0)\ (a' + b')$$

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:

$$
\begin{aligned}
& toD\ (u * v) \\
\equiv\ & \{\ \text{definition of } toD\ \} \\
& liftA_2\ D\ (u * v)\ (toD\ (d\ (u * v))) \\
\equiv\ & \{\ d\ (u * v)\ \} \\
& liftA_2\ D\ (u * v)\ (toD\ (d\ u * v + d\ v * u)) \\
\equiv\ & \{\ \text{induction for } toD\ /\ (+)\ \} \\
& liftA_2\ D\ (u * v)\ (toD\ (d\ u * v) + toD\ (d\ v * u)) \\
\equiv\ & \{\ \text{induction for } toD\ /\ (*)\ \} \\
& liftA_2\ D\ (u * v)\ (toD\ (d\ u) * toD\ v + \\
& \qquad toD\ (d\ v) * toD\ u) \\
\equiv\ & \{\ liftA_2, (*), (+) \text{ on functions }\} \\
& \lambda x \to liftA_2\ D\ (u\ x * v\ x)\ (toD\ (d\ u)\ x * toD\ v\ x + \\
& \qquad toD\ (d\ v)\ x * toD\ u\ x)
\end{aligned}
$$

and then on the right:

$$\begin{aligned}
&\mathit{toD}\ u * \mathit{toD}\ v\\
\equiv&\quad \{\text{ definition of } \mathit{toD}\ \}\\
&\mathit{liftA}_2\ D\ u\ (\mathit{toD}\ (d\ u)) * \mathit{liftA}_2\ D\ v\ (\mathit{toD}\ (d\ v))\\
\equiv&\quad \{\ \mathit{liftA}_2 \text{ and } (*) \text{ on functions }\}\\
&\lambda x \rightarrow D\ (u\ x)\ (\mathit{toD}\ (d\ u)\ x) * D\ (v\ x)\ (\mathit{toD}\ (d\ v)\ x)
\end{aligned}$$

A sufficient definition is

$$a@(D\ a_0\ a') * b@(D\ b_0\ b') = D\ (a_0 + b_0)\ (a' * b + b' * a)$$

because

$$\mathit{toD}\ u\ x \equiv D\ (u\ x)\ (\mathit{toD}\ (d\ u)\ x)$$

$$\mathit{toD}\ v\ x \equiv D\ (v\ x)\ (\mathit{toD}\ (d\ v)\ x)$$

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:

$$\begin{aligned}
&\mathit{toD}\ (\mathit{sin}\ u)\\
\equiv&\quad \{\text{ definition of } \mathit{toD}\ \}\\
&\mathit{liftA}_2\ D\ (\mathit{sin}\ u)\ (\mathit{toD}\ (d\ (\mathit{sin}\ u)))\\
\equiv&\quad \{\ d\ (\mathit{sin}\ u)\ \}\\
&\mathit{liftA}_2\ D\ (\mathit{sin}\ u)\ (\mathit{toD}\ (d\ u * \mathit{cos}\ u))\\
\equiv&\quad \{\text{ induction for } \mathit{toD}\ /\ (*)\ \}\\
&\mathit{liftA}_2\ D\ (\mathit{sin}\ u)\ (\mathit{toD}\ (d\ u) * \mathit{toD}\ (\mathit{cos}\ u))\\
\equiv&\quad \{\text{ induction for } \mathit{toD}\ /\ \mathit{cos}\ \}\\
&\mathit{liftA}_2\ D\ (\mathit{sin}\ u)\ (\mathit{toD}\ (d\ u) * \mathit{cos}\ (\mathit{toD}\ u))\\
\equiv&\quad \{\ \mathit{liftA}_2, \mathit{sin}, \mathit{cos} \text{ and } (*) \text{ on functions }\}\\
&\lambda x \rightarrow D\ (\mathit{sin}\ (u\ x))\ (\mathit{toD}\ (d\ u)\ x * \mathit{cos}\ (\mathit{toD}\ u\ x))
\end{aligned}$$

and then the right:

$$\begin{aligned}
&\mathit{sin}\ (\mathit{toD}\ u)\\
\equiv&\quad \{\text{ definition of } \mathit{toD}\ \}\\
&\mathit{sin}\ (\mathit{liftA}_2\ D\ u\ (\mathit{toD}\ (d\ u)))\\
\equiv&\quad \{\ \mathit{liftA}_2 \text{ and } \mathit{sin} \text{ on functions }\}\\
&\lambda x \rightarrow \mathit{sin}\ (D\ (u\ x)\ (\mathit{toD}\ (d\ u)\ x))
\end{aligned}$$

To make the left and right final forms equal, define

$$\mathit{sin}\ a@(D\ a_0\ a') \equiv D\ (\mathit{sin}\ a_0)\ (a' * \mathit{cos}\ a)$$

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 $D\ a \rightarrow D\ a$ instead of $a \rightarrow a$.

$$\begin{aligned}
&\textbf{infix}\ 0 \bowtie\\
&(\bowtie) :: (\mathit{Num}\ a) \Rightarrow (a \rightarrow a) \rightarrow (D\ a \rightarrow D\ a) \rightarrow (D\ a \rightarrow D\ a)\\
&(f \bowtie f')\ a@(D\ a_0\ a') = D\ (f\ a_0)\ (a' * f'\ a)
\end{aligned}$$

With this new definition of $(\bowtie)$, *all* of the chain-rule-based definitions in Figure 5 (first-order derivatives) carry over *without change* to compute infinite derivative towers. For instance,

$$\begin{aligned}
&\textbf{instance}\ \mathit{Floating}\ a \Rightarrow \mathit{Floating}\ (D\ a)\ \textbf{where}\\
&\quad \mathit{exp} = \mathit{exp} \bowtie \mathit{exp}\\
&\quad \mathit{log} = \mathit{log} \bowtie \mathit{recip}\\
&\quad \mathit{sqrt} = \mathit{sqrt} \bowtie \mathit{recip}\ (2 * \mathit{sqrt})\\
&\quad \mathit{sin} = \mathit{sin} \bowtie \mathit{cos}\\
&\quad \mathit{cos} = \mathit{cos} \bowtie -\mathit{sin}\\
&\quad \ldots
\end{aligned}$$

Now the operators and literals on the right of $(\bowtie)$ are overloaded for the type $D\ a \rightarrow D\ a$. For instance, in the definition of sqrt,

$$\begin{aligned}
&2 && :: D\ a \rightarrow D\ a\\
&\mathit{recip} && :: (D\ a \rightarrow D\ a) \rightarrow (D\ a \rightarrow D\ a)\\
&(*) && :: (D\ a \rightarrow D\ a) \rightarrow (D\ a \rightarrow D\ a)\\
& && \quad \rightarrow (D\ a \rightarrow D\ a)
\end{aligned}$$

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:

$$\begin{aligned}
&2 * \mathit{sin}\ (\mathit{idD}\ x)\\
\equiv\ & D\ 2\ 0 * \mathit{sin}\ (D\ x\ 1)\\
\equiv\ & D\ 2\ 0 * D\ (\mathit{sin}\ x)\ (1 * \mathit{cos}\ x)\\
\equiv\ & D\ (2 * \mathit{sin}\ x)\ (0 * \mathit{sin}\ x + (1 * \mathit{cos}\ x) * 2)
\end{aligned}$$

In this case there is a wasted $\mathit{sin}\ \ x$, multiplication by zero, and addition to zero. The situation is *much* worse with higher derivatives:

$$\begin{aligned}
&2 * \mathit{sin}\ (\mathit{idD}\ x)\\
\equiv\ & D\ 2\ 0 * \mathit{sin}\ (D\ x\ 1)\\
\equiv\ & D\ 2\ 0 * D\ (\mathit{sin}\ x)\ (1 * \mathit{cos}\ (\mathit{idD}\ x))\\
\equiv\ & D\ (2 * \mathit{sin}\ x)\ (0 * \mathit{sin}\ (\mathit{idD}\ x) + (1 * \mathit{cos}\ (\mathit{idD}\ x)) * 2)
\end{aligned}$$

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

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

$$\textbf{data}\ D\ a = D\ a\ a \quad \text{-- first-order, non-optimized}$$

with

$$\textbf{data}\ D\ a = D\ a\ (\mathit{Maybe}\ a) \quad \text{-- first-order, optimized}$$

The constant and identity cases change accordingly:

$$\begin{aligned}
&\mathit{constD} :: \mathit{Num}\ a \Rightarrow a \rightarrow D\ a\\
&\mathit{constD}\ x = D\ x\ \mathit{Nothing}\\[4pt]
&\mathit{idD} :: \mathit{Num}\ a \Rightarrow a \rightarrow D\ a\\
&\mathit{idD}\ x = D\ x\ (\mathit{Just}\ 1)
\end{aligned}$$

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

$$\begin{aligned}
&\textbf{infixl}\ 6\ \mathring{+}\\
&(\mathring{+}) :: \mathit{Num}\ a \Rightarrow \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a\\
&\mathit{Nothing} \mathring{+} b' && = b'\\
&a' \qquad\quad \mathring{+} \mathit{Nothing} && = a'\\
&\mathit{Just}\ a' \ \mathring{+} \mathit{Just}\ b' && = \mathit{Just}\ (a' + b')\\[4pt]
&\textbf{infixl}\ 7\ \mathring{*}\\
&(\mathring{*}) :: \mathit{Num}\ a \Rightarrow \mathit{Maybe}\ a \rightarrow a \rightarrow \mathit{Maybe}\ a\\
&\mathit{Nothing}\ \mathring{*}\ \_ = \mathit{Nothing}\\
&\mathit{Just}\ a'\ \ \mathring{*}\ b = \mathit{Just}\ (a' * b)
\end{aligned}$$

The $(\mathring{+})$ definition mirrors the Monoid instance of $\mathit{Maybe}\ (\mathit{Sum}\ a)$; while the $(\mathring{*})$ definition is a fmap using the $\mathit{Functor}$ instance of Maybe.

The changes required for AD are minimal: simply replace addition of derivatives and multiplication by derivatives with $(\mathring{+})$ and $(\mathring{*})$ in the chain rule

$$\begin{array}{l} \mathbf{infix}\ 0 \bowtie \\ (\bowtie) :: (\mathit{Num}\ a) \Rightarrow (a \to a) \to (a \to a) \to (D\ a \to D\ a) \\ (f \bowtie f')\ (D\ a\ a') = D\ (f\ a)\ (a' \mathring{*} f'\ a) \end{array}$$

and in the derivative of sums and products

$$\begin{array}{l} D\ a\ a' + D\ b\ b' = D\ (a + b)\ (a' \mathring{+} b') \\ D\ a\ a' * D\ b\ b' = D\ (a * b)\ (a' \mathring{*} b \mathring{+} b' \mathring{*} a) \end{array}$$

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

$\mathbf{data}\ D\ a = D\ a\ (\mathit{Maybe}\ (D\ a))$ -- 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 (Karczmarczuk 2001)) was to overload numeric operators to operate on the following (co)recursive type:

$\mathbf{data}\ D\ b = D\ b\ (D\ b)$

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 $f :: \mathbb{R}^2 \to \mathbb{R}$. 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 $f :: \mathbb{R}^2 \to \mathbb{R}^3$ 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:

$$\begin{array}{l} f_1 :: \mathbb{R} \to \mathbb{R} \\ f_1\ x = x^2 + 3 * x + 1 \end{array}$$

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 $d\ f_1\ 5$ 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:

$$\begin{array}{l} f_2 :: \mathbb{R}^2 \to \mathbb{R} \\ f_2\ (x, y) = 2 * x * y + 3 * x + 5 * y + 7 \end{array}$$

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 ϑ (where east is zero and north is 90 degrees), the east and north components are $\mathit{cos}\ \vartheta$ and $\mathit{sin}\ \vartheta$, respectively. When heading in the direction ϑ, 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 $(\Leftrightarrow)$ to mean dot (inner) product, then this change in altitude is $d\ f_2\ (x, y) \Leftrightarrow (dx, dy)$.

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

$$f_3 :: \mathbb{R}^2 \to \mathbb{R}^3$$

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 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 ‘$timesVec$‘ (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 turn a small step in the function's domain into a resulting step in the range.

- In f_1, the (scalar) derivative c means $(c*)$, i.e., multiply by c.
- In f_2, the (vector) derivative v means $(v \langle\!\cdot\!\rangle)$.
- In f_3, the (matrix) derivative m means $(m$‘$timesVec$‘$)$.

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:

```
class AdditiveGroup v where
  0      :: v
  (+)    :: v → v → v
  negate :: v → v
```

Next, given a field s, a *vector space* over s adds a scaling operation:

```
class AdditiveGroup v ⇒ Vector s v where
  (·) :: s → v → v
```

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

```
class Vector s v ⇒ InnerSpace s v where
  (<·>) :: v → v → 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:

```
instance AdditiveGroup v ⇒ AdditiveGroup (a → v) where
  0      = pure 0
  (+)    = liftA₂ (+)
  negate = fmap negate
instance Vector s v ⇒ Vector s (a → v) where
  (·) s = fmap (s·)
```

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 $(\cdot) \equiv (*)$.
- 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:

```
newtype u ⊸ v = LMap (u → 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 $(\circledcirc)$ are identity and composition.

```
linear :: (Vector s u, Vector s v) ⇒ (u → v) → (u ⊸ v)
lapply :: (Vector s u, Vector s v) ⇒ (u ⊸ v) → (u → v)
id_L   :: (Vector s u) ⇒ u ⊸ u
(⊚)    :: (Vector s u, Vector s v) ⇒
            (v ⊸ w) → (u ⊸ v) → (u ⊸ w)
```

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

```
(◇) :: (Vector s u, Vector s v, Vector s w) ⇒
         (u ⊸ w) → (v ⊸ w) → ((u, v) ⊸ w)
```

Semantically,

$$(l \diamond m)\ \text{‘}lapply\text{‘}\ (da, db) \equiv l\ \text{‘}lapply\text{‘}\ da + m\ \text{‘}lapply\text{‘}\ db$$

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

$$(s', t') \langle\!\cdot\!\rangle (da, db) = s' \cdot da + t' \cdot db$$

Dually to $(\diamond)$, another way to form linear maps is by "zipping":

$$(\star) :: (w \multimap u) \to (w \multimap v) \to (w \multimap (u, v))$$

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:

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

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

$$d\ (g \circ f)\ x \equiv d\ g\ (f\ x) \circledcirc d\ f\ x$$

where $(\circledcirc)$ is *composition* of linear maps. More succinctly,

$$d\ (g \circ f) \equiv (d\ g \circ f) \mathbin{\hat{\circledcirc}} d\ f$$

using lifted composition:

$$(\hat{\circledcirc}) = liftA_2\ (\circledcirc)$$

10.1 First-order generalized derivatives

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

```
data a ⊳ b = D b (a ⊸ 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:

$$toD :: (Vector\ s\ u, Vector\ s\ v) \Rightarrow u \to (u \to v) \to u \rhd v$$
$$toD_x\ f = D\ (f\ x)\ (d\ f\ x)$$

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 *fmap*, *pure*, or $liftA_2$ in a simple, regular pattern, e.g.,

$$\begin{aligned} fromInteger &= pure \circ fromInteger \\ (*) &= liftA_2\ (*) \\ sin &= fmap\ sin \\ &\ldots \end{aligned}$$

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

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

Consider one example:

$$sin = fmap\ sin$$

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

$$toD_x\ (sin\ u) \equiv sin\ (toD_x\ u)$$

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

$$toD_x\ (fmap\ sin\ u) \equiv fmap\ sin\ (toD_x\ u)$$

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 *fmap*.

The other two definition styles (using *pure* and $liftA_2$) work out similarly. For example, if toD_x is an $Applicative$ morphism, then

$$\begin{aligned} & toD_x\ (fromInteger\ n) \\ \equiv\ & \{\ fromInteger \text{ for } a \rhd b\ \} \\ & toD_x\ (pure\ (fromInteger\ n)) \\ \equiv\ & \{\ toD_x \text{ is an } Applicative \text{ morphism}\ \} \\ & pure\ (fromInteger\ n) \\ \equiv\ & \{\ fromInteger \text{ for functions}\ \} \\ & fromInteger\ n \end{aligned}$$

$$\begin{aligned} & toD_x\ (u * v) \\ \equiv\ & \{\ (*) \text{ for } a \rhd b\ \} \\ & toD_x\ (liftA_2\ (*)\ u\ v) \\ \equiv\ & \{\ toD_x \text{ is an } Applicative \text{ morphism}\ \} \\ & liftA_2\ (*)\ (toD_x\ u)\ (toD_x\ v) \\ \equiv\ & \{\ (*) \text{ on functions}\ \} \\ & toD_x\ u * toD_x\ v \end{aligned}$$

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 μ is a morphism on $Functor$ and $Applicative$, and the numeric instances for *both* F and G are defined as in Figure 4, then μ is a numeric morphism.

Thus, we have only to come up with $Functor$ and $Applicative$ instances for $(\rhd)\ u$ such that toD_x is a $Functor$ and $Applicative$ morphism.

10.2 Functor

First look at $Functor$. The morphism condition (*naturality*), η-expanded, is

$$fmap\ g\ (toD_x\ f) \equiv toD_x\ (fmap\ g\ f)$$

Using the definition of toD on the left gives

$$fmap\ g\ (D\ (f\ x)\ (d\ f\ x))$$

Simplifying the RHS,

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

So the morphism condition is equivalent to

$$fmap\ g\ (D\ (f\ x)\ (d\ f\ x)) \equiv D\ (g\ (f\ x))\ (d\ g\ (f\ x) \odot d\ f\ x)$$

Now it's easy to find a sufficient definition:

$$fmap\ g\ (D\ fx\ dfx) = D\ (g\ fx)\ (d\ g\ fx \odot dfx)$$

This definition is not executable, however, since d is not. Fortunately, all uses of *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 *fmap* definition, as in Section 3.

$$\textbf{instance}\ Functor\ ((\rhd a))\ \textbf{where}\ fmap\ g = g \bowtie d\ g$$
$$\begin{aligned} & (\bowtie) :: (Vector\ s\ u, Vector\ s\ v, Vector\ s\ w) \Rightarrow \\ & \qquad (v \to w) \to (v \to (v \multimap w)) \to (u \rhd v) \to (u \rhd w) \\ & (g \bowtie dg)\ (D\ fx\ dfx) = D\ (g\ fx)\ (dg\ fx \odot dfx) \end{aligned}$$

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

$$\begin{aligned} sin &= sin \bowtie d\ sin \\ sqrt &= sqrt \bowtie d\ sqrt \\ &\ldots \end{aligned}$$

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

$$\begin{aligned} sin &= sin \bowtie cos \\ sqrt &= sqrt \bowtie recip\ (2 * sqrt) \\ &\ldots \end{aligned}$$

Now we have an executable implementation again.

10.3 Applicative/Monoidal

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

$$\begin{aligned} & \textbf{class}\ Functor\ f \Rightarrow Applicative\ f\ \textbf{where} \\ & \quad pure :: a \to f\ a \\ & \quad (\circledast) :: f\ (a \to b) \to f\ a \to f\ b \end{aligned}$$

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

$$\begin{aligned} & \textbf{class}\ Monoidal\ f\ \textbf{where} \\ & \quad unit :: f\ () \\ & \quad (\star) :: f\ a \to f\ b \to f\ (a, b) \end{aligned}$$

For instance, the function instance is

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoidal}\ ((\to)\ a)\ \textbf{where} \\ \quad \mathit{unit}\ = \mathit{const}\ () \\ \quad f \star g = \lambda x \to (f\ x, g\ x)\end{array}$$

The $\mathit{Applicative}$ class is equivalent to $\mathit{Functor}$+$\mathit{Monoidal}$ (McBride and Paterson 2008, Section 7). To get from $\mathit{Functor}$ and $\mathit{Monoidal}$ to $\mathit{Applicative}$, define

$$\begin{array}{l}\mathit{pure}\ a\ = \mathit{fmap}\ (\mathit{const}\ a)\ \mathit{unit} \\ \mathit{fs} \circledast \mathit{xs} = \mathit{fmap}\ \mathit{app}\ (\mathit{fs} \star \mathit{xs})\end{array}$$

where

$$\begin{array}{l}\mathit{app} :: (a \to b, a) \to b \\ \mathit{app}\ (f, x) = f\ x\end{array}$$

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

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

$$\begin{array}{l}\mathit{unit} \equiv \mathit{toD}_x\ \mathit{unit} \\ \mathit{toD}_x\ f \star \mathit{toD}_x\ g \equiv \mathit{toD}_x\ (f \star g)\end{array}$$

Filling in the definition of toD,

$$\begin{array}{l}\mathit{unit} \equiv D\ (\mathit{unit}\ x)\ (d\ \mathit{unit}\ x) \\ \quad D\ (f\ x)\ (d\ f\ x) \star D\ (g\ x)\ (d\ g\ x) \\ \equiv D\ ((f \star g)\ x)\ (d\ (f \star g)\ x)\end{array}$$

The reason for switching from $\mathit{Applicative}$ to $\mathit{Monoidal}$ is that differentiation is very simple with the latter:

$$\begin{array}{l}d\ \mathit{unit}\quad \equiv \mathit{const}\ 0 \\ d\ (f \star g) \equiv d\ f\ \hat{\star}\ d\ g\end{array}$$

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

$$(\hat{\star}) = \mathit{liftA}_2\ (\star)$$

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

Now simplify the morphism properties, using unit and $(\star)$ for functions, and their derivatives:

$$\begin{array}{l}\mathit{unit} \equiv D\ ()\ 0 \\ D\ (f\ x)\ (d\ f\ x) \star D\ (g\ x)\ (d\ g\ x) \\ \qquad \equiv D\ (f\ x, g\ x)\ (d\ f\ x \star d\ g\ x)\end{array}$$

So the following simple definitions suffice:

$$\begin{array}{l}\mathit{unit} = D\ ()\ 0 \\ D\ \mathit{fx}\ \mathit{dfx} \star D\ \mathit{gx}\ \mathit{dgx} = D\ (\mathit{fx}, \mathit{gx})\ (\mathit{dfx} \star \mathit{dgx})\end{array}$$

The switch from $\mathit{Applicative}$ to $\mathit{Monoidal}$ introduced $\mathit{fmap}\ \mathit{app}$ (in the definition of $(\circledast)$). Because of the meaning of fmap on $u \triangleright v$, we will need a derivative for app. Fortunately, 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 app at (f, x) is $d\ f\ x$. Allowing only f to vary, $f\ x$ is *linear* in f, so it (considered as a linear map) is its own derivative. That is, using $(\$)$ as infix function application,

$$\begin{array}{l}d\ (\$\, x)\ f \equiv \mathit{linear}\ (\$\, x) \\ d\ (f\, \$)\ x \equiv d\ f\ x\end{array}$$

As mentioned in Section 9, $(\diamond)$ takes the place of dot product for combining contributions from partial derivatives, so

$$d\ \mathit{app}\ (f, x) = \mathit{linear}\ (\$\, x) \diamond d\ f\ x$$

There is an alternative to using app. Recall that app was introduced in interpreting $(\circledast)$ from the $\mathit{Applicative}$ interface in terms of fmap and $(\star)$. The numeric instances only use $(\circledast)$ indirectly, via liftA_2. We can instead define liftA_2 more directly:

$$\mathit{liftA}_2\ h\ \mathit{as}\ \mathit{bs} = \mathit{fmap}\ (\mathit{uncurry}\ h)\ (\mathit{as} \star \mathit{bs})$$

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

$$\begin{array}{l}d\ (\mathit{uncurry}\ (+)) \equiv \mathit{dAdd} \\ d\ (\mathit{uncurry}\ (*)\ ) \equiv \mathit{dMul}\end{array}$$

where

$$\begin{array}{l}\mathit{dAdd}\ (x, y) = \mathit{id}_L \diamond \mathit{id}_L \\ \mathit{dMul}\ (x, y) = \mathit{linear}\ (*y) \diamond \mathit{linear}\ (x*)\end{array}$$

The usual derivative rules for $(+)$ and $(*)$ follow from the liftA_2-based definitions, and so needn't be programmed explicitly.

$$\begin{array}{l}\quad D\ x\ x' + D\ y\ y' \\ \equiv \mathit{liftA}\ (+)\ (D\ x\ x')\ (D\ y\ y') \\ \equiv \mathit{fmap}\ (\mathit{uncurry}\ (+))\ (D\ x\ x' \star D\ y\ y') \\ \equiv \mathit{fmap}\ (\mathit{uncurry}\ (+))\ (D\ (x, y)\ (x' \star y')) \\ \equiv D\ (\mathit{uncurry}\ (+)\ (x, y))\ (d\ (\mathit{uncurry}\ (+))\ (x, y) \odot (x' \star y')) \\ \equiv D\ (x + y)\ (d\ (\mathit{uncurry}\ (+))\ (x, y) \odot (x' \star y')) \\ \equiv D\ (x + y)\ ((\mathit{id}_L \diamond \mathit{id}_L) \odot (x' \star y')) \\ \equiv D\ (x + y)\ (x' + y')\end{array}$$

$$\begin{array}{l}\quad D\ x\ x' * D\ y\ y' \\ \equiv \mathit{liftA}\ (*)\ (D\ x\ x')\ (D\ y\ y') \\ \equiv \mathit{fmap}\ (\mathit{uncurry}\ (*))\ (D\ x\ x' \star D\ y\ y') \\ \equiv \mathit{fmap}\ (\mathit{uncurry}\ (*))\ (D\ (x, y)\ (x' \star y')) \\ \equiv D\ (\mathit{uncurry}\ (*)\ (x, y))\ (d\ (\mathit{uncurry}\ (*))\ (x, y) \odot (x' \star y')) \\ \equiv D\ (x * y)\ (d\ (\mathit{uncurry}\ (*))\ (x, y) \odot (x' \star y')) \\ \equiv D\ (x * y)\ ((\mathit{linear}\ (*y) \diamond \mathit{linear}\ (x*)) \odot (x' \star y')) \\ \equiv D\ (x * y)\ (x' * y + x * y')\end{array}$$

10.4 Fun with rules

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

$$\begin{array}{l}(*)\ \ = \mathit{liftA}_2\ (*) \\ \mathit{sin}\ \ = \mathit{fmap}\ \mathit{sin} \\ \mathit{sqrt} = \mathit{fmap}\ \mathit{sqrt} \\ \quad \ldots\end{array}$$

Section 10.2 made these definitions executable in spite of their appeal to the non-executable d by (a) refactoring fmap to split the d from the residual function $(\bowtie)$, (b) inlining fmap, and (c) rewriting applications of d with known derivative rules.

Now let's get the compiler to do these steps for us. Figure 6 list the derivatives of known functions as rewrite rules (Peyton Jones et al. 2001), minus rule names. Notice that these definitions are simpler and more modular than the standard differentiation rules, as they do not have the chain rule mixed in. For instance, compare (a) $d\ \mathit{sin} = \mathit{cos}$, (b) $d\ (\mathit{sin}\ u) = \mathit{cos}\ u * d\ u$, and (c) $d\ (\mathit{sin}\ u)\ x = \mathit{cos}\ u\ x * d\ u\ x$. With these rules in place, we can use the incredibly simple *fmap*-based definitions of our methods.

The definition of fmap must get inlined so as to reveal the d applications, which then get rewritten according to the rules. Fortunately, the fmap definition is tiny, which encourages its inlining. One could add an `INLINE` pragma for emphasis.

The current implementation of rewriting in GHC is somewhat fragile, so it may be a while before this sort of technique is robust enough for every day use.

```
{- RULES
```

$$
\begin{array}{ll}
d\ (uncurry\ (+)) & = dAdd \\
d\ (uncurry\ (*)) & = dMul \\
d\ negate & = -1 \\
d\ abs & = signum \\
d\ signum & = 0 \\
d\ recip & = -\ sqr\ recip \\
d\ exp & = exp \\
d\ log & = recip \\
d\ sqrt & = recip\ (2 * sqrt) \\
d\ sin & = cos \\
d\ cos & = -\ sin \\
d\ asin & = recip\ (sqrt\ (1 - sqr)) \\
d\ acos & = recip\ (-sqrt\ (1 - sqr)) \\
d\ atan & = recip\ (1 + sqr) \\
d\ sinh & = cosh \\
d\ cosh & = sinh \\
d\ asinh & = recip\ (sqrt\ (1 + sqr)) \\
d\ acosh & = recip\ (-sqrt\ (sqr - 1)) \\
d\ atanh & = recip\ (1 - sqr)
\end{array}
$$

```
-}
```

Figure 6. Derivatives as rewrite rules

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:

$$
\begin{array}{l}
\textbf{class}\ Functor\ s\ f\ \textbf{where} \\
\quad fmap :: (Vector\ s\ u, Vector\ s\ v) \Rightarrow \\
\qquad\qquad (u \to v) \to (f\ u \to f\ v) \\
\textbf{class}\ Functor\ s\ f \Rightarrow Monoidal\ s\ f\ \textbf{where} \\
\quad unit :: \ f\ () \\
\quad (\star)\ \ :: \ (Vector\ s\ u, Vector\ s\ v) \Rightarrow \\
\qquad\qquad f\ u \to f\ v \to f\ (u, v)
\end{array}
$$

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:

$$\textbf{data}\ u \hookrightarrow v = FD\ (u \to v)\ (u \to (u \multimap v))$$

This representation allows a simple and effective implementation of d:

$$
\begin{array}{l}
d :: (Vector\ s\ u, Vector\ s\ v) \Rightarrow \\
\quad (u \hookrightarrow v) \to (u \to (u \multimap v)) \\
d\ (FD\ \_\ f') = f'
\end{array}
$$

With these definitions, the simple numeric method definitions (via *fmap* and $liftA_2$) 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:

$$
\begin{array}{l}
\textbf{data}\ u \rhd^{*} v = D\ v\ (u \rhd^{*} (u \multimap v)) \\
toD :: (Vector\ s\ u, Vector\ s\ v) \Rightarrow u \to (u \to v) \to u \rhd^{*} v \\
toD_x\ f = D\ (f\ x)\ (toD_x\ (d\ f))
\end{array}
$$

The naturality (functor morphism) property is

$$fmap\ g \circ toD_x \equiv toD_x \circ fmap\ g$$

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:

$$
\begin{array}{l}
\quad fmap\ g\ (D\ (f\ x)\ (toD_x\ (d\ f))) \\
\equiv D\ ((fmap\ g\ f)\ x)\ (toD_x\ (d\ (fmap\ g\ f)))
\end{array}
$$

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

$$
\begin{array}{l}
\quad D\ ((g \circ f)\ x)\ (toD_x\ (d\ (g \circ f))) \\
\equiv \quad \{\ \text{Generalized chain rule}\ \} \\
\quad D\ (g\ (f\ x))\ (toD_x\ ((d\ g \circ f)\ \hat{\odot}\ d\ f)) \\
\equiv \quad \{\ liftA_2\ \text{morphism, } (\hat{\odot}) \equiv liftA_2\ (\odot)\ \} \\
\quad D\ (g\ (f\ x))\ (toD_x\ (d\ g \circ f)\ \hat{\odot}\ toD_x\ (d\ f))) \\
\equiv \quad \{\ fmap \equiv (\circ)\ \text{on functions}\ \} \\
\quad D\ (g\ (f\ x))\ (toD_x\ (fmap\ (d\ g)\ f)\ \hat{\odot}\ toD_x\ (d\ f))) \\
\equiv \quad \{\ fmap\ \text{morphism}\ \} \\
\quad D\ (g\ (f\ x))\ (fmap\ (d\ g)\ (toD_x\ f)\ \hat{\odot}\ toD_x\ (d\ f))
\end{array}
$$

Summarizing, toD_x is a *Functor* morphism iff

$$
\begin{array}{l}
\quad fmap\ g\ (D\ (f\ x)\ (toD_x\ (d\ f))) \\
\equiv D\ (g\ (f\ x))\ (fmap\ (d\ g)\ (toD_x\ f)\ \hat{\odot}\ toD_x\ (d\ f))
\end{array}
$$

Given this form of the morphism condition, and recalling the definition of toD, it's easy to find a *fmap* definition for $(\rhd^{*})$:

$$fmap\ g\ fxs@(D\ fx\ dfx) = D\ (g\ fx)\ (fmap\ (d\ g)\ fxs\ \hat{\odot}\ dfx)$$

The only change from $(\rhd)$ is $(\hat{\odot})$ in place of $(\odot)$.

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 $(u \hookrightarrow v)$, 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 $(\rhd)$ above. The only difference is using $(\hat{\star})$ in place of $(\star)$.

$$
\begin{array}{l}
\textbf{instance}\ Vector\ s\ u \Rightarrow Monoidal\ s\ ((\rhd^{*})\ u)\ \textbf{where} \\
\quad unit = D\ ()\ 0 \\
\quad D\ u\ u' \star D\ v\ v' = D\ (u, v)\ (u'\ \hat{\star}\ v')
\end{array}
$$

To optimize out zeros in either $u \rhd v$ or $u \rhd^{*} v$, 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 $(\star)$. To handle the *Nothing* vs *Just*, add an extra *fmap* in the *fmap* definition, and add another $liftA_2$ in the $(\star)$ definition. For instance,

$$
\begin{array}{l}
\textbf{data}\ u \rhd^{*} v = D\ v\ (Maybe\ (u \rhd^{*} (u \multimap v))) \\
fmap\ g\ fxs@(D\ fx\ dfx) = \\
\quad D\ (g\ fx)\ (fmap\ (fmap\ (d\ g)\ fxs\ \hat{\odot})\ dfx)
\end{array}
$$

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 cluttering 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 mutiplications 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 $O(n^2)$ 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 $g \circ f$, 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 strucures stitched together out of simpler pieces (ultimately vector spaces). Explore differentiation in the setting of these rich structures.

Efficiency analysis. Forward-mode AD for $\mathbb{R}^n \to \mathbb{R}$ 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

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton-Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253. ACM Press, 2005.

Conal Elliott. Denotational design with type class morphisms. Technical Report 2009-01, LambdaPix, March 2009a. URL `http://conal.net/papers/type-class-morphisms`.

Conal Elliott. Push-pull functional reactive programming. In *Proceedings of the Haskell Symposium*, 2009b.

Jason Foutz. Higher order multivariate automatic differentiation in haskell. Blog post, February 2008. URL `http://metavar.blogspot.com/2008/02/higher-order-multivariate-automatic.html`.

Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(04):327–351, 2000.

Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79(4):197–201, 2001.

Jerzy Karczmarczuk. Functional coding of differential forms. In *Scottish Workshop on Functional Programming*, 1999.

Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1), 2001.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

M. Douglas McIlroy. The music of streams. *Information Processing Letters*, 77(2-4):189–195, 2001.

Henrik Nilsson. Functional automatic differentiation with Dirac impulses. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 153–164, Uppsala, Sweden, August 2003. ACM Press.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 2007 Symposium on Principles of Programming Languages*, pages 155–60, Nice, France, January 2007.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *In Haskell Workshop*. ACM SIGPLAN, 2001.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In *Implementation and Application of Functional Languages, IFL'05*, September 2005.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher Order Symbolic Computation*, 21(4):361–376, 2008.

Michael Spivak. *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus*. HarperCollins Publishers, 1971.

R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463 – 464, 1964.

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), $Basis\ v$, and a function that interprets a basis representation as a vector. Another method extracts coordinates (coefficients) for a vector with respect to basis elements.

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

The $Enumerable$ 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:

```
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 $Void$ 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.

```
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 $Left$ or $Right$. The vectors corresponding to these basis elements are $(ub, 0)$ or $(0, vb)$, where ub corresponds to a basis element for u, and vb 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.

```
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
```

Triples etc, can be handled similarly or reduced to nested pairs.

Basis types are usually finite and small, so the decompositions can be memoized for efficiency, e.g., using memo tries (Elliott 2009a).

A.2 Linear maps

Semantically, a *linear map* is a function $f :: a \rightarrow b$ such that, for all scalar values s and "vectors" $u, v :: a$, the following properties hold:

$$f\ (s\ \cdot\ u) \equiv s \cdot f\ u$$
$$f\ (u + v) \equiv f\ u + f\ v$$

By repeated application of these properties,

$$f\ (s_1 \cdot u_1 + \ldots + s_n \cdot u_n) \equiv s_1 \cdot f\ u_1 + \ldots + s_n \cdot f\ u_n$$

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 $(u \multimap v)$ to $(u \rightarrow v)$. 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 u = sumV [coord u e · uv 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 $coord$ method can be changed to return $v \multimap s$, 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 $lapply\ lm$ can apply f to the value of every basis element, and then linearly combine results.

A simple trick fixes this efficiency problem: *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 $lapply\ lm$ 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).

$$\textbf{type}\ u \multimap v = Basis\ u \xrightarrow{M} v$$

The conversion functions $linear$ and $lapply$ need just a little tweaking. Split $memo$ into its definition $untrie \circ trie$, and then move $untrie$ into $lapply$. We'll also have to add $HasTrie$ constraints:

```
$linear :: (\,Vector\ s\ u, Vector\ s\ v$
          $, HasBasis\ s\ u, HasTrie\ (Basis\ u)) \Rightarrow$
          $(u \to v) \to (u \multimap v)$
$linear\ f = trie\ (f \circ basisValue)$
$lapply :: (\,Vector\ s\ u, Vector\ s\ v$
          $, HasBasis\ s\ u, HasTrie\ (Basis\ u)) \Rightarrow$
          $(u \multimap v) \to (u \to v)$
$lapply\ lm =$
  $linearCombo \circ fmap\ (first\ (untrie\ lm)) \circ decompose$
```

Now we can build up linear maps conveniently and efficiently by using the $Functor$ and $Applicative$ 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 $liftA_2\ h\ m\ n$ is the linear map that applies h to the results of m and n.

$$lapply\ (liftA_2\ h\ m\ n)\ a = h\ (lapply\ m\ a)\ (lapply\ n\ a)$$

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

$$lapply\ (liftA_2\ h\ m\ n) = liftA_2\ h\ (lapply\ m)\ (lapply\ n)$$

In other words, the meaning of a $liftA_2$ is the $liftA_2$ of the meanings.