Beautiful differentiation

Conal Elliott

LambdaPix

1 September, 2009  ICFP
Differentiation
Derivatives have many uses.

For instance,
- optimization
- root-finding
- surface normals
- curve and surface tessellation
There are three common differentiation techniques.

- Numeric
- Symbolic
- "Automatic" (*forward & reverse modes*)
What’s a derivative?

For scalar domain:

\[ d :: \text{Scalar } s \Rightarrow (s \rightarrow s) \rightarrow (s \rightarrow s) \]

\[ d \ f \ x = \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} \]
What’s a derivative?

For scalar domain:

\[ d :: \text{Scalar } s \Rightarrow (s \to s) \to (s \to s) \]

\[ d f x = \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} \]

What about non-scalar domains?
Return to this question later.
Aside: We can treat functions like numbers.

\[
\text{instance } \text{Num } \beta \Rightarrow \text{Num } (\alpha \rightarrow \beta) \text{ where}
\]
\[
u + v = \lambda x \rightarrow u \ x + v \ x
\]
\[
u \ast v = \lambda x \rightarrow u \ x \ast v \ x
\]
\[
\ldots
\]

\text{instance } \text{Floating } \beta \Rightarrow \text{Floating } (\alpha \rightarrow \beta) \text{ where}

\[
sin \ u = \lambda x \rightarrow \sin (u \ x)
\]
\[
cos \ u = \lambda x \rightarrow \cos (u \ x)
\]
\[
\ldots
\]
We can treat applicatives like numbers.

```
instance Num β ⇒ Num (α → β) where
  (+) = liftA2 (+)
  (∗) = liftA2 (∗)
  ...

instance Floating β ⇒ Floating (α → β) where
  sin = fmap sin
  cos = fmap cos
  ...
```
What is automatic differentiation?

- Computes function & derivative values in tandem
- “Exact” method
- Numeric, not symbolic
Scalar, first-order AD

Overload functions to work on function/derivative value pairs:

```haskell
data D α = D α α
```

For instance,

```
D a a' + D b b' = D (a + b) (a' + b')
D a a' * D b b' = D (a * b) (b' * a + a' * b)
sin (D a a') = D (sin a) (a' * cos a)
sqrt (D a a') = D (sqrt a) (a' / (2 * sqrt a))
...```

Are these definitions correct?

Conal Elliott (LambdaPix)

Beautiful differentiation 1 September, 2009 ICFP 9 / 32
Scalar, first-order AD

Overload functions to work on function/derivative value pairs:

```hs
data D α = D α α
```

For instance,

\[
D a a' + D b b' = D (a + b) (a' + b') \\
D a a' * D b b' = D (a * b) (b' * a + a' * b) \\
sin (D a a') = D (\sin a) (a' * \cos a) \\
sqrt (D a a') = D (\sqrt a) (a' / (2 * \sqrt a)) \\
\ldots
\]

Are these definitions correct?
What is automatic differentiation — really?

- What does AD mean?
- How does a correct implementation arise?
- Where else might these answers take us?
What does AD mean?
**What does AD mean?**

```haskell
data D α = D α α

toD :: (α → α) → (α → D α)
toD f = λx → D (f x) (d f x)

Spec: toD combinations correspond to function combinations, e.g.,

toD u + toD v ≡ toD (u + v)
toD u * toD v ≡ toD (u * v)
recip (toD u) ≡ toD (recip u)
sin (toD u) ≡ toD (sin u)
cos (toD u) ≡ toD (cos u)

I.e., toD preserves structure.
```
How does a correct implementation arise?
How does a correct implementation arise?

Goal: \( \forall u. \, \sin (toD u) \equiv toD (\sin u) \)
How does a correct implementation arise?

Goal: \( \forall u. \sin (\text{toD } u) \equiv \text{toD } (\sin u) \)

Simplify each side:

\[
\sin (\text{toD } u) \equiv \lambda x \rightarrow \sin (\text{toD } u x) \\
\equiv \lambda x \rightarrow \sin (D (u x) (d u x))
\]

\[
\text{toD } (\sin u) \equiv \lambda x \rightarrow D (\sin u x) (d (\sin u) x) \\
\equiv \lambda x \rightarrow D ((\sin \circ u) x) ((d u \ast \cos u) x) \\
\equiv \lambda x \rightarrow D (\sin (u x)) (d u x \ast \cos (u x))
\]
How does a correct implementation arise?

Goal: \( \forall u. \sin (\text{toD } u) \equiv \text{toD } (\sin u) \)

Simplify each side:

\[
\begin{align*}
\sin (\text{toD } u) & \equiv \lambda x \rightarrow \sin (\text{toD } u \ x) \\
& \equiv \lambda x \rightarrow \sin (D (u \ x) (d \ u \ x))
\end{align*}
\]

\[
\begin{align*}
\text{toD } (\sin u) & \equiv \lambda x \rightarrow D (\sin u \ x) (d (\sin u) \ x) \\
& \equiv \lambda x \rightarrow D ((\sin \circ u) \ x) ((d \ u \ast \cos u) \ x) \\
& \equiv \lambda x \rightarrow D (\sin (u \ x)) (d \ u \ x \ast \cos (u \ x))
\end{align*}
\]

Sufficient:

\[
\begin{align*}
\sin (D \ ux \ dux) & = D (\sin ux) (dux \ast \cos ux)
\end{align*}
\]
Where else might these answers take us?
Where else might these answers take us?

In this talk

- Prettier definitions
- Higher-order derivatives
- Higher-dimensional functions
Digging deeper — the scalar chain rule

\[ d (g \circ u) x \equiv d g (u x) \ast d u x \]

For scalar domain & range. Variations for other dimensions.

Define and reuse:

\[ (g \triangleleft dg) (D ux dux) = D (g ux) (dg ux \ast dux) \]

For instance,

\[
\begin{align*}
  \sin &= \sin \triangleleft \cos \\
  \cos &= \cos \triangleleft \lambda x \rightarrow -\sin x \\
  \sqrt{x} &= \sqrt{x} \triangleleft \lambda x \rightarrow \text{recip} (2 \ast \sqrt{x})
\end{align*}
\]
Function overloadings make for prettier definitions.

```haskell
instance Floating α ⇒ Floating (D α) where
  exp  = exp △ exp
  log  = log △ recip
  sqrt = sqrt △ recip (2 * sqrt)
  sin  = sin △ cos
  cos  = cos △ −sin
  acos = acos △ recip (−sqrt (1 − sqr))
  atan = atan △ recip (1 + sqr)
  sinh = sinh △ cosh
  cosh = cosh △ sinh

sqr x = x * x
```
Scalar, higher-order AD

Generate *infinite towers* of derivatives (Karczmarczuk 1998):

\[
\textbf{data } D \alpha = D \alpha (D \alpha)
\]

Suffices to tweak the chain rule:

\[
(g \bowtie dg) \ (D \ ux_0 \ dux) = D (g \ ux_0) (dg \ ux_0 \ast dux) \quad \text{-- old}
\]

\[
(g \bowtie dg) \ ux_0@(D \ ux_0 \ dux) = D (g \ ux_0) (dg \ ux \ast dux) \quad \text{-- new}
\]

Most other definitions can then go through unchanged.
The derivations adapt.
What’s a derivative – really?

For scalar domain:

\[
d f x = \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - f x}{\varepsilon}
\]
What’s a derivative – really?

For scalar domain:

\[ d f x = \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} \]

Redefine: unique scalar \( s \) such that

\[ \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} - s \equiv 0 \]
What’s a derivative – really?

For scalar domain:

\[ d f(x) = \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} \]

Redefine: unique scalar \( s \) such that

\[ \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - f(x) - s \cdot \varepsilon}{\varepsilon} = 0 \]

Equivalently,

\[ \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - f(x) - s \cdot \varepsilon}{\varepsilon} = 0 \]

or

\[ \lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - (f(x) + s \cdot \varepsilon)}{\varepsilon} = 0 \]
What’s a derivative – really?

\[
\lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - (f(x) + s \cdot \varepsilon)}{\varepsilon} \equiv 0
\]

Derivatives are linear maps. Captures all "partial derivatives" for all dimensions. See Calculus on Manifolds by Michael Spivak.
What’s a derivative – really?

$$\lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - (f(x) + s \cdot \varepsilon)}{\varepsilon} \equiv 0$$

Now generalize: unique \textit{linear map} $T$ such that:

$$\lim_{\varepsilon \to 0} \frac{|f(x + \varepsilon) - (f(x) + T \varepsilon)|}{|\varepsilon|} \equiv 0$$
What’s a derivative – really?

\[
\lim_{\varepsilon \to 0} \frac{f(x + \varepsilon) - (f x + s \cdot \varepsilon)}{\varepsilon} \equiv 0
\]

Now generalize: unique linear map \( T \) such that:

\[
\lim_{\varepsilon \to 0} \frac{|f(x + \varepsilon) - (f x + T \varepsilon)|}{|\varepsilon|} \equiv 0
\]

Derivatives are linear maps.

Captures all “partial derivatives” for all dimensions. See *Calculus on Manifolds* by Michael Spivak.
The chain rules all unify into one.

Generalize from

\[ d (g \circ u) \equiv d g (u \times) \ast d u \times \]

etc
The chain rules all unify into one.

Generalize from

\[ d \left( g \circ u \right) x \equiv d g \left( u x \right) * d u x \]

etc to

\[ d \left( g \circ u \right) x \equiv d g \left( u x \right) \circ d u x \]
Higher-dimensional functions

Generalized derivatives

Derivative values are linear maps: $\alpha \rightarrow \beta$.

$$d :: (\text{Vector } s \alpha, \text{Vector } s \beta) \Rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow (\alpha \rightarrow \beta))$$

First-order AD:

**data** $\alpha \triangleright \beta = D \beta (\alpha \rightarrow \beta)$

Higher-order AD:

**data** $\alpha \triangleright^* \beta = D \beta (\alpha \triangleright^* (\alpha \rightarrow \beta))$

$\approx \beta \times (\alpha \rightarrow \beta) \times (\alpha \rightarrow (\alpha \rightarrow \beta)) \times \ldots$
What’s a linear map?

Preserves linear combinations:

\[ h (s_1 \cdot u_1 + \ldots + s_n \cdot u_n) \equiv s_1 \cdot h u_1 + \ldots + s_n \cdot h u_n \]
What’s a linear map?

Preserves linear combinations:

\[ h(s_1 \cdot u_1 + \ldots + s_n \cdot u_n) \equiv s_1 \cdot h u_1 + \ldots + s_n \cdot h u_n \]

Fully determined by behavior on basis of \( \alpha \), so

\textbf{type} \( \alpha \to \beta = Basis \alpha \xrightarrow{M} \beta \)

Memoized for efficiency.
What’s a linear map?

Preserves linear combinations:

\[ h (s_1 \cdot u_1 + \ldots + s_n \cdot u_n) \equiv s_1 \cdot h u_1 + \ldots + s_n \cdot h u_n \]

Fully determined by behavior on basis of \( \alpha \), so

\[
\text{type } \alpha \rightarrow \beta = \text{Basis } \alpha \xrightarrow{M} \beta
\]

Memoized for efficiency.

Vectors, matrices, etc re-emerge as memo-tries. Statically dimension-typed!
What's a basis?

class Vector s v ⇒ HasBasis s v where
  type Basis v :: *
  coord :: v → (Basis v → s)
  basisValue :: Basis v → v
instance HasBasis Double Double where
  type Basis Double = ()
  coord s = λ() → s
  basisValue () = 1

instance (HasBasis s u, HasBasis s v)
  ⇒ HasBasis s (u, v) where
  type Basis (u, v) = Basis u 'Either' Basis v
  coord (u, v) = coord u 'either' coord v
  basisValue (Left a) = (basisValue a, 0)
  basisValue (Right b) = (0, basisValue b)
Automatic differentiation – naturally
Can we make AD even simpler?

Recall our function overloadings:

```haskell
instance Num β ⇒ Num (α → β) where
  (+) = liftA2 (+)
  (∗) = liftA2 (∗)
  ...

instance Floating β ⇒ Floating (α → β) where
  sin = fmap sin
  cos = fmap cos
  ...
```

These definitions are standard for applicative functors. Could they work for $D$?
Could we simply define AD via the standard

\[ \sin = \text{fmap} \sin \]

etc? What is \text{fmap}? Require \( \text{toD}_x \) be a \textit{natural transformation}:

\[ \text{fmap} g \circ \text{toD}_x \equiv \text{toD}_x \circ \text{fmap} g \]

where

\[ \text{toD}_x \ u = D (u \ x) (d \ u \ x) \]

Define \text{fmap} from this naturality condition.
Derive AD \textit{naturally}

\[
\text{toD}_x (fmap \ g \ u) \equiv \text{toD}_x (g \circ u) \\
\equiv D ((g \circ u) \ x) \ (d \ (g \circ u) \ x) \\
\equiv D (g \ (u \ x)) \ (d \ g \ (u \ x) \circ d \ u \ x)
\]

\[
fmap g \ (\text{toD}_x u) \equiv fmap g \ (D \ (u \ x) \ (d \ u \ x))
\]

Sufficient definition:

\[
fmap g \ (D \ ux \ dux) = D \ (g \ ux) \ (d \ g \ ux \circ dux)
\]

Similar derivation for \textit{liftA}_2 (for (+), (\ast), etc).
Sufficient definition:

\[ \text{fmap } g \ (D \ ux \ dux) = D \ (g \ ux) \ (d \ g \ ux \circ \ dux) \]

Oops. \( d \) doesn't have an implementation.

Solution A: Inline \textit{fmap} for each \textit{fmap }\( g \) and rewrite \( d \ g \) to known derivative.

Solution B: Generalize \textit{Functor} to allow non-function arrows, and replace functions by differentiable functions.
Conclusions

- Specification as a *structure-preserving semantic function*.
- Implementation *derived systematically* from specification.
- Prettier implementation via *functions-as-numbers*.
- *Infinite derivative towers* with nearly no extra code.
- Generalize to differentiation over *vector spaces*.
- Even simpler specification/derivation via *naturality*.