As a motivating example, note that without *simplify* we have *derive* $(X :\!*\!: X)$ == $(Const\ 1 :\!*\!: X) :\!+\!: (X :\!*\!: Const\ 1)$, and that the syntax tree of the second derivative is twice that size.

4.10.1 Project: Optimisation using Newton's method

This subsection describes a larger exercise (or small project) you can use to practice what you have learnt so far. It is heavily based on Chapter 3 and Chapter 4 (the *FunExp* type, *eval*, *derive*, *D*, tupling, homomorphisms, *FD*, *apply*, ...) so it pays off to work through those parts carefully.

Part 1 The evaluation of the second derivative is given by

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

a) Let $P\ (h) =$ "h is a homomorphism from *FunExp* to $FunSem = \mathbb{R} \to \mathbb{R}$". Express P in logic and show $\neg\ P\ (eval'')$.

b) Given the types in the skeleton code below, define instances of the classes *Additive*, *AddGroup*, *Multiplicative*, *MulGroup*, and *Transcendental*, for *Tri a*. Test your results using algebraic identities like $sin\hat{\ }2 + cos\hat{\ }2 = const\ one$.

c) Define a homomorphism *evalDD* from *FunExp* to *FunTri a* (for any type *a* in the class *Transcendental*). You don't need to prove that it is a homomorphism in this part.

d) Show that *evalDD* is a homomorphism for the case of multiplication.

Part 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:

```
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    -- should be f' x (derivative of f at x)
        next  = x − (fx / fx')
```

a) Implement Newton's method, using $Tri\ \mathbb{R} \rightarrow Tri\ \mathbb{R}$ for the type of the first argument. In other words, use the code above to implement

$$newtonTri :: (Tri\ \mathbb{R} \rightarrow Tri\ \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$$

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

b) Test your implementation on the following functions:

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

Note that these functions can work at different types: $\mathbb{R} \rightarrow \mathbb{R}$ or $Dup\ \mathbb{R} \rightarrow Dup\ \mathbb{R}$ or $Tri\ \mathbb{R} \rightarrow Tri\ \mathbb{R}$, etc.

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:

$$newton\ f\ \epsilon\ x = last\ (newtonList\ f\ \epsilon\ x)$$
$$newtonList\ f\ \epsilon\ x = x : \textbf{if} \ldots \textbf{then}\ [\,]\ \textbf{else} \ldots$$

Part 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

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

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 to test if you get the expected behaviour.

Skeleton code Here is some useful skeleton Haskell code to start from, and the *Algebra* and *FunExp* modules are also available on github.

```
{-# LANGUAGE FlexibleContexts, FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}
module A2_Skeleton where
import Prelude hiding ((+), (-), (*), (/), negate, recip, (^),
                       π, sin, cos, exp, fromInteger, fromRational)
import DSLsofMath.Algebra
import DSLsofMath.FunExp
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)
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
  π = piTri; sin = sinTri; cos = cosTri; exp = expTri
(piTri, sinTri, cosTri, expTri) = undefined
```