Functional Differentiation of Computer Programs
by Jerzy Karczmarczuk

Henning Zimmer

March 22, 2006

Outline

Why do we want to compute derivatives ?

Derivatives are useful for ...

- solving Optimization Problems
- Image Processing (Feature Extraction, Object Recognition)
- 3-D-Modelling (geom. properties of curves and surfaces)
- Many fields of scientific computing like engineering, . . .

Why do we want to compute derivatives ?

Derivatives are useful for ...

- solving Optimization Problems
- Image Processing (Feature Extraction, Object Recognition)
- 3-D-Modelling (geom. properties of curves and surfaces)
- Many fields of scientific computing like engineering, . . .

We show a

- purely functional implementation (using Haskell)
- only based on numerics (no symbolic computations)
- relying on overloading of arithmetic operators,
  lazy evaluation and type classes concept
- yielding (point-wise) derivatives of ..
- .. *any* order, using *'co-recursive'* data structures and
- .. *any* mathematical function definable in Haskell code

Outline

3 ways ... (I)

We have 3 ways to compute derivatives:

1. Finite differences approximation:

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

- *Inaccurate* if Δx is too big,
- *Cancellation errors* if Δx is too small.

3 ways ... (I)

We have 3 ways to compute derivatives:

1. Finite differences approximation:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

   - *Inaccurate* if Δx is too big,
   - *Cancellation errors* if Δx is too small.

2. Symbolic differentiation: 'manual', formal method
   - Exact, but *quite costly*
   - Control structures like loops, etc. have to be 'unfolded' $\rightsquigarrow$ symbolic interpretation of whole program

3 ways ... (II)

3. **Computational Differentiation - CD**: Our approach !
   - Numeric algorithms, based on standard arithmetic operations, with known differential properties (school knowledge!)
   - As *exact* as numerical evaluation of symbolic derivatives (but lacks symbolical (analytical) results)
     based on overloading (already implemented in C++)
   - Functional implementation relies on co-recursive data structures

     $$R\ \alpha = C\ \alpha \mid T\ \alpha\ (R\ \alpha)$$

     for computing derivatives of *any* order!
   - **Drawback**: discontinuous or non-differentiable functions (e.g. `abs x`) also yield values for their derivatives, which is unsatisfactory

Outline

First approach: 'We are not lazy!'

We start with a simple approach

- only compute first derivatives
- *without* lazy evaluation
- yielding a quite efficient solution
- introduce 'extended numerical' structure:

```
type Dx = (Double, Double)
```

First approach: 'We are not lazy!'

We start with a simple approach

- only compute first derivatives
- *without* lazy evaluation
- yielding a quite efficient solution
- introduce 'extended numerical' structure:

```
type Dx = (Double, Double)
```

- grouping numerical value (main value) e of an expression with value of first derivative e' *at the same point*: (e, e')
- `(c, 0.0)` for constants c and `(x, 1.0)` for variables x.
- Could replace `double` by any ring $(R, +, \times)$ or field $(F, +, \times, /)$
- Remark: No symbolic calculations $\rightsquigarrow$ constants and variables don't need to have explicit names !
  e.g.: `(3.141, 0.0)` or `(2.523, 1.0)`

Overloaded Arithmetic

- Define overloaded arithmetic operators for type `Dx`
- implementing basic derivation laws
  sum-, product-, quotient-rule, ...

Overloaded Arithmetic

- Define overloaded arithmetic operators for type `Dx`
- implementing basic derivation laws
  sum-, product-, quotient-rule, ...

```
(x,a)+(y,b) = (x+y, a+b) (:: Dx -> Dx -> Dx)
(x,a)-(y,b) = (x-y, a-b)
(x,a)*(y,b) = (x*y, x*b+a*y)
negate (x,a) = (negate x, negate a)
(x,a)/(y,b) = (x/y, (a*y-x*b/(y*y))
recip (x,a) = (w,(negate a)*w*w) where w=recip x
```

Overloaded Arithmetic

- Define overloaded arithmetic operators for type `Dx`
- implementing basic derivation laws
  sum-, product-, quotient-rule, ...

```
(x,a)+(y,b) = (x+y, a+b) (:: Dx -> Dx -> Dx)
(x,a)-(y,b) = (x-y, a-b)
(x,a)*(y,b) = (x*y, x*b+a*y)
negate (x,a) = (negate x, negate a)
(x,a)/(y,b) = (x/y, (a*y-x*b/(y*y))
recip (x,a) = (w,(negate a)*w*w) where w=recip x
```

- Also auxiliary functions to construct constants and variables and a conversion function

```
dCst z = (z, 0.0)    dVar z = (z, 1.0)
fromDouble z = dCst z
```

Haven’t we forgot something?

Haven't we forgot something?

- Chain rule: $d(f(g(x))) = f'(g(x)) \cdot d(g(x))$
- Important for derivatives of elementary functions like $\sin, \cos, \log, \ldots$
- These functions `f` are lifted to the `Dx` domain, given their *derivative form* `f'`

```
dlift f f' (x,a) = (f x , a * f' x)
exp = dlift exp exp
sin = dlift sin cos
```

- .. same for $cos, \sqrt{x}, log$
- Now we can define arbitrary complicated mathematical functions like `f x = x*x * cos(x)`
- .. and `f 6.5` $\rightsquigarrow$ `(41.260827, 3.606820)` $\equiv (f(6.5), f'(6.5))$

Haskell type classes

- Approach doesn't use Haskell's type classes [1]
- Introduce modified algebraic style library ($\equiv$ mathematical hierarchy) of type classes:

[1] generic operations: declared within classes, datatypes accepting them are instances of them

Haskell type classes

- Approach doesn't use Haskell's type classes [1]
- Introduce modified algebraic style library ($\equiv$ mathematical hierarchy) of type classes:

- `AddGroup` for addition and subtraction
- `Monoid` for multiplication, `Group` for division
- `Ring` for *structures* supporting addition and multiplication, `Field` adding division
- `Module` abstracts multiplication of complex object by element of basic domain (e.g.: $\lambda \cdot \vec{v}$)
- `Number` uses `fromInt, fromDouble` to convert standard numbers in our `Dx` domain

[1] generic operations: declared within classes, datatypes accepting them are instances of them

Outline

Differential Algebra and 'Lazy towers of derivatives'

- Compute (as promised) *'all'* derivatives of functions (exact: an *a priori* unknown number)
- Data structure, representing expression of infinite domain: num. value e_0 and all derivatives $[e_0, e_1, e_2, \ldots]$ ($e_i \equiv e^{(i)}$) without explicit truncation, created by co-recursion!

Differential Algebra and 'Lazy towers of derivatives'

- Compute (as promised) *'all'* derivatives of functions (exact: an *a priori* unknown number)
- Data structure, representing expression of infinite domain: num. value e_0 and all derivatives $[e_0, e_1, e_2, \ldots]$ ($e_i \equiv e^{(i)}$) without explicit truncation, created by co-recursion!
- Need background in Differential Algebra
- Field $(F, +, \times, /)$ with derivation $a \mapsto a'$
- $F = \mathbb{R}$ is trivial: $\forall x \in \mathbb{R} : x \mapsto 0$
- Extend field to $F(x)$ by adjoining symbolic x
- If mathematical structure of the expressions known, we can discard the x $\rightsquigarrow$ no symbolic computations
- E.g.: Represent polynomial by list of its coefficients

Get it started

- Important: We assume that x and x' are algebraic independent and thus assign to expressions e all derivatives $e', e'', \ldots$ by the derivation operator $e_n \mapsto e_{n+1}$
- We use no indeterminate and just operate on infinite, lazy lists of a priori independent elements
- We define the *co-recursive*, infinite, parameterized type

```
data Dif a = C a | D a (Dif a)
```

Get it started

- Important: We assume that x and x' are algebraic independent and thus assign to expressions e all derivatives $e', e'', \ldots$ by the derivation operator $e_n \mapsto e_{n+1}$
- We use no indeterminate and just operate on infinite, lazy lists of a priori independent elements
- We define the *co-recursive*, infinite, parameterized type

```
data Dif a = C a | D a (Dif a)
```

- `C a` codes a constant a whose derivative is 0
- `D e (D a (D b ...))` codes the numerical value of the expression (e) and the remainder the tower of derivatives ($a = e', b = e'', \ldots$)
- In general, a should be an instance of a field, e.g. `Double`

Overloaded Arithmetics for Dif domain

- The derivation operator `df :: a -> a` is declared in `class Diff a`
- *Lifting* procedures: **`df (C _) = C 0.0 ; df (D _ p) = p`**
- We implement the basic derivation laws
- The sum-rule is trivial, with `Dif` a instance of `AddGroup` class:

```
C x + C y = C (x+y)
C x + D y y' = D (x+y) y'
D x x' + D y y' = D (x+y) (x'+y')
neg = fmap neg
```

[2] **`x*>s = fmap (x*) s`**

Overloaded Arithmetics for Dif domain

- The derivation operator `df :: a -> a` is declared in `class Diff a`
- *Lifting* procedures: **`df (C _) = C 0.0 ; df (D _ p) = p`**
- We implement the basic derivation laws
- The sum-rule is trivial, with `Dif` a instance of `AddGroup` class:

```
C x + C y = C (x+y)
C x + D y y' = D (x+y) y'
D x x' + D y y' = D (x+y) (x'+y')
neg = fmap neg
```

- Same for product-rule and unaltered constants (`Monoid` class):

```
C x * C y = C (x*y)
C x * p = x*>p
p@(D x x')*q@(D y y') = D (x*y)(x'*q+p*y')
```
[2]

[2] `x*>s = fmap (x*) s`

Overloaded Arithmetics (II)

- **Reciprocal** $(\frac{1}{u(x)})' = \frac{-u'(x)}{u(x)^2}$ heavily uses *lazy evaluation* (`Group` class):

```
recip (C x) = C (recip x)
recip (D x x') = ip where
  ip = D (recip x) (neg x'*ip*ip)
```

- further trivial cases left out !

Overloaded Arithmetics (II)

- **Reciprocal** $(\frac{1}{u(x)})' = \frac{-u'(x)}{u(x)^2}$ heavily uses *lazy evaluation* (`Group` class):

```
recip (C x) = C (recip x)
recip (D x x') = ip where
  ip = D (recip x) (neg x'*ip*ip)
```

- further trivial cases left out !

- **Division** might present some problems: $\frac{0}{0}$

```
p@(D x x') / q@(D y y')
   | x==0.0 && y==0.0 = x'/y' --L' Hopital--
   | otherwise = D (x/y) (x'*q - p*y'/(q*q))
```

Lifting and the chain rule

- Transcendental functions `f` like exp, sin, ... need *lifting* to the `Dif` domain
- Definition of their list of formal derivatives `fq`, using *lazy evaluation* (`Group` class)
- E.g.: $(\exp(u(x)))' = u'(x) \cdot \exp(u(x))$

```
dlift (f:fq) p@(D x x') =
  D (f x) (x' * dlift fq p) {--Chain rule--}

exp (D x x') = r where r = D (exp x) (x'*r)

sin =
 dlift (cycle[sin,cos,(neg . sin),(neg . cos)])
```

- $cos, log, \sqrt{x}$ in the same manner!

Lifting and the chain rule

- Transcendental functions `f` like exp, sin, ... need *lifting* to the `Dif` domain
- Definition of their list of formal derivatives `fq`, using *lazy evaluation* (`Group` class)
- E.g.: $(\exp(u(x)))' = u'(x) \cdot \exp(u(x))$

```
dlift (f:fq) p@(D x x') =
  D (f x) (x' * dlift fq p) {--Chain rule--}

exp (D x x') = r where r = D (exp x) (x'*r)

sin =
 dlift (cycle[sin,cos,(neg . sin),(neg . cos)])
```

- $cos, log, \sqrt{x}$ in the same manner!
- and that's it ... **we're done !!!**
- Now: `df (df (df (f 6.5)))` $\rightsquigarrow$ `-30.288818` $\equiv f'''(6.5)$

Outline

Example applications

- Wide spread, huge application domain, ’ranging from reactor diagnostic, meteorology, oceanography, up to biostatistics’ and quantum theory

Example applications

- Wide spread, huge application domain, 'ranging from reactor diagnostic, meteorology, oceanography, up to biostatistics' and quantum theory
- **One example**: Elegant coding of differential recurrences, like the *Hermite function*, without explicit truncation of recurrent computation !

$$
\begin{aligned}
H_0(x) &= exp(\frac{-x^2}{2}) \\
H_n(x) &= \frac{1}{\sqrt{2n}}(x \cdot H_{n-1}(x) - \frac{d}{dx}(H_{n-1}(x)))
\end{aligned}
$$

```
herm n x = cc where
  D cc _ = hr n (dVar x)
  hr 0 x = exp(neg x * x / fromDouble 2.0)
  hr n x = (x*z - df z)/(sqrt(fromInteger (2*n)))
    where z=hr (n-1) x
```

Outline

Final Remarks - Pro's and Con's

- clear, readable, compact (especially for towers!) and semantically powerful $\rightsquigarrow$ ***nice coding tool!***
- *Thunks* of lazy evaluation may introduce space leaks, when computing derivatives of high order
  Remedy: use truncated strict variant, like 1st approach, given number of derivatives to compute
- not extremely efficient, hence outperformed by C++ implementations and semi-automatic systems
- Still useable and faster than symbolic systems

Final Remarks - Pro's and Con's

- clear, readable, compact (especially for towers!) and semantically powerful $\rightsquigarrow$ ***nice coding tool!***
- *Thunks* of lazy evaluation may introduce space leaks, when computing derivatives of high order
  Remedy: use truncated strict variant, like 1st approach, given number of derivatives to compute
- not extremely efficient, hence outperformed by C++ implementations and semi-automatic systems
- Still useable and faster than symbolic systems
- *Claim*: straight forward generalization to vector or tensor objects
- Control structures (*if-then-else*) need arithm. relations on (infinite) `Dif` type
  Simplified remedy: just compare main values

Summary

- We've seen: Rewarding application of modern functional programming paradigms to scientific computing (usually domain of low-level languages)

Contribution

Summary

- We've seen: Rewarding application of modern functional programming paradigms to scientific computing (usually domain of low-level languages)

Contribution

- Type inference, Overloading $\Rightarrow$ overloaded arithmetic operators, declare differentiation *variables*
- Lazy evaluation $\Rightarrow$ derivation operator, applicable arbitrary (*a priori* unknown) number of times, without explicit truncation!
- Type classes, Lifting $\Rightarrow$ extended arithmetics, valid for any *basic domain*, e.g.: $\mathbb{C}, \mathbb{P}$

Outline

References

- *Karczmarczuk, Jerzy,* **Functional Differentiation of Computer Programs,**
  Journal of HOSC (14), (2001), pp. 35-57
- *Karczmarczuk, Jerzy,* **Generating power of lazy semantics,**
  Journal of Theoretical Computer Science (vol. 187), (1997), pp. 203-219