

DSLsofMath 2021: Assignment 2

Optimisation using Newton's method

This assignment is based on the lectures from weeks 3 and 4 (the *FunExp* type, *eval*, *derive*, *D*, tupling, homomorphisms, *FD*, *apply*, ...) so it pays off to work through those notes carefully.

1. The evaluation of the second derivative is given by

$$eval'' = eval' \circ derive = eval \circ derive \circ derive$$

- (a) Show that *eval''* is not a homomorphism from *FunExp* to *FunSem* = $\mathbb{R} \rightarrow \mathbb{R}$.
- (b) Given the following types

```
type Tri a      = (a, a, a)
type TriFun a = Tri (a → a)  -- = (a → a, a → a, a → a)
type FunTri a = a → Tri a    -- = a → (a, a, a)
```

Define instances of the classes *Additive*, *AddGroup*, *Multiplicative*, *MulGroup*, *Algebraic*, and *Transcendental*, for *Tri a* and define a homomorphism *evalDD* from *FunExp* to *FunTri a* (for any type *a* in *Field*). You don't need to prove that it is a homomorphism in this part.

- (c) Show that *evalDD* is a homomorphism for the case of multiplication.
2. Newton's method allows us to find zeros of a large class of functions in a given interval. The following description of Newton's method follows Bird and Wadler [1988], page 23:

```
type R = Double
newton :: (R → R) → R → R → R
newton f ε x = if abs fx < ε
               then x
               else if fx' ≠ 0 then newton f ε next
                   else newton f ε (x + ε)

where fx      = f x
      fx'     = undefined  -- f' x (derivative of f at x)
      next    = x - (fx / fx')
```

- (a) Implement Newton's method, using *Tri R* → *Tri R* for the type of the first argument. In other words, use the code above to implement

```
newtonTri :: (Tri R → Tri R) → R → R → R
```

in order to obtain the appropriate value for *f' x*.

(b) Test your implementation on the following functions:

```
test0 x = x^2           -- one (double) zero, in zero
test1 x = x^2 - one     -- two zeros, in +-one
test2 x = sin x         -- many, many zeros (in  $n * \pi$  for all  $n :: \mathbb{Z}$ )
test3 n x y = y^n - constTri x -- test3 n x, has zero in "nth roots of x"
    -- where constTri is the embedding of Const
```

For each of these functions, apply Newton's method to a number of starting points from a sensible interval. For example:

```
map (newton test1 0.001) [-2.0, -1.5 .. 2.0]
```

but be aware that the method might not always converge!

For debugging is advisable to implement *newton* in terms of *newtonList*, a minor variation which returns a list of the approximations encountered on the way to the final answer:

```
newton f  $\epsilon$  x = last (newtonList f  $\epsilon$  x)
newtonList f  $\epsilon$  x = x : if ... then [] else ...
```

3. We can find the optima of a twice-differentiable function on an interval by finding the zeros of its derivative on that interval, and checking the second derivative. If $f' x_0$ is zero, then
 - if $f'' x_0 < 0$, then x_0 is a maximum
 - if $f'' x_0 > 0$, then x_0 is a minimum
 - if $f'' x_0 = 0$, then, if $f'' (x_0 - \epsilon) * f'' (x_0 + \epsilon) < 0$ (i.e., f'' changes its sign in the neighbourhood of x_0), x_0 is an inflection point (not an optimum)
 - otherwise, we don't know

Use Newton's method to find the optima of the test functions from point 2. That is, implement a function

```
optim :: (Tri  $\mathbb{R}$   $\rightarrow$  Tri  $\mathbb{R}$ )  $\rightarrow$   $\mathbb{R}$   $\rightarrow$   $\mathbb{R}$   $\rightarrow$  Result  $\mathbb{R}$ 
```

so that *optim* $f \epsilon x$ uses Newton's method to find a zero of f' starting from x . If y is the result (i.e. $f' y$ is within ϵ of 0), then check the second derivative, returning *Maximum* y if $f'' y < 0$, *Minimum* y if $f'' y > 0$, and *Dunno* y if $f'' = 0$.

As before, use several starting points.

Hint: you might want to modify the code you've written for Newton's method at point 2.

Formalities

Submission: Assignments are to be submitted via Canvas

Deadline: 2021-03-05

Grading: Discussions with each of the teams during one of the slots 2021-03-08.

Skeleton code

Here is some useful Haskell code to start from, and the *Algebra* module (defining the numeric classes) is available on github.

```
{-# LANGUAGE FlexibleContexts, FlexibleInstances, TypeSynonymInstances #-}
import Prelude
    hiding ((+), (-), (*), (/), negate, recip, (^), fromInteger, sin, pi, cos, exp)
import DSLsofMath.Algebra
instance Additive a      => Additive (Tri a)      where (+) = addTri; zero = zeroTri
instance (Additive a, Multiplicative a)
    => Multiplicative (Tri a)  where (*) = mulTri; one = oneTri
instance AddGroup a      => AddGroup (Tri a)      where negate = negateTri
instance (AddGroup a, MulGroup a)
    => MulGroup (Tri a)      where recip = recipTri
    (addTri, zeroTri, mulTri, oneTri, negateTri, recipTri) = undefined
instance Transcendental a => Transcendental (Tri a) where
    pi = piTri; sin = sinTri; cos = cosTri; exp = expTri
    (piTri, sinTri, cosTri, expTri) = undefined
```

References

R. Bird and P. Wadler. *Introduction to Functional Programming, 1988*. Prentice-Hall, Englewood Cliffs, NJ, 1988.